Computational Intelligence Methods and Applications

Baochang Zhang
Tiancheng Wang
Sheng Xu
David Doermann

Neural Networks with Model Compression

Springer

Computational Intelligence Methods and Applications

The monographs and textbooks in this series explain methods developed in computational intelligence (including evolutionary computing, neural networks, and fuzzy systems), soft computing, statistics, and artificial intelligence, and their applications in domains such as heuristics and optimization; bioinformatics, computational biology, and biomedical engineering; image and signal processing, VLSI, and embedded system design; network design; process engineering; social networking; and data mining.

Baochang Zhang • Tiancheng Wang • Sheng Xu •
David Doermann

Neural Networks with Model Compression

Springer

Baochang Zhang
Institute of Artificial Intelligence
Beihang University
Beijing, China

Tiancheng Wang
Institute of Artificial Intelligence
Beihang University
Beijing, China

Sheng Xu
School of Automation Science and Electrical Engineering
Beihang University
Beijing, China

David Doermann
Department of Computer Science and Engineering
University at Buffalo, State University
Buffalo, NY, USA

ISSN 2510-1765 ISSN 2510-1773 (electronic)
Computational Intelligence Methods and Applications
ISBN 978-981-99-5067-6 ISBN 978-981-99-5068-3 (eBook)
https://doi.org/10.1007/978-981-99-5068-3

Preface

With the swift development of information technology, cloud computing with centralized data processing cannot meet the needs of applications that require processing massive amounts of data, and they can only be effectively used when privacy requires the data to remain at the front-end device. Thus, edge computing has become necessary to handle the data from embedded devices. Intelligent edge devices benefit many requirements within real-time unmanned aerial systems, industrial systems, and privacy-preserving applications.

In recent years, deep learning has been applied to different applications, dramatically improving many artificial intelligence (AI) tasks. However, the incomparable accuracy of deep learning models is achieved by paying the cost of hungry memory consumption and high computational complexity, which significantly impedes their deployment in edge devices with low memory resources. For example, the VGG-16 network can achieve 92.7% top-5 test accuracy on image classification tasks with the ImageNet dataset. Still, the entire network contains about 140 million 32-bit floating-point parameters, requiring more than 500 megabytes of storage space and performing 1.6×10^{10} floating-point operations. Yet, FPGA-based embedded devices typically have only a few thousand compute units, which cannot handle the millions of floating-point operations in standard deep neural network models. On the other hand, complex neural networks are often accompanied by slower computing speed and longer inference time, which are not allowed in applications with strict latency requirements, such as vehicle detection and tracking. Therefore, a natural thought is to perform model compression and acceleration in neural networks without significantly decreasing the model performance.

This book introduces the significant advancements of neural networks with model compression. While quantized operations can enhance the efficiency of neural networks, they typically result in a decrease in performance. In the last 5 years, many methods have been introduced to improve the performance of quantized neural networks. To better review these methods, we focus on six aspects: gradient approximation, quantization, structural design, loss design, optimization, and neural architecture search. We also review the applications of neural networks with model compression in visual and audio analysis. There are also other model compression techniques, such as model compression with network pruning, widely used in edge computing, which we introduce for completeness in this book. From our previous studies, network pruning and quantized neural networks can be used simultaneously to complement each other, whereas network pruning on quantized neural networks can further compress models and improve the generalization ability for many downstream applications.

Beijing, China	Baochang Zhang
Beijing, China	Tiancheng Wang
Beijing, China	Sheng Xu
Buffalo, NY, USA	David Doermann

Contents

1 Introduction 1
- 1.1 Background 1
- 1.2 Introduction of Deep Learning 2
- 1.3 Model Compression and Acceleration 4
- References 5

2 Binary Neural Networks 7
- 2.1 Introduction 7
- 2.2 Gradient Approximation 9
- 2.3 Quantization 10
- 2.4 Structural Design 14
- 2.5 Loss Design 17
- 2.6 Optimization 19
- 2.7 Algorithms for Binary Neural Networks 22
  - 2.7.1 BNN: Binary Neural Network 23
  - 2.7.2 XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks 24
  - 2.7.3 SA-BNN: State-Aware Binary Neural Network 26
  - 2.7.4 PCNN: Projection Convolutional Neural Networks 31
- References 44

3 Binary Neural Architecture Search 49
- 3.1 Introduction 49
- 3.2 Neural Architecture Search 50
  - 3.2.1 ABanditNAS: Anti-bandit for Neural Architecture Search 50
  - 3.2.2 IDARTS: Interactive Differentiable Architecture Search 59
  - 3.2.3 Fast and Unsupervised Neural Architecture Evolution for Visual Representation Learning 67
- 3.3 Binary Neural Architecture Search 76
  - 3.3.1 BNAS: Binarized Neural Architecture Search for Efficient Object Recognition 76

3.3.2 BDetNAS: A Fast Binarized Detection Neural Architecture Search 88
References 95

4 Quantization of Neural Networks 101
4.1 Introduction 101
4.2 Quantitative Arithmetic Principles 101
4.3 Uniform and Nonuniform Quantization 102
4.4 Symmetric and Asymmetric Quantization 103
4.5 Comparison of Different Quantization Methods 104
4.5.1 LSQ: Learned Step Size Quantization 104
4.5.2 TRQ: Ternary Neural Networks with Residual Quantization 108
4.5.3 OMPQ: Orthogonal Mixed Precision Quantization 119
References 127

5 Network Pruning 131
5.1 Introduction 131
5.2 Structured Pruning 133
5.3 Unstructured Pruning 134
5.4 Network Pruning 134
5.4.1 Efficient Structured Pruning Based on Deep Feature Stabilization 134
5.4.2 Toward Compact and Sparse CNNs via Expectation-Maximization 147
5.4.3 Pruning Multi-view Stereo Net for Efficient 3D Reconstruction 157
5.4.4 Cogradient Descent for Dependable Learning 168
5.5 Network Pruning on BNNs 189
5.5.1 Rectified Binary Convolutional Networks with Generative Adversarial Learning 189
5.5.2 BONN: Bayesian Optimized Binary Neural Network 198
References 213

6 Applications 219
6.1 Introduction 219
6.2 Image Classification 219
6.3 Speech Recognition 220
6.3.1 1-Bit WaveNet: Compression of a Generative Neural Network in Speech Recognition with Two Binarized Methods 220
6.4 Object Detection and Tracking 227
6.4.1 Data-Adaptive Binary Neural Networks for Efficient Object Detection and Recognition 228

6.4.2 Amplitude Suppression and Direction Activation in Networks for Faster Object Detection 238
6.4.3 Q-YOLO: Efficient Inference for Real-Time Object Detection 249
References 256

Chapter 1
Introduction

1.1 Background

Recently, there has been a significant increase in the complexity of deep learning models, with models becoming more and more intricate [2, 3, 7–10]. However, the hardware on which these models are deployed has not kept up with the increasing computational demands. Practical limitations such as latency, battery life, and temperature have created a significant gap between the computational requirements of these models and the available hardware resources.

To bridge this gap, network quantization has emerged as a popular approach [1, 4–6]. Network quantization involves mapping single-precision floating-point weights or activations to lower bit integers, leading to compression and acceleration of the model. One notable technique in this area is binary neural network (BNN), which is the simplest version of low-bit networks and has gained significant attention due to its highly compressed parameters and activation features [1]. Notably, the company Xnor.ai has become prominent for its work on BNNs. Founded in 2016, the company has raised substantial funding to develop tools that enable AI algorithms to run on devices rather than remote data centers. This approach allows for greater privacy and faster processing. Recently, Apple Inc. acquired Xnor.ai and plans to leverage BNN technology to enhance user privacy and accelerate processing on its devices.

Deep learning has gained significant importance due to its exceptional performance; however, it faces challenges in terms of large memory requirements and high computational demands, making it difficult to deploy on resource-constrained front-end devices. For instance, unmanned systems rely on UAVs as computing terminals with limited memory and computational resources, posing obstacles to real-time data processing using convolutional neural networks (CNNs). To address these efficiency concerns, binary neural networks (BNNs) have emerged as promising solutions for practical applications. BNNs are neural networks that binarize weights,

B. Zhang et al., *Neural Networks with Model Compression*,
Computational Intelligence Methods and Applications,
https://doi.org/10.1007/978-981-99-5068-3_1

offering improved storage and computation efficiency. Taking this approach further, 1-bit CNNs achieve extreme compression by binarizing both the weights and activations, reducing the model size and computational costs even further. Such highly compressed models are well-suited for front-end computing tasks. Alongside BNNs, other techniques like pruning neural networks involving quantization are widely utilized in edge computing.

This book comprehensively analyzes the latest advancements in model compression technologies specifically designed for front-end computing. It offers an extensive review and summary of existing research, categorized into binary neural networks, binary neural architecture search, quantization of neural networks, and network pruning. Furthermore, the book explores the practical applications of these techniques in computer vision and speech recognition, shedding light on their potential for future applications in edge computing.

1.2 Introduction of Deep Learning

Deep learning is a subset of machine learning that focuses on developing and applying artificial neural networks with multiple layers, also known as deep neural networks. It is inspired by the structure and function of the human brain, specifically the interconnectedness of neurons.

Deep learning models, also known as deep neural networks, comprise multiple layers of interconnected artificial neurons called units or nodes. These layers include an input layer, one or more hidden layers, and an output layer. Each unit in the network receives input signals, applies a mathematical transformation to them, and produces an output signal that is passed to the next layer. The weights associated with each connection between the units determine the strength and impact of the signals. The key features and concepts of deep learning are as follows:

Neural Network Architecture Deep learning models can have many architectures, depending on the task and data being addressed. Common architectures include feedforward neural networks, convolutional neural networks (CNNs) for image analysis, recurrent neural networks (RNNs) for sequence data, and transformers for natural language processing tasks.

Training Deep learning models learn from data through training. During training, the model is presented with a labeled dataset and adjusts its weights to minimize the difference between its predictions and the true labels. This optimization is achieved using an algorithm called back propagation (BP), which calculates the gradients of the model's performance concerning the weights and updates the weights accordingly. The process iterates until the model converges to a satisfactory level of performance.

Activation Functions Activation functions introduce nonlinearities to the neural network, allowing it to model complex relationships between inputs and outputs.

Common activation functions include sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU). They help the network learn nonlinear patterns and make the model more expressive.

Loss Functions Loss functions measure the discrepancy between the predicted outputs of the model and the true labels in the training data. They provide a quantitative measure of how well the model is performing. Common loss functions include mean squared error (MSE) for regression tasks and categorical cross-entropy for classification tasks.

Optimization Algorithms Optimization algorithms are used to update the neural network weights during training. Stochastic gradient descent (SGD) is a widely used optimization algorithm that iteratively adjusts the weights based on the gradients computed through back propagation. Variants of SGD, such as Adam and RMSprop, are also commonly employed to improve training efficiency and convergence.

Regularization Deep learning models are prone to overfitting, which occurs when the model becomes too specialized to the training data and performs poorly on unseen data. Regularization techniques, such as L1 and L2 regularization, dropout, and early stopping, are used to prevent overfitting and improve the model's generalization ability.

One of the key advantages of deep learning is its ability to automatically learn feature representations from raw data. Traditionally, in machine learning, feature engineering is a crucial step where domain experts manually extract relevant features from the data. In deep learning, the neural network learns these features directly from the raw data during training. This removes the need for manual feature engineering and allows the model to discover complex patterns and representations.

Deep learning has achieved remarkable success in various domains. In computer vision, deep neural networks have achieved state-of-the-art results in tasks such as image classification, object detection, and image segmentation. Deep learning has revolutionized machine translation, sentiment analysis, and speech recognition in natural language processing. It has also been applied to recommender systems, drug discovery, finance, and autonomous vehicles.

The success of deep learning is due to several factors. Firstly, the availability of large-scale datasets, such as ImageNet for computer vision or the Common Crawl dataset for natural language processing, has enabled the training of deep neural networks with millions or even billions of parameters. Secondly, advancements in computing power, particularly GPUs (graphics processing units), have accelerated the training process by performing parallel computations. Lastly, developing efficient algorithms like stochastic gradient descent and its variants has made it feasible to train deep neural networks effectively.

However, deep learning also presents challenges. Training deep neural networks requires substantial computational resources, and training times can be lengthy, especially for complex models. Deep learning models are also data-hungry and often require large labeled datasets, which may only sometimes be readily available.

Overfitting, where the model becomes too specialized to the training data and performs poorly on unseen data, is another challenge that needs to be addressed.

Researchers have been exploring techniques to overcome these challenges in recent years, such as transfer learning, which enables pre-training on large-scale datasets and fine-tuning on smaller task-specific datasets. There is also ongoing research on model compression, model acceleration, developing more efficient architectures, regularization techniques, and ways to leverage smaller datasets effectively.

Deep learning has revolutionized the field of artificial intelligence, enabling machines to learn and make intelligent decisions from vast amounts of data. Its ability to learn complex patterns and representations has significantly advanced in various domains.

1.3 Model Compression and Acceleration

Model compression and acceleration techniques reduce deep learning models' size and computational requirements, making them more efficient and practical for deployment on resource-constrained devices or in real-time applications. These techniques aim to balance model performance and efficiency, enabling faster inference times and reducing memory footprint while preserving or minimizing the loss in accuracy.

Pruning Pruning involves removing unimportant connections or weights from a trained neural network. It can be done in various ways, such as magnitude-based pruning, where weights below a certain threshold are pruned, or structured pruning, where entire filters or layers are pruned. Pruning reduces the number of parameters and connections in the network, resulting in a more compact model.

Quantization Quantization reduces the precision of weights and activations in the neural network from floating-point representation (32-bit) to lower bit representations (e.g., 8-bit or even lower). By using lower precision, quantization reduces memory usage and improves computational efficiency, as integer operations are typically faster than floating-point operations.

Low-Rank Factorization This technique reduces the number of parameters in a neural network by approximating weight matrices using low-rank factorization methods. Composing weight matrices into smaller matrices of lower rank can significantly reduce the number of parameters while maintaining reasonable accuracy.

Knowledge Distillation Knowledge distillation involves training a smaller "student" network to mimic the behavior of a larger "teacher" network. The teacher network provides soft targets (probability distributions) instead of hard labels during training. The student network learns to generalize from the teacher's knowledge, resulting in a compact model that can achieve comparable performance to the larger model.

Model Architecture Design Efficient model architecture design aims to create compact and lightweight models from scratch. Techniques like depth-wise separable convolutions, bottleneck layers, and skip connections can reduce the number of parameters and computational complexity while maintaining or improving performance.

Model Parallelism and Model Parallel Training Model parallelism divides a deep learning model across multiple devices or processors, allowing parallel computation and reducing the memory requirements for model inference. Similarly, model parallel training divides the training process across multiple devices, reducing the memory demand during training and enabling faster convergence.

Hardware Acceleration Hardware accelerators, such as graphics processing units (GPUs), tensor processing units (TPUs), or field-programmable gate arrays (FPGAs), are specialized devices designed to accelerate deep learning computations. These accelerators can significantly speed up the inference and training processes and improve energy efficiency.

These techniques can be used individually or in combination to achieve model compression and acceleration. The choice of techniques depends on the specific requirements of the deployment scenario and the trade-off between model size, computational efficiency, and accuracy.

Model compression and acceleration techniques have enabled the deployment of deep learning models on edge devices, mobile devices, and embedded systems, making real-time inference and applications like object detection, speech recognition, and natural language processing feasible in resource-constrained environments. These techniques have also paved the way for advancements in autonomous vehicles, the Internet of Things (IoT), and edge computing, where efficient and lightweight models are crucial for efficient and scalable deployment.

References

1. Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.
2. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
3. Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
4. Hyungjun Kim, Kyungsu Kim, Jinseok Kim, and Jae-Joon Kim. Binaryduo: Reducing gradient mismatch in binary activation network by coupling binary activations. In *International Conference on Learning Representations*.
5. Chunlei Liu, Wenrui Ding, Xin Xia, Baochang Zhang, Jiaxin Gu, Jianzhuang Liu, Rongrong Ji, and David Doermann. Circulant binary convolutional networks: Enhancing the performance of 1-bit DCNNs with circulant back propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2691–2699, 2019.

6. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.
7. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, pages 4510–4520, 2018.
8. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
9. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
10. Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *arXiv preprint arXiv:1707.01083*, 2017.

Chapter 2
Binary Neural Networks

2.1 Introduction

This chapter provides an overview of the most recent developments in binary neural network (BNN) technologies, with a particular focus on their suitability for front-end, edge-based computing. The content includes a thorough examination and synthesis of current research, organized into various categories such as gradient approximation, quantization techniques, architectural considerations, loss functions, optimization methods, and binary neural architecture search. Moreover, the chapter delves into the real-world applications of BNNs in computer vision and speech recognition while also contemplating the promising future prospects of BNNs across diverse domains.

In this chapter, we conduct a comprehensive review of the noteworthy advancements in binary neural networks and 1-bit CNNs. While binarization operations offer improved efficiency, they often come at the cost of reduced performance. However, over the past 5 years, several techniques have emerged to enhance the performance of binary neural networks significantly. To facilitate a comprehensive review of these methods, we categorize them into six key aspects: gradient approximation, quantization, structural design, loss design, optimization, and binary neural architecture search.

Additionally, we delve into the applications of BNNs in object detection, object tracking, and audio analysis, assessing their efficacy and potential in these specific domains. By presenting a holistic examination of the recent advancements and practical use-cases of binary neural networks, we aim to shed light on the promising future prospects of this technology.

BinaryConnect [12] was the first work attempting to confine weights to either $+1$ or -1 during propagation without binarizing the inputs. Binary operations

B. Zhang et al., *Neural Networks with Model Compression*,
Computational Intelligence Methods and Applications,
https://doi.org/10.1007/978-981-99-5068-3_2

possess simplicity and ease of comprehension. One of the approaches to binarize convolutional neural networks (CNNs) involves the utilization of a sign function:

$$\omega_b = \begin{cases} +1, \, if \quad \omega \geq 0 \\ -1, \, otherwise \end{cases}, \tag{2.1}$$

where ω_b is the binarized weight and ω the real-valued weight. A second way is to binarize scholastically:

$$\omega_b = \begin{cases} +1, \, with \quad probability \quad p = \sigma(\omega) \\ -1, \quad with \quad probability \quad 1 - p \end{cases}, \tag{2.2}$$

where σ is the "hard sigmoid" function. The training process for binary neural networks differs slightly from that of full-precision neural networks. During forward propagation, binary neural networks employ binarized weights instead of full-precision weights, while backward propagation follows conventional methods. The gradient $\frac{\partial C}{\partial \omega_b}$ (where C is the cost function) needs to be calculated and then combined with the learning rate to directly update the full-precision weights.

BinaryNet [25] extends beyond BinaryConnect by not only binarizing the weights but also quantizing the activations. To enforce both weights and activations to be either +1 or -1, BinaryNet introduces two methods. Additionally, it incorporates several modifications to accommodate binary activations. Firstly, it implements shift-based batch normalization (SBN) to avoid additional multiplications. Secondly, it employs shift-based AdaMax instead of the ADAM learning rule, which reduces the number of multiplications. The third modification concerns the operation performed on the input of the first layer, though specific details are not provided in this statement. For continuous-valued inputs of the first layer, BinaryNet represents them as fixed-point numbers with m bits of precision.

While BinaryConnect and BinaryNet demonstrate promising performance on representative datasets (as shown in Table 5.1), they struggle to perform well on larger datasets. The constraint of weights to +1 and -1 hinders effective learning. Therefore, new methods for training binary neural networks and 1-bit networks need to be developed to address these limitations. It is worth noting that QNN (quantized neural networks) [26] proposed training neural networks with extremely low-bit weights and activations, but the specific details of QNN are omitted in this review as we primarily focus on binary networks.

Wang et al. [60] proposed binarized deep neural networks (BDNNs) for image classification tasks, where all the values and operations in the network are binarized. While BinaryNet deals with convolutional neural networks, BDNNs target essential artificial neural networks consisting of full-connection layers. Bitwise neural networks [31] also present a completely bitwise network where all participating variables are bipolar binaries.

2.2 Gradient Approximation

In BNNs and 1-bit networks, the parameter updates involve full-precision weights using the gradient $\frac{\partial C}{\partial \omega_b}$. However, during forward propagation, a sign function is applied between full-precision and binarized weights. Consequently, the gradient of the sign function must be considered when updating the full-precision weights. As the derivative of the sign function is zero almost everywhere and becomes infinite at zero points, approximations using derivable functions are commonly employed to effectively handle the update process.

The first solution for addressing this issue in a 1-bit network was introduced by BinaryNet [25]. Assuming that an estimator of g_q for the gradient $\frac{\partial C}{\partial q}$, where q is $Sign(r)$, has been obtained, the straight-through estimator of $\frac{\partial C}{\partial r}$ is simply:

$$g_r = g_q 1_{|r| \leq 1}, \tag{2.3}$$

where $1_{|r| \leq 1}$ equals 1 when $|r| \leq 1$. And it equals 0 in other cases. It can also be seen as propagating the gradient through the hard $tanh$, which is a piecewise-linear activation function.

The Bi-Real Net [44] addresses the approximation of the derivative of the sign function for activations in binary neural networks. Instead of using $Htanh$ [25] for this purpose, the Bi-Real Net employs a piecewise polynomial function, resulting in a more accurate approximation.

Furthermore, the Bi-Real Net introduces a magnitude-aware gradient for weights. In traditional binary neural networks, the gradient $\frac{\partial C}{\partial W}$ is solely determined by the sign of weights and is independent of their magnitude. To enhance the learning process, the Bi-Real Net replaces the sign function with a magnitude-aware function, allowing the model to take into account both the sign and magnitude of weights during parameter updates. This approach contributes to more effective and fine-grained weight updates, leading to improved overall performance in binary neural networks.

Xu et al. [72] propose a higher-order approximation for weight binarization in binary neural networks. They use a long-tailed approximation for activation binarization, striking a balance between tight approximation and smooth back propagation.

In DSQ [17], a differentiable soft quantization function is introduced to approximate the standard binary and uniform quantization process. This function uses hyperbolic tangent functions to gradually approach the staircase function, specifically for low-bit quantization (similar to the sign function in 1-bit CNN). The binary DSQ function is as follows:

$$Q_s(x) = \begin{cases} -1, & x < -1 \\ 1, & x > 1 \\ stanh(kx), & otherwise \end{cases}, \tag{2.4}$$

with

$$k = \frac{1}{2} log(\frac{2}{\alpha} - 1), s = \frac{1}{1-\alpha}. \tag{2.5}$$

DSQ approximates uniform quantization well, especially with a small α, making it valuable for training high-accuracy quantized models. Being differentiable, DSQ allows for smooth parameter updates, contributing to improved accuracy compared to non-differentiable methods.

In summary, the methods discussed introduce differentiable functions to approximate the sign function in BinaryConnect. This allows for more accurate computation of gradients during training. As a result, binary neural networks and 1-bit networks converge more easily during the training process, leading to improved network performance and higher accuracy. These differentiable approximations have significantly advanced the field of binary neural networks and made them more practical and effective for various applications.

2.3 Quantization

BinaryConnect and BinaryNet use simple quantization methods where the binary weights are generated by taking the sign of full-precision weights after their update. However, this approach may lead to significant quantization errors.

Before discussing new methods to improve the quantization process, let's clarify the notations used in XNOR-Net [53] for each layer in a convolutional neural network. For each layer in a convolutional neural network, I is the input, W is the weight filter, B is the binarized weight (+-1), and H is the binarized input.

In their work, Rastegari et al. [53] introduce binary weight networks (BWN) and XNOR-Networks. BWN approximates weights with binary values, representing a variation of binary neural networks. On the other hand, XNOR-Networks binarize both weights and activation bits, making it a 1-bit network. Both networks utilize a scaling factor.

In BWN, the real-valued weight filter W is estimated using a binary filter B and a scaling factor α. The convolutional operation is then approximated as follows:

$$I * W \approx (I \oplus B)\alpha, \tag{2.6}$$

where $\oplus$ indicates a convolution without multiplication. By introducing the scaling factor, binary weight filters reduce memory usage by a factor of $32\times$ compared to single-precision filters. To ensure W is approximately equal to αB, BWN defines an optimization problem, and the optimal solution is:

$$B^* = sign(W), \tag{2.7}$$

$$\alpha^* = \frac{W^T sign(W)}{n} = \frac{\sum |W_i|}{n} = \frac{1}{n} \| W_r \|_{l_1}. \tag{2.8}$$

Indeed, in both BWN and XNOR-Networks, the optimal estimation of binary weight filters involves taking the sign of weight values. The optimal scaling factor, α, for BWN is the absolute average of the absolute weight values. This scaling factor is crucial in the calculation of gradients during back propagation, allowing for effective weight updates.

For XNOR-Networks, another scaling factor, β, is used when binarizing the input I into H. The core idea of XNOR-Networks is similar to BWN, but the introduction of β during activation binarization provides additional optimization benefits. The experiments demonstrate that this approach significantly outperforms BinaryConnect and BNN on ImageNet.

In Xu et al.'s work [72], a trainable scaling factor is defined for both weights and activations, enhancing the adaptability and performance of quantized neural networks.

LQ-Nets [76] quantize both weights and activations using arbitrary bit widths, including 1 bit. The learnable nature of the quantizers allows them to be compatible with bitwise operations, preserving the fast inference benefits of properly quantized neural networks.

Based on XNOR-Net [53], HORQ [37] introduces a high-order binarization scheme to achieve a more accurate approximation while retaining the advantages of binary operations. High-order residual quantization (HORQ) calculates the residual error and then performs additional thresholding operations to further approximate the residual. This binary approximation of the residual can be considered a higher-order binary input. Similar to XNOR-Net, HORQ defines the first-order residual tensor $R_1(x)$ by computing the difference between the real input and the first-order binary quantization:

$$R_1(x) = X - \beta_1 H_1 \approx \beta_2 H_2, \tag{2.9}$$

where $R_1(x)$ is a real value tensor. By this analogy, $R_2(x)$ can be seen as the second-order residual tensor, and $\beta_3 H_3$ also approximates it. After recursively performing the above operations, they obtain order-K residual quantization:

$$X = \sum_{i=1}^{K} \beta_i H_i. \tag{2.10}$$

During the training of the HORQ network, the input tensor can be reshaped into a matrix, allowing it to be expressed as any order of residual quantization. By considering higher-order residual approximations, HORQ-Net achieves a more accurate representation of binary values. Experimental results demonstrate that HORQ-Net outperforms XNOR-Net in terms of accuracy on the CIFAR dataset.

ABC-Net [38] is another network designed to improve the performance of binary networks. ABC-Net approximates the full-precision weight filter W with a linear combination of M binary filters $B_1, B_2, \ldots, B_M \in \{+1, -1\}$ such that $W \approx \alpha_1\beta_1 + \ldots + \alpha_M\beta_M$. These binary filters are fixed as follows:

$$B_i = F_{u_i}(W) := sign(\bar{W} + u_i std(W)), i = 1, 2, \ldots, M, \tag{2.11}$$

where $\bar{W}$ and $std(W)$ are the mean and standard derivation of W. For activations, ABC-Net employs multiple binary activations to alleviate information loss. Like the binarization weights, the real activation I is estimated using a linear combination of N activations $A_1, A_2, \ldots, A_N$ such that $I = \beta_1 A_1 + \ldots + \beta_N A_N$, where

$$A_1, A_2, \ldots, A_N = H_{v_1}(R), H_{v_2}(R), \ldots, H_{v_N}(R). \tag{2.12}$$

$H(R)$ in Eq. 2.12 is a binary function, h is a bounded activation function, I is the indicator function, and v is a shift parameter. Unlike the input weights, the parameters β and v are trainable. Without explicit linear regression, the network tunes $\beta_n' s$ and $v_n' s$ during training and is fixed for testing. They are expected to learn and utilize the statistical features of full-precision activations.

Ternary-binary network (TBN) [57] is a convolutional neural network with ternary inputs and binary weights. It leverages accelerated ternary-binary matrix multiplication, using efficient operations like XOR, AND, and bit count commonly found in standard CNNs. TBN achieves an optimal trade-off between memory, efficiency, and performance. Wang et al. [59] propose a two-step quantization framework (TSQ) that decomposes network quantization into two stages: code learning and transformation function learning based on the learned codes. TSQ is mainly designed for 2-bit neural networks.

LBCNN [28] introduces a local binary convolution (LBC) layer, inspired by local binary patterns (LBP) used in image descriptors, especially in face recognition. The LBC layer comprises fixed, sparse, and predefined binary convolutional filters that remain unchanged during training. It includes a nonlinear activation function and learnable linear weights. The linear weights combine the activated filter responses, approximating the corresponding activated filter responses of a standard convolutional layer. The LBC layer significantly reduces the number of learnable parameters, offering parameter savings of $9x$ to $169x$ compared to a standard convolutional layer. Additionally, due to the sparse and binary nature of the weights, it results in up to $169x$ savings in model size when compared to conventional convolutions.

In MCN [61], modulation filters (M-Filters) are introduced to recover binarized filters. M-Filters are designed to approximate unbinarized convolutional filters within an end-to-end framework. Each layer shares only one M-Filter, leading to a significant reduction in model size. To reconstruct the unbinarized filters, MCN employs a modulated process based on the M-Filters and binarized filters. Figure 2.1 illustrates an example of the modulation process. In this example, the M-Filter has four planes, each expanding to a 3D matrix according to the channels of the binarized filter. The reconstructed filter Q is obtained through the $\circ$ operation between the binarized filter and each expanded M-Filter.

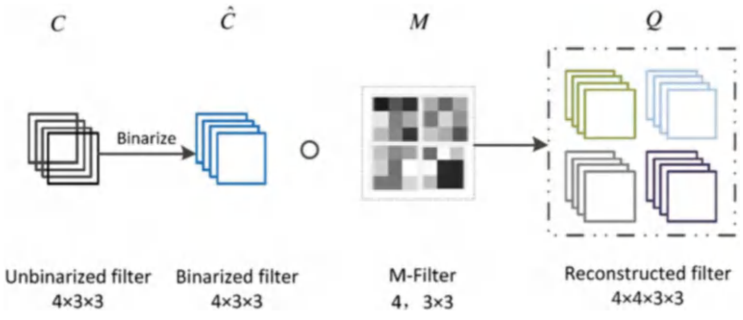


Fig. 2.1 Modulation process based on an M-Filter

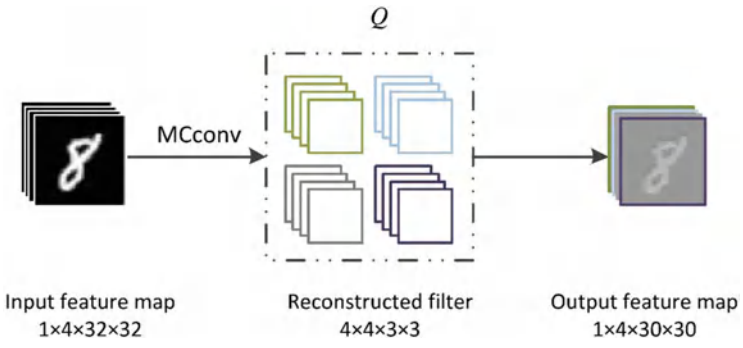


Fig. 2.2 MCNs' convolution

As depicted in Fig. 2.2, the reconstructed filters Q are utilized to compute the output feature maps F. Figure 2.2 shows four planes, resulting in four channels in the feature maps. The key advantage of MCN's convolution is that it maintains the same number of input and output channels for each feature map, facilitating module replication and easy implementation of MCNs.

Unlike previous approaches that independently binarize each filter, Bulat et al. [8] propose parameterizing the weight tensor of each layer using a matrix or tensor decomposition. The binarization process involves using a quantization function (e.g., sign function) for the reconstructed weights, while computation in the latent factorized space is performed in the real domain. This approach offers several advantages. First, the latent factorization enforces a coupling of filters before binarization, leading to a significant improvement in the accuracy of trained models. This coupling allows the model to capture more complex and fine-grained features, contributing to higher accuracy in tasks. Second, during training, each convolutional layer's binary weights are parametrized using a real-valued matrix or tensor decomposition. However, during inference, reconstructed (binary) weights are used, which retains the efficiency benefits of binary neural networks during the testing phase.

In contrast to previous approaches that use the same binary method for both weights and activations, Huang et al. [24] propose a different approach. They believe that the best performance for binarized neural networks can be achieved by applying different quantization methods to weights and activations. In their method, they simultaneously binarize the weights while quantizing the activations. This simultaneous approach aims to reduce bandwidth.

ReActNet [43] introduces a novel approach to binarized neural networks. It replaces the traditional sign function with ReAct-Sign and the PReLU function with ReAct-PReLU. These operations involve a simple channel-wise reshaping and shifting operation for the activation distribution. In ReAct-Sign and ReAct-PReLU, the parameters can be updated during training, allowing the network to learn and adapt to the data. This feature makes ReActNet more flexible and capable of capturing complex patterns in the data, leading to improved performance compared to traditional binarized neural networks.

Compared to XNOR-Net [53], both HORQ-Net [37] and ABC-Net [38] use multiple binary weights and activations, leading to improved performance on binary tasks. However, this improvement comes at the cost of increased complexity, which goes against the initial intention of binary neural networks to be efficient and speedy. To address this challenge, new neural network architectures are continuously being explored. MCN [61] and LBCNN [28] propose innovative filters while quantizing parameters. Additionally, they introduce new loss functions to learn these additional filters.

2.4 Structural Design

Indeed, the fundamental structure of networks like BinaryConnect [12] and BinaryNet [25] closely resembles that of traditional convolutional neural networks, which may not be optimally suited for binary processing. As a result, researchers have sought to modify the architecture of binary neural networks to enhance their accuracy.

In XNOR-Net [53], the block structure in a typical CNN is changed to further decrease information loss due to binarization. A typical block in a CNN typically contains different layers in the following order: 1-Convolutional, 2-BatchNorm, 3-Activation, and 4-Pooling. Before binarization, the input is normalized to have zero means. This normalization step is crucial in minimizing quantization error during thresholding at zero. The order of the layers in XNOR-Net is shown in Fig. 2.3.

In the context of Bi-Real Net [44], the poor performance of 1-bit CNNs is attributed to their limited representation capacity. Representation capacity refers to the number of possible configurations of a variable, which could be a scalar, vector, matrix, or tensor. To address this limitation and increase the representation capability of 1-bit CNNs, Bi-Real Net proposes a straightforward shortcut. The shortcut in Bi-Real Net preserves the real-valued activations before the sign function, effectively increasing the network's representation capacity. The structure

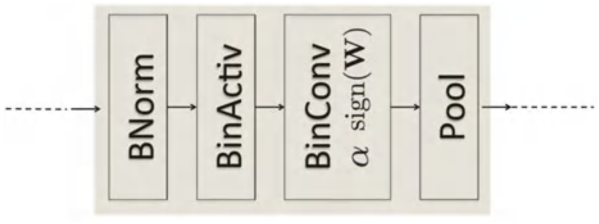


Fig. 2.3 A block in XNOR-Net

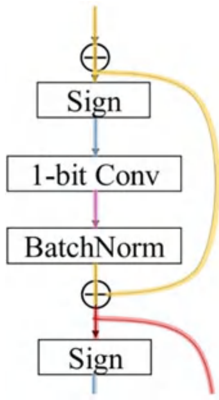


Fig. 2.4 1-bit CNN with shortcut

of the block is depicted as “Sign → 1-bit convolution → batch normalization → addition operator” in Fig. 2.4. The shortcut connects the input activations, which pass through the sign function in the current block, to the output activations after the batch normalization in the same block. These two sets of activations are then combined using an addition operator. The resulting combined activations are then passed to the sign function in the subsequent block.

By introducing this shortcut and preserving the real activations, Bi-Real Net seeks to enhance the expressiveness of 1-bit CNNs, ultimately improving their performance and accuracy in various tasks.

BinaryDenseNet [6] is a new binary neural network (BNN) architecture that addresses the main drawbacks of BNNs. It is based on DenseNets [23], which utilize shortcut connections to maintain the information flow throughout the depth of the network. However, the bottleneck design in DenseNets reduces the flow of information between layers, which is not suitable for BNNs due to their limited representation capacity. To overcome this limitation, BinaryDenseNet increases the growth rate or the number of blocks in the architecture to achieve satisfactory performance. Specifically, to maintain the same number of parameters as a given BinaryDenseNet, the growth rate is halved, and the number of blocks is doubled simultaneously. The architecture of BinaryDenseNet is shown in Fig. 2.5.

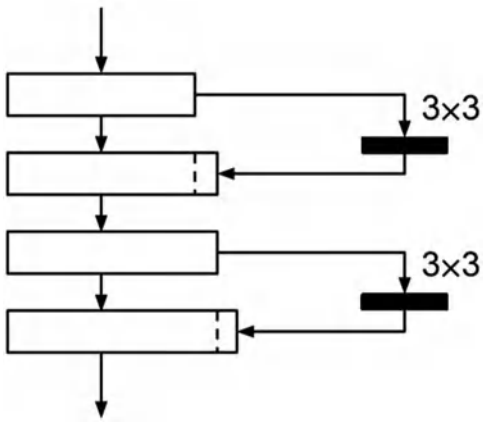


Fig. 2.5 BinaryDenseNet

MeliusNet [4] introduces a novel architecture that utilizes alternating Dense-Blocks to increase the feature capacity. Additionally, they propose an Improvement-Block to enhance the quality of the features. This approach enables 1-bit CNNs to achieve accuracy comparable to the popular compact network MobileNet-v1 while maintaining similar model size, number of operations, and accuracy. The building blocks of MeliusNet are shown in Fig. 2.6.

Group-Net [81] is another approach that enhances the performance of 1-bit CNNs through structural design. The inspiration behind Group-Net comes from the idea of a fixed number of binary digits representing a floating-point number in a computer. Group-Net proposes a novel approach to decompose a network into binary structures while ensuring that its representability is preserved. Instead of directly quantizing the network via "value decomposition," Group-Net leverages this structured approach.

Bulat et al. [9] were pioneers in exploring the impact of neural network binarization on localization tasks, such as human pose estimation and face alignment. They introduced a novel hierarchical, parallel, and multiscale residual architecture that leads to remarkable performance improvements over the standard bottleneck block, all while keeping the number of parameters unchanged. This achievement effectively bridges the gap between the original network and its binarized version. The new architecture introduced by Bulat et al. enhances the size of the receptive field, which enables the network to capture more context from the input data. Additionally, it improves the gradient flow within the network, leading to more efficient and effective learning.

LightNN [15] is a novel model that replaces multiplications in traditional neural networks with efficient shift and add operations. This innovative approach forms a new kind of model that significantly reduces the computational complexity while maintaining high accuracy.

In this section, we have discussed several works that modify the structure of binary neural networks, leading to improved performance and convergence. XNOR-Net and Bi-Real Net make subtle adjustments to the original networks to enhance

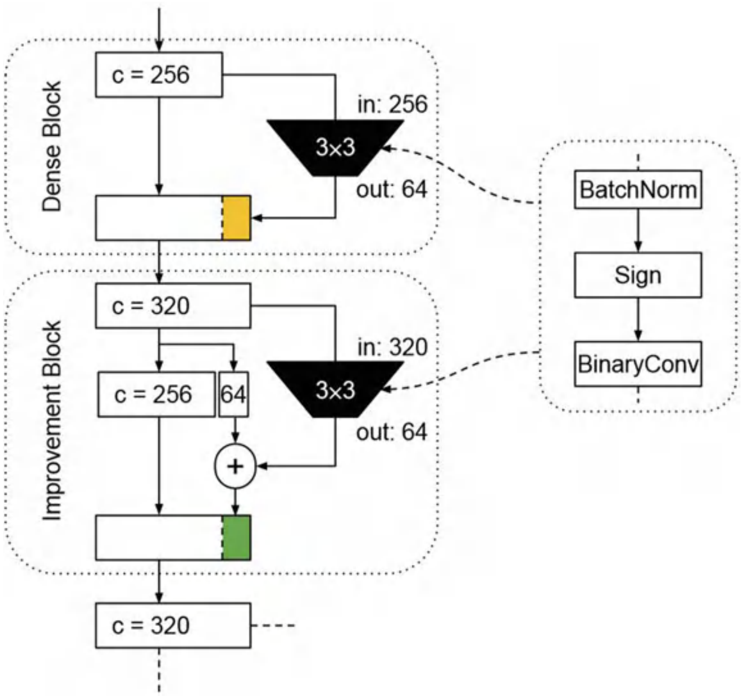


Fig. 2.6 Building blocks of MeliusNet (c denotes the number of channels in the feature map)

their representation capacity. On the other hand, MCN introduces new filters and convolutional operations to improve the overall accuracy of the network. Moreover, the loss function is also adapted to incorporate the new filters, which will be further elaborated in Sect. 2.5.

2.5 Loss Design

In binary neural networks (BNNs), the loss function plays a crucial role in estimating the difference between the actual and predicted values of the model. While classical loss functions like least squares loss and cross-entropy loss are commonly used in standard neural networks for classification and regression tasks, specific loss functions have been developed to suit the unique requirements of BNNs.

In MCNs [61], a novel loss function is introduced, which combines three components: filter loss, center loss, and softmax loss, in an end-to-end framework. The overall loss function in MCNs is composed of two main parts:

$$L = L_M + L_S. \tag{2.13}$$

The first part L_M is:

$$L_M = \frac{\theta}{2}\sum_{i,l}\left\|C_i^l - \hat{C}_i^l \circ M^l\right\|^2 + \frac{\lambda}{2}\sum_m \left\| f_m(\hat{C}, \mathbf{M}) - \bar{f}(\hat{C}, \mathbf{M})\right\|^2, \tag{2.14}$$

where C is the full-precision weights, $\hat{C}$ is the binarized weights, M is the M-Filters defined in Sect. 4.5.3, f_m denotes the feature map of the last convolutional layer for the mth sample, and $\bar{f}$ denotes the class-specific mean feature map of previous samples. The first entry of L_M represents the filter loss, while the second entry calculates the center loss using a conventional loss function, such as the softmax loss.

In PCNNs (projection convolutional neural networks) [19], a novel projection loss is introduced for discrete back propagation. It defines the quantization of the input variable as a projection onto a set, enabling the use of a projection loss for optimization.

BONNs (Bayesian-optimized 1-bit CNNs) [77] propose a Bayesian-optimized 1-bit CNN model, aiming to significantly improve the performance of 1-bit CNNs. BONNs incorporate prior distributions of full-precision kernels, features, and filters into a Bayesian framework to construct 1-bit CNNs comprehensively in an end-to-end manner. In BONNs, the quantization error is denoted as y, and the full-precision weights as x. To minimize the reconstructed error, they maximize $p(x|y)$, optimizing x for quantization. This optimization problem can be converted to a maximum a posteriori (MAP) since the distribution of x is known. For feature quantization, a similar method is employed; the Bayesian loss is as follows:

$$\begin{aligned} L_B = \frac{\lambda}{2}\sum_{l=1}^{l}\sum_{i=1}^{C_o^l}\sum_{n=1}^{C_i^l}\{&\left\|\hat{k}_n^{l,i} - w^l \circ k_n^{l,i}\right\|_2^2 \\ &+ v(k_{n+}^{l,i} - \mu_{i+}^l)^T(\Psi_{i+}^l)^{-1}(k_{n+}^{l,i} - \mu_{i+}^l) \\ &+ v(k_{n-}^{l,i} - \mu_{i-}^l)^T(\Psi_{i-}^l)^{-1}(k_{n-}^{l,i} - \mu_{i-}^l) \\ &v log(det(\Psi^l))\} + \frac{\theta}{2}\sum_{m=1}^{M}\{\left\| f_m - c_m\right\|^2 \\ &+ \sum_{n=1}^{N_f}\left[\sigma_{m,n}^{-2}(f_{m,n} - c_{m,n})^2 + log(\sigma_{m,n}^2)\right]\}, \end{aligned} \tag{2.15}$$

where k is the full-precision kernels, w is the reconstructed matrix, v is the variance of y, μ is the mean of the kernels, Ψ is the covariance of the kernels, f_m are the features of class m, and c is the mean of f_m.

In the work by Zheng et al. [78], they introduce a novel quantization loss that measures the discrepancy between binary weights and learned real values. The theoretical analysis provided by Zheng et al. demonstrates the importance of minimizing this weight quantization loss to enhance the performance of binarized neural networks. On the other hand, Ding et al. [14] propose the use of a distribution loss to explicitly regulate the activation flow within the network. They develop a systematic framework to formulate this distribution loss, which helps in guiding the training process effectively. The empirical results from Ding et al.'s work illustrate that their proposed distribution loss is robust in terms of selecting training hyperparameters.

These methods all aim to minimize the error and information loss of quantization, which improves the compactness and capacity of 1-bit CNNs.

2.6 Optimization

Absolutely, researchers have been actively seeking new training methods to enhance the performance of binary neural networks (BNNs) and overcome their inherent limitations. These methods are aimed at improving the effectiveness of BNNs across various tasks and applications. One approach involves integrating techniques from other fields into BNNs. By borrowing insights and methods from different domains, researchers aim to augment the capabilities and performance of BNNs. This cross-disciplinary approach allows for innovative solutions that can address specific challenges faced by binary networks. Moreover, improving the training process is a key focus for enhancing BNNs. Researchers are exploring modifications to the optimization algorithms used in classical BNNs. These adaptations target the optimization process to achieve better convergence, stability, and overall performance.

The work by Sari et al. [60] sheds light on the importance of the BatchNorm layer in the training process of binary neural networks (BNNs). They demonstrate that BatchNorm plays a crucial role in preventing exploding gradients, which can be a significant issue in BNNs due to the binary nature of the weights. Their findings also suggest that the standard initialization methods commonly used in full-precision networks may not be suitable for BNNs, highlighting the need for specialized techniques to handle weight initialization in binary networks. Additionally, they provide insights into the components of BatchNorm, showing that only minibatch centering is necessary, which can simplify the implementation of BatchNorm in BNNs. On the other hand, the experiments conducted by Alizadeh et al. [1] offer valuable empirical evidence regarding common tricks used in binary training models. They show that techniques like gradient and weight clipping, often employed to stabilize training in BNNs, are primarily needed during the final stages of training to achieve the best performance.

XNOR-Net++ [10] presents an innovative training algorithm for 1-bit convolutional neural networks (CNNs), building upon the foundation of XNOR-Net. In XNOR-Net++, the authors introduce a novel approach to combine activation and weight scaling factors into a single scalar, which is learned discriminatively through back propagation. By unifying these scaling factors, the method aims to streamline the training process and enhance the efficiency of 1-bit CNNs. Additionally, XNOR-Net++ explores various strategies to construct the shape of the scale factors while ensuring that the computational budget remains fixed.

The work by Leng et al. [35] draws inspiration from the alternating direction method of multipliers (ADMM) to address the challenges of training binary neural networks. By leveraging the principles of ADMM, they propose a novel approach to decouple the continuous parameters from the discrete constraints in the network. This decoupling allows them to break down the original complex optimization problem into several subproblems, each with its own set of constraints. To solve these subproblems efficiently, Leng et al. employ different gradient and iterative quantization algorithms. By doing so, they achieve considerably faster convergence rates compared to traditional optimization methods used in binary neural networks.

In the work of deterministic binary filters (DBFs) [56], the researchers propose a novel approach to learn weighted coefficients of predefined orthogonal binary bases instead of directly learning the convolutional filters, as is typically done in conventional methods. DBFs generate filters by representing them as a linear combination of orthogonal binary codes. These orthogonal binary bases are predefined, and the learning process focuses on finding the optimal weighted coefficients for these bases. By doing so, the filters can be efficiently generated in real time.

BinaryRelax [75] presents a two-phase algorithm for training convolutional neural networks (CNNs) with quantized weights, including binary weights. The goal is to overcome the challenges posed by hard constraints on binary weights during training. In the first phase, BinaryRelax relaxes the hard constraint of binary weights into a continuous regularizer using the Moreau envelope [48]. This regularization term is defined as the squared Euclidean distance between the weights and the set of quantized weights. By gradually increasing the regularization parameter, BinaryRelax narrows the gap between the continuous weights and the quantized state, effectively transitioning toward a binary solution. In the second phase, BinaryRelax introduces the same quantization scheme but with a small learning rate. This guarantees that the weights eventually converge to fully quantized binary values.

CBCNs [41] propose a novel approach to enhance the capacity of binarized convolutional features using circulant filters (CiFs) and circulant binary convolution (CBConv). CiFs are 4D tensors of size $K \times K \times H \times H$, generated by applying a circulant transfer matrix M to a learned filter. The matrix M rotates the learned filter at different angles, effectively expanding its representation capacity. To create a CiF, the original 2D $H \times H$ learned filter is transformed into a 3D tensor by replicating it three times and concatenating them. This 3D tensor is then combined with the circulant transfer matrix M to form the 4D CiF. By utilizing circulant filters and circulant binary convolution, CBCNs can improve the representation capacity of binarized neural networks without altering the model size (Fig. 2.7).

Rectified binary convolutional networks (RBCNs) [40] introduce a novel approach to train 1-bit binary networks using a generative adversarial network (GAN). The training process involves using the guidance of the corresponding full-precision model, which leads to significant performance improvements in

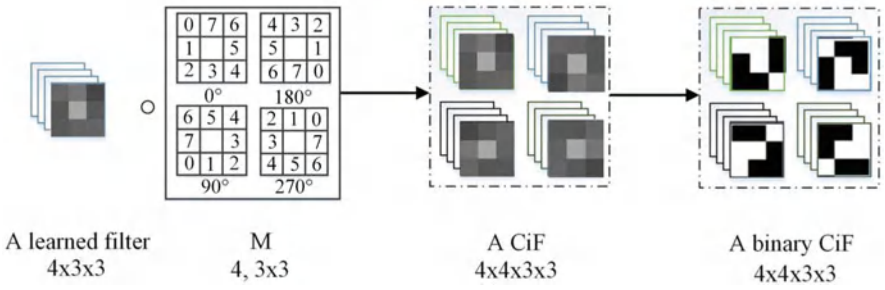


Fig. 2.7 The generation of CiF

1-bit CNNs. The key innovation in RBCNs is the incorporation of rectified convolutional layers, which are designed to be generic and flexible. These layers can be easily integrated into existing deep convolutional neural networks (DCNNs) like WideResNets and ResNets.

Martinez et al. [45] focus on minimizing the discrepancy between the binary output and the corresponding real-valued convolution in 1-bit CNNs. They propose a real-to-binary attention matching approach that is tailored for training these networks. Additionally, they introduce a progressive bridging strategy to reduce the architectural gap between real and binary networks through a sequence of teacher-student pairs.

In contrast, Bethge et al. [5] take a different approach by directly training a binary network from scratch, without relying on pre-trained full-precision models or other standard methods. Their training implementation is based on the BMXNet framework [74].

Helwegen et al. [22] highlight that latent weights with real values in binary neural networks serve a different purpose compared to weights in real-valued networks. They propose the binary optimizer (Bop), specifically designed for BNNs, to handle the unique characteristics of binary weights effectively during the optimization process.

BinaryDuo [30] presents a novel training scheme for binary activation networks by coupling two binary activations into a ternary activation during training. They achieve this by first decoupling a ternary activation into two binary activations, effectively doubling the number of weights. However, to maintain the parameter size of the decoupled model and the baseline model, they reduce the coupled ternary model. The independent update of each weight after decoupling allows for better optimization, as the two weights no longer share the same value.

BENN [80] leverages classical ensemble methods to enhance the performance of 1-bit CNNs. While ensemble techniques were traditionally believed to have limited impact on robust classifiers like deep neural networks, BENN's analysis and experiments demonstrate that ensembles are exceptionally effective in boosting BNNs. The ensemble strategies used in BENN draw from various works such as [7, 11, 49].

Table 2.1 Experimental results of some famous binary methods on ImageNet

Methods	Weights	Activations	Model	Binarized accuracy		Full-precision accuracy	
				Top 1	Top 5	Top 1	Top 5
XNOR-Net [53]	Binary	Binary	ResNet-18	51.2	73.2	69.3	89.2
ABC-Net [38]	Binary	Binary	ResNet-50	70.1	89.7	76.1	92.8
LBCNN [27]	Binary	–	–	62.43[a]	–	64.94	–
Bi-Real Net [44]	Binary	Binary	ResNet-34	62.2	83.9	73.3	91.3
RBCN [40]	Binary	Binary	ResNet-18	59.5	81.6	69.3	89.2
BinaryDenseNet [6]	–	–	–	62.5	83.9	–	–

[a] 13×13 Filter

TentacleNet [47] builds on the theory of ensemble learning and makes further advancements beyond BENN. TentacleNet demonstrates that binary ensembles can achieve high accuracy while requiring fewer computational resources.

BayesBiNN [46] adopts a principled approach to discrete optimization by using a distribution over the binary variable. They introduce a Bayesian learning rule [29] to estimate a Bernoulli approximation to the posterior, resulting in a principled method for dealing with binary neural networks (Table 2.1).

2.7 Algorithms for Binary Neural Networks

Binarization, the most extreme form of quantization, is the main focus of this book. It involves representing data using only one bit, either -1 (or 0) or $+1$, resulting in 1-bit quantization. Both weights and activations in a binary neural network can be compressed into a single bit, leading to significant memory savings and hardware-friendly advantages, such as faster execution, reduced memory consumption, and improved power efficiency. Groundbreaking works like BNN [25] and XNOR-Net [53] have demonstrated the effectiveness of binarization, with XNOR-Net achieving up to 58.

Since the advent of binary neural networks, extensive research has been conducted in computer vision and machine learning fields [21, 42, 54], leading to their application in various tasks, including image classification [12, 44, 51, 53, 66, 73], object detection [64, 67, 70, 71], point cloud processing [50, 68], object reidentification [69], and more. Binarization's hardware-friendly benefits and practical applications have made it a promising area of research in recent years.

Binarizing a layer in a neural network helps identify its significance and impact on performance. If performance suffers after binarization, the layer is crucial for the network. This process aids explainable machine learning and verifies if binarization preserves essential information. Understanding binarized models contributes to improving binary neural networks.

Researchers have extensively studied model binarization to understand its behaviors and its relationship with the architecture of deep neural networks. Exploring binary neural networks helps answer fundamental questions about network topology and deep network functionality. Thorough exploration of binary neural network studies contributes to a better understanding of effective and reliable deep learning models. Notable works, like Bi-Real Net [44], have revealed how components in binary neural networks function, such as incorporating shortcuts to mitigate information loss due to binarization.

The structure of shortcuts in binary neural networks, similar to ResNet shortcuts, allows for better information flow between shallow and deep layers during both forward and backward propagation. This mechanism helps in avoiding issues like gradient disappearance and improves the overall performance of the network. Ensemble approaches in binary neural networks, like building weak classifier groups, can lead to performance improvements. However, they may also encounter overfitting problems. Understanding the trade-off between the number of neurons and the bit width is essential, as it can influence the network's performance. Interestingly, real-valued neurons may not be necessary in deep neural networks, aligning with the idea of biological neural networks. Reducing the bit width of specific layers can be an efficient method to examine the interpretability of deep neural networks. Investigating how sensitive different layers are to binarization is crucial in designing effective binary neural networks. Typically, the first and last layers in binary neural networks should be kept at higher precision since they play a more critical role in predicting the network's output. This section attempts to state the nature of binary neural networks by introducing some representative work.

2.7.1 BNN: Binary Neural Network

Given an N-layer CNN model, we denote its weight set as $\mathbf{W} = \{\mathbf{w}^n\}_{n=1}^N$ and the input feature map set as $\mathbf{A} = \{\mathbf{a}_{in}^n\}_{n=1}^N$. The $\mathbf{w}^n \in \mathbb{R}^{C_{out}^n \times C_{in}^n \times K^n \times K^n}$ and $\mathbf{a}_{in}^n \in \mathbb{R}^{C_{in}^n \times W_{in}^n \times H_{in}^n}$ are the convolutional weight and the input feature map in the n-th layer, where C_{in}^n, C_{out}^n, and K^n, respectively, represent the input channel number, the output channel number, and the kernel size. In addition, W_{in}^n and H_{in}^n are the width and height of the feature maps. Then, the convolutional outputs $\mathbf{a}_{out}^n$ can be technically formulated as:

$$\mathbf{a}_{out}^n = \mathbf{w}^n \otimes \mathbf{a}_{in}^n, \tag{2.16}$$

where $\otimes$ represents the convolution operation. In this book, we omit the nonlinear function for simplicity. Following the prior works [12, 25], binary neural network (BNN) intends to represent $\mathbf{w}^n$ and $\mathbf{a}^n$ in a binary discrete set as:

$$\mathbb{B} := \{-1(0), +1\}.$$

Thus, the 1-bit format of $\mathbf{w}^n$ and $\mathbf{a}^n$ is respectively $\mathbf{b}^{\mathbf{w}^n} \in \mathbb{B}^{C^n_{out} \times C^n_{in} \times K^n \times K^n}$ and $\mathbf{b}^{\mathbf{a}^n_{in}} \in \mathbb{B}^{C^n_{in} \times W^n_{in} \times H^n_{in}}$ such that the efficient XNOR and bit-count instructions can approximate the floating-point convolutional outputs as:

$$\mathbf{a}^n_{out} \approx \mathbf{b}^{\mathbf{w}^n} \odot \mathbf{b}^{\mathbf{a}^n_{in}}, \tag{2.17}$$

where $\circ$ represents channel-wise multiplication and $\odot$ denotes XNOR and bit-count instructions.

However, this quantization mode will cause the output amplitude to increase dramatically, different from the full-precision convolution calculation, and cause the homogenization of characteristics [53]. Several novel objects are proposed to address this issue, which will be introduced in the following.

2.7.2 XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks

The scaling factor was first proposed by XNOR-Net [53] to solve this problem. The weights and the inputs to the convolutional and fully connected layers in XNOR-Nets are approximated with binary values $\mathbb{B}$.

The XNOR-Net binarization approach seeks to identify the most accurate convolutional approximations. Specifically, XNOR-Net employs a scaling factor, which plays a vital role in the learning of BNNs, and improves the forward pass of BNNs as:

$$\mathbf{a}^n_{out} \approx \boldsymbol{\alpha}^n \circ (\mathbf{b}^{\mathbf{w}^n} \odot \mathbf{b}^{\mathbf{a}^n_{in}}), \tag{2.18}$$

where $\boldsymbol{\alpha}^n = \{\alpha^n_1, \alpha^n_2, \ldots, \alpha^n_{C^n_{out}}\} \in \mathbb{R}_+^{C^n_{out}}$ is known as the channel-wise scaling factor vector to mitigate the output gap between Eq. 2.16 and its approximation of Eq. 2.18. We denote $\mathcal{A} = \{\boldsymbol{\alpha}^n\}_{n=1}^N$. Since the weight values are binary, XNOR-Net can implement the convolution with additions and subtractions. In the following, we state the XNOR operation for a specific convolution layer, thus omitting the superscript n for simplicity. Most existing implementations simply follow earlier studies [44, 53]to optimize $\mathcal{A}$ based on nonparametric optimization as:

$$\boldsymbol{\alpha}^*, \mathbf{b}^{\mathbf{w}*} = \arg\min_{\boldsymbol{\alpha}, \mathbf{b}^{\mathbf{w}}} J(\boldsymbol{\alpha}, \mathbf{b}^{\mathbf{w}}), \tag{2.19}$$

$$J(\boldsymbol{\alpha}, \mathbf{b}^{\mathbf{w}}) = \|\mathbf{w} - \boldsymbol{\alpha}^n \circ \mathbf{b}^{\mathbf{w}}\|_2^2. \tag{2.20}$$

By expanding Eq. 2.20, we have:

$$J(\boldsymbol{\alpha}, \mathbf{b}^{\mathbf{w}}) = \boldsymbol{\alpha}^2 (\mathbf{b}^{\mathbf{w}})^{\mathsf{T}} \mathbf{b}^{\mathbf{w}} - 2\boldsymbol{\alpha} \circ \mathbf{w}^{\mathsf{T}} \mathbf{b}^{\mathbf{w}} + \mathbf{w}^{\mathsf{T}} \mathbf{w} \tag{2.21}$$

where $\mathbf{b}^{\mathbf{w}} \in \mathbb{B}$. Thus, $(\mathbf{b}^{\mathbf{w}})^{\mathsf{T}}\mathbf{b}^{\mathbf{w}} = C_{in} \times K \times K$. $\mathbf{w}^{\mathsf{T}}\mathbf{w}$ is also a constant due to $\mathbf{w}$ being a known variable. Thus, Eq. 2.21 can be rewritten as:

$$J(\boldsymbol{\alpha}, \mathbf{b}^{\mathbf{w}}) = \boldsymbol{\alpha}^2 \times C_{in} \times K \times K - 2\boldsymbol{\alpha} \circ \mathbf{w}^{\mathsf{T}}\mathbf{b}^{\mathbf{w}} + constant. \tag{2.22}$$

The optimal solution can be achieved by maximizing the following constrained optimization:

$$\mathbf{b}^{\mathbf{w}*} = \arg\max_{\mathbf{b}^{\mathbf{w}}} \mathbf{w}^{\mathsf{T}}\mathbf{b}^{\mathbf{w}}, s.t. \quad \mathbf{b}^{\mathbf{w}} \in \mathbb{B}, \tag{2.23}$$

which can be solved by the sign function:

$$\mathbf{b}^{\mathbf{w}_i} = \begin{cases} +1 \ \mathbf{w}_i \geq 0 \\ -1 \ \mathbf{w}_i \ < \ 0 \end{cases}$$

which is the optimal solution and is also widely used as a general solution to BNNs in the following numerous works [44]. To find the optimal value for the scaling factor $\boldsymbol{\alpha}^*$, we take the derivative of $J(\cdot)$ *w.r.t.* $\boldsymbol{\alpha}$ and set it to zero as:

$$\boldsymbol{\alpha}^* = \frac{\mathbf{w}^{\mathsf{T}}\mathbf{b}^{\mathbf{w}}}{C_{in}^n \times K^n \times K^n}. \tag{2.24}$$

By replacing $\mathbf{b}^{\mathbf{w}}$ with the sign function, we have that a closed-form solution of $\boldsymbol{\alpha}$ can be derived via the channel-wise absolute mean (CAM) as:

$$\boldsymbol{\alpha}_i = \frac{\|\mathbf{w}_{i,:,:,:}\|_1}{C_{in} \times K \times K} \tag{2.25}$$

$\boldsymbol{\alpha}_i = \frac{\|\mathbf{w}_{i,:,:,:}\|_1}{M}$. Therefore, the optimal estimation of a binary weight filter can be achieved simply by taking the sign of weight values. The optimal scaling factor is the average of the absolute weight values.

Based on the explicitly solved $\boldsymbol{\alpha}^*$, the training objective of the XNOR-Net-like BNNs is given in a bilevel form:

$$\begin{aligned} \mathbf{W}^* &= \arg\min_{\mathbf{W}} \mathcal{L}(\mathbf{W}; \mathcal{A}^*), \\ s.t. \ & \arg\min_{\boldsymbol{\alpha}^n, \mathbf{b}^{\mathbf{w}^n}} J(\boldsymbol{\alpha}, \mathbf{b}^{\mathbf{w}}), \end{aligned} \tag{2.26}$$

which is also known as hard binarization [44]. In the following, we show some variants of such a binarization function.

2.7.3 SA-BNN: State-Aware Binary Neural Network

Binary neural networks (BNNs) have received much attention due to their memory and computation efficiency. However, the sizable performance gap between BNNs and their full-precision counterparts hinders BNNs from being deployed in resource-constrained platforms. The challenge of the performance drop instinctively comes from the minimal binarization states $\{-1, 1\}$, which would bring many propagation errors in forward and backward procedures and lead to misleading weight update.

We want to suggest a method to make the training more efficient by suppressing the fluctuation of the weight update. Specifically, we find that existing methods [44, 53] possess the identical gradient amplitude for all quantization states $\{-1, 1\}$. According to our analysis, the frequent weight flip is more likely to happen in this case. The intuition here is about "whether we can calibrate the amplitude of the two states slightly distinctive to make their chance of weight flip different to increase the difficulty of frequent weight flip further?" Inspired by this, a novel state-aware binary neural network (SA-BNN) [39] equipped with a well-designed state-aware gradient is proposed in this paper. Expressly, we set separate learnable gradient coefficients for different states. In this way, the unnecessary weight update can be impeded efficiently. Besides, we lead to a theorem that the state-aware gradient can effectively mitigate the frequent weight flip problem, alleviating the ineffective update issue in BNN optimization.

2.7.3.1 Method

To suppress the frequent weight flips in BNNs, we propose the following state-aware gradient to stabilize the optimization:

$$\frac{\partial \mathscr{L}}{\partial x} = \begin{cases} \frac{\partial \mathscr{L}}{\partial \hat{x}}(\tau_{-1}\frac{\partial \hat{x}}{\partial x}) \text{ if } \hat{x} = -1 \\ \frac{\partial \mathscr{L}}{\partial \hat{x}}(\tau_{1}\frac{\partial \hat{x}}{\partial x}) \quad \text{otherwise} \end{cases}, \tag{2.27}$$

where $\tau_{-1}, \tau_1 \in \mathbb{R}$ are learnable coefficients, which are introduced on the activation gradients to distinctively treat the two states. We do not apply the distinguishable parameters $\tau = \{\tau_{-1}, \tau_1\}$ on the weight gradient ($\frac{\partial \mathscr{L}}{\partial w}$), since the weights themselves are learnable in the training process. It is equivalent to regard the state-aware coefficients τ and the weights as a whole. Therefore, we do not consider the state-aware gradient on the weights and instead focus on that on activation in the following. According to Eq. 2.27, we leverage an extra scale factor on the activation gradients for each binarization state to impose a mild constraint on the weight updating. When the two scale factors are equal ($\tau_{-1} = \tau_1$) , it reduces to the traditional weight updating with state-consistent gradients. Otherwise, it is the proposed state-aware gradient-based BNNs. Next, we analyze the difference between these two mechanisms.

Proposition 2.1 *The state-aware gradients ($|\tau_{-1}| \neq |\tau_1|$) can suppress frequent weight flip effectively compared with the corresponding state-consistent gradients ($|\tau_{-1}| = |\tau_1|$), leading to more stable training.*

Based on the gradient chain rule, the weight-updating procedure can be described as:

$$\begin{aligned}
w^{l,t+1} = w^{l,t} - \eta\frac{\partial \mathcal{L}}{\partial w^{l,t}} &= w^{l,t} - \eta\frac{\partial \mathcal{L}}{\partial \hat{x}^{l+1,t}}(\tau^{l+1,t}\frac{\partial \hat{x}^{l+1,t}}{\partial x^{l+1,t}})\frac{\partial x^{l+1,t}}{\partial \hat{w}^{l,t}}\frac{\partial \hat{w}^{l,t}}{\partial w^{l,t}} \\
&= w^{l,t} - \eta\frac{\partial \mathcal{L}}{\partial \hat{x}^{l+1,t}}(\tau^{l+1,t}\frac{\partial \hat{x}^{l+1,t}}{\partial x^{l+1,t}})\hat{x}^{l,t}\frac{\partial \hat{w}^{l,t}}{\partial w^{l,t}} \\
&= w^{l,t} - \tau^{l+1,t}b^{l,t},
\end{aligned} \tag{2.28}$$

where η is the learning rate, t represents the t-th iteration, and

$$b^{l,t} = \eta\frac{\partial \mathcal{L}}{\partial \hat{x}^{l+1,t}}\frac{\partial \hat{x}^{l+1,t}}{\partial x^{l+1,t}}\hat{x}^{l,t}\frac{\partial \hat{w}^{l,t}}{\partial w^{l,t}}.$$

For simplicity, we ignore the layer index superscript l in the following analysis. According to Eq. 2.28, to enable a weight flip (namely, let $\text{sign}(w^{t+1}) \neq \text{sign}(w^t)$), it requires to satisfy the constraints $\text{sign}(\tau^t b^t) = \text{sign}(w^t)$ and $|\tau^t b^t| > |w^t|$, where $|\cdot|$ represents the amplitude of the input. We assume the initial state $\text{sign}(w^t) = -1$, and the process is similar for the initial state $\text{sign}(w^t) = 1$.

1. If $|\tau_{-1}| = |\tau_1|$, the flip probability from the iteration t to $t+1$ is:

$$P(\text{sign}(w^t) \neq \text{sign}(w^{t+1})) = N_{|w^t|}/N, \tag{2.29}$$

where $N_{|w^t|}$ represents the total number of b^t satisfying $\text{sign}(\tau_1^t b^t) = \text{sign}(w^t)$ and $|\tau_1^t b^t| > |w^t|$, and N represents the total number of b. Similarly, the flip probability from the iteration $t+1$ to $t+2$ is

$$P(\text{sign}(w^{t+1}) \neq \text{sign}(w^{t+2})) = N_{|w^{t+1}|}/N, \tag{2.30}$$

where $N_{|w^{t+1}|}$ represents the total number of b^{t+1} satisfying $\text{sign}(\tau_{-1}^{t+1} b^{t+1}) = \text{sign}(w^{t+1})$ and $|\tau_{-1}^{t+1} b^{t+1}| > |w^{t+1}|$. Thus, the sequential flip probability from the iteration t to $t+2$ is:

$$\begin{aligned}
P((\text{sign}(w^t) \neq \text{sign}(w^{t+1})) &\cap (\text{sign}(w^{t+1}) \\
&\neq \text{sign}(w^{t+2}))) = (N_{|w^t|}N_{|w^{t+1}|})/N^2.
\end{aligned} \tag{2.31}$$

2. If $|\tau_{-1}| < |\tau_1|$, it remains the same flip probability from the iteration t to $t+1$ as Eq. 2.29. However, when considering the flip probability from iteration $t+1$

to $t+2$, the number of b^{t+1} that satisfying $|\tau_{-1}^{t+1}b^{t+1}| > |w^{t+1}|$, in this case, is less than that in the case of $|\tau_{-1}| = |\tau_1|$.

Therefore, the state-aware gradient (i.e., $|\tau_{-1}| < |\tau_1|$) has a lower probability of sequential weight flip compared with the conventional state-consistent methods (i.e., $|\tau_{-1}| = |\tau_1|$):

$$\begin{aligned} &P((\text{sign}(w^t) \neq \text{sign}(w^{t+1})) \cap (\text{sign}(w^{t+1}) \neq \text{sign}(w^{t+2}))||\tau_{-1}| < |\tau_1|) \\ &< P((\text{sign}(w^t) \neq \text{sign}(w^{t+1})) \cap (\text{sign}(w^{t+1}) \neq \text{sign}(w^{t+2}))||\tau_{-1}| = |\tau_1|). \end{aligned} \tag{2.32}$$

3. If $|\tau_1| < |\tau_{-1}|$, the process is similar to 2). The state-aware gradient also has a lower probability of sequential weight flip as:

$$\begin{aligned} &P((\text{sign}(w^t) \neq \text{sign}(w^{t+1})) \cap (\text{sign}(w^{t+1}) \neq \text{sign}(w^{t+2}))||\tau_1| < |\tau_{-1}|) \\ &< P((\text{sign}(w^t) \neq \text{sign}(w^{t+1})) \cap (\text{sign}(w^{t+1}) \neq \text{sign}(w^{t+2}))||\tau_{-1}| = |\tau_1|). \end{aligned} \tag{2.33}$$

Based on the above analysis, we propose an efficient yet simple solution to realize the state-aware gradient:

$$x^{l+1} = \begin{cases} (\text{sign}(\tau_{-1}^l x^l) * \text{sign}(w^l))\alpha & \text{if } \hat{x} = -1 \\ (\text{sign}(\tau_1^l x^l) * \text{sign}(w^l))\alpha & \text{otherwise} \end{cases}. \tag{2.34}$$

Compared to traditional BNNs, we multiply the scale τ on the activation based on its state. Note that our paper's learnable coefficients τ are per-channel granularity. In this way, our SA-BNN is established in exchange for a small increase in computational complexity (only an extra point-wise product between τ and x).

In particular, Helwegen et al. [22] argue that latent weights are not necessary for gradient-based optimization of BNNs, and they directly update the state of binarized weights with:

$$w^t = \begin{cases} -w^{t-1} & \text{if } |g^t| \geq \beta \text{ and } \text{sign}(g^t) = \text{sign}(w^{t-1}) \\ w^{t-1} & \text{otherwise} \end{cases}, \tag{2.35}$$

where $g^t = (1-\gamma)g^{t-1} + \gamma \frac{\partial \mathcal{L}}{\partial w^t}$, g^t is the exponential moving average and γ is the adaptivity rate. Then, under the constraint of $\gamma = 1$, it is easy for the weight to flip when $|\frac{\partial \mathcal{L}}{\partial w^t}| \geq \beta$ and hard to flip when $|\frac{\partial \mathcal{L}}{\partial w^t}| < \beta$, in which β is consistent with the coefficients τ in our method. However, the method in [22] suppresses the weight flip equally for different states, while SA-BNN treats different binarization states distinctively by employing an independent coefficient for each state. SA-BNN can effectively suppress the frequent weight flip problem, alleviating the ineffective update issue in BNN optimization. Moreover, unlike the handcrafted hyperparameters β, the coefficients τ are learnable, avoiding careful tuning during optimization.

Furthermore, Bai et al. [2] propose ProxQuant by formulating the quantized network training as a regularized learning problem instead and optimizing it via the prox-gradient method. Specifically, ProxQuant has access to additional gradient information at non-quantized points, which avoids the misleading weight update in training. However, unlike the ProxQuant, which suppresses the frequent weight flip by designing a dedicated optimizer, SA-BNN alleviates this problem by introducing independent learnable coefficients for different states, which can work with existing methods for back propagation and stochastic gradient descent.

In addition, due to the non-differentiability of the sign function in the binarization process, most existing works employ a surrogate for the gradients [44, 53], in which the gradients are forced to be 0 for values outside $[-1, +1]$. However, once the value falls outside the truncation interval, the corresponding weight cannot be updated anymore. This phenomenon greatly limits the training ability of backward propagation [52]. Different from these methods (i.e., $\tau_{-1} = \tau_1$), our SA-BNN has the ability to preserve more gradients through learnable coefficients, thus alleviating the unreliable gradients in BNN optimization.

2.7.3.2 Experiments

We perform experiments on the large-scale dataset ImageNet (ILSVRC12) [55], which contains approximately 1.2 million training images and $50K$ validation images from 1000 categories. In our experiments, we employ 224×224 random crop and center crop for training and inference, respectively. We use ResNet as our backbone, including ResNet-18, ResNet-34, and ResNet-50 [21]. We use Adam [33] with the momentum of 0.9 and set the weight decay to be 0. For the 18-layer SA-BNN, we run the training algorithm for 90 epochs with a batch size of 256. The learning rate starts from 0.001 and is decayed twice by multiplying 0.1 at the 75th and the 85th epoch. For the 34-layer SA-BNN, the training process includes 90 epochs, and the batch size is set to 256. The learning rate starts from 0.001 and is multiplied by 0.1 at the 60th and the 80th epoch, respectively. For the 50-layer SA-BNN, the training process is 70 epochs, and the batch size is 64. The learning rate starts from 0.0005 and is multiplied by 0.1 at the 40th and the 60th epoch, respectively.

We carry out a comparative study with six methods: IR-Net [52], Bop [22], CI-Net [63], BONN [20], Bi-Real Net [44], and XNOR-Net [53] on ResNet-18, ResNet-34, and ResNet-50 in Table 2.2. These six works are representative methods of binarizing both weights and activations for CNNs and achieving state-of-the-art results.

The comparison in Table 2.2 demonstrates that our SA-BNNs outperform other networks by a considerable margin regarding the Top-1 accuracy. Note that the results of the other six works are quoted directly from the corresponding references. Specifically, the proposed SA-BNN with backbone ResNet-18 outperforms its

Table 2.2 Comparison on Top-1 and Top-5 accuracy (%) of SA-BNN with other state-of-the-art binarization methods, including IR-Net [52], Bop [22], CI-Net [63], BONN [20], Bi-Real Net [44], and XNOR-Net [53]. "FP" means full precision

		SA-BNN	IR-Net	Bop	CI-Net	BONN	Bi-Real Net	XNOR-Net	FP
ResNet-18	Top-1	61.7	58.1	56.6	59.9	59.3	56.4	51.2	69.3
	Top-5	82.8	80.0	79.4	84.2	81.6	79.5	73.2	89.2
ResNet-34	Top-1	65.5	62.9	–	64.9	–	62.2	–	73.3
	Top-5	85.8	84.1	–	86.6	–	83.9	–	91.3
ResNet-50	Top-1	68.7	–	–	–	–	62.6	63.1	74.7
	Top-5	87.4	–	–	–	–	83.9	83.6	92.1

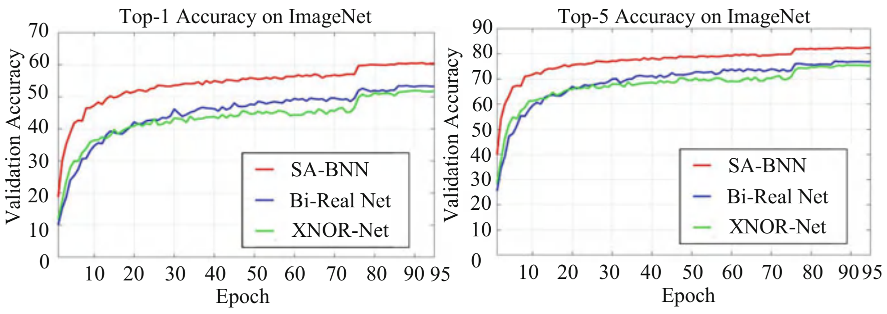


Fig. 2.8 Validation accuracy curves of SA-BNN, Bi-Real Net, and XNOR-Net with ResNet-18 backbone on ImageNet

counterpart Bi-Real Net by 5.3% and achieves a roughly 2% relative improvement over CI-Net. Similar improvements can be observed for ResNet-34 and ResNet-50 networks. In Fig. 2.8, we plot the validation accuracy curves of XNOR-Net, Bi-Real Net, and SA-BNN (without the contribution of PBN and SC). All networks are implemented under the same hyperparameter setting. It clearly shows that our method converges faster and better by learning distinctive gradient coefficients for binarization states than XNOR-Net and Bi-Real Net. Moreover, our training curve is smoother, indicating the training process is more stable. Therefore, SA-BNN is more competitive than other state-of-the-art binary networks.

We further analyze the memory usage saving and speedup in Table 2.3. We keep the weights and activations in the first convolutional and the last fully connected layers to be full-precision [44, 53]. For a fair comparison, we use FLOPs [44] and BOPs [3] to measure the total multiplication computation and bitwise operations in SA-BNNs, respectively. For ResNet-18 and ResNet-34, the proposed SA-BNNs reduce the memory usage by 11.14× and 15.81×, respectively, and achieve computation reduction by 10.74× and 18.21×, in comparison with the full-precision networks. Compared with Bi-Real Net, we obtain more than 4% accuracy improvement on ResNet-18 with small additional memory and computational cost.

Table 2.3 Memory usage, FLOPs, and BOPs calculation in our method. "MU" represents memory usage and "MS" represents memory saving

		MU	MS	FLOPs	BOPs	Speedup
ResNet-18	SA-BNN	33.6 Mbit	11.14×	1.68×10^8	1.08×10^{10}	10.74×
	Bi-Real Net	33.6 Mbit	11.14×	1.63×10^8	1.04×10^{10}	11.06×
	XNOR-Net	33.7 Mbit	11.10×	1.67×10^8	1.07×10^{10}	10.86×
	Full-precision	374.1 Mbit	–	1.81×10^9	1.16×10^{11}	–
ResNet-34	SA-BNN	44.1 Mbit	15.81×	2.01×10^8	1.29×10^{10}	18.21×
	Bi-Real Net	43.7 Mbit	15.97×	1.93×10^8	1.24×10^{10}	18.99×
	XNOR-Net	43.9 Mbit	15.88×	1.98×10^8	1.27×10^{10}	18.47×
	Full-precision	697.3 Mbit	–	3.66×10^9	2.34×10^{11}	–
ResNet-50	SA-BNN	144.4 Mbit	5.43×	3.89×10^8	2.49×10^{10}	14.65×
	Bi-Real Net	143.1 Mbit	5.48×	3.74×10^8	2.39×10^{10}	15.24×
	XNOR-Net	143.2 Mbit	5.47×	3.81×10^8	2.44×10^{10}	14.96×
	Full-precision	784.0 Mbit	–	5.70×10^9	3.65×10^{11}	–

2.7.4 PCNN: Projection Convolutional Neural Networks

Modulated convolutional networks (MCNs) are presented in [62] to binarize kernels, achieving better results than the baselines. However, in the inference step, MCNs require reconstructing full-precision convolutional filters from binarized filters, limiting their use in computationally limited environments. It has been theoretically and quantitatively demonstrated that simplifying the convolution procedure via binarized kernels and approximating the original unbinarized kernels is an up-and-coming solution toward DCNNs' compression.

Although prior BNNs significantly reduce storage requirements, they also generally have significant accuracy degradation compared to those using full-precision kernels and activations. This is mainly because CNN binarization could be solved by considering discrete optimization in the back propagation (BP) process. Discrete optimization methods can often guarantee the quality of the solutions they find and lead to much better performance in practice [16, 32, 34]. Second, the loss caused by the binarization of CNNs has yet to be well studied.

We propose a new discrete back propagation via projection (DBPP) algorithm to efficiently build our projection convolutional neural networks (PCNNs) [18] and obtain highly accurate yet robust BNNs. Theoretically, we achieve a projection loss by taking advantage of our DBPP algorithms' ability to perform discrete optimization on model compression. The advantages of the projection loss also lie in that it can be jointly learned with the conventional cross-entropy loss in the same pipeline as back propagation. The two losses are simultaneously optimized in continuous and discrete spaces, optimally combined by the projection approach in a theoretical framework. They can enrich the diversity and thus improve modeling capacity. As shown in Fig. 2.9, we develop a generic projection convolution layer that can be used in existing convolutional networks. Both the quantized kernels and

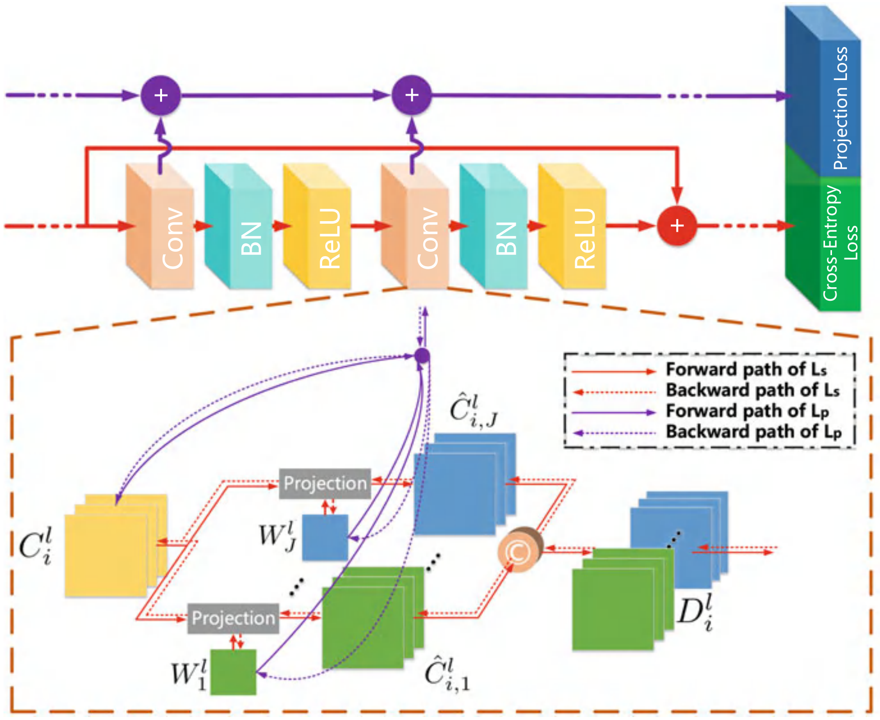


Fig. 2.9 In PCNNs, a new discrete back propagation via projection is proposed to build binarized neural networks in an end-to-end manner. Full-precision convolutional kernels C^l_i are quantized by projection as $\hat{C}^l_{i,j}$. Due to multiple projections, the diversity is enriched. The resulting kernel tensor D^l_i is used the same as in conventional ones. Both the projection loss L_p and the traditional loss L_s are used to train PCNNs. We illustrate our network structure *basic block unit* based on ResNet, and more specific details are shown in the dotted box (projection convolution layer). © indicates the concatenation operation on the channels. Note that inference does not use projection matrices W^l_j and full-precision kernels C^l_i

the projection are jointly optimized in an end-to-end manner. Our project matrices are optimized but not for reference, resulting in a compact and efficient learning architecture. As a general framework, other loss functions (e.g., center loss) can also be used to further improve the performance of our PCNNs based on a progressive optimization method.

Discrete optimization is one of the hot topics in mathematics and is widely used to solve computer vision problems [32, 34]. Conventionally, the discrete optimization problem is solved by searching for an optimal set of discrete values concerning minimizing a loss function. This paper proposes a new discrete back propagation algorithm that uses a projection function to binarize or quantize the input variables in a unified framework. Due to the flexible projection scheme, we obtain diverse binarized models with higher performance than the previous ones.

2.7.4.1 Projection

In our work, we define the quantization of the input variable as a projection onto a set:

$$\Omega := \{a_1, a_2, \ldots, a_U\}, \tag{2.36}$$

where each element $a_i, i = 1, 2, \ldots, U$ satisfies the constraint $a_1 < a_2 < \ldots < a_U$ and is the discrete value of the input variable. Then we define the projection of $x \in \mathrm{R}$ onto Ω as:

$$P_\Omega(\omega, x) = \arg\min_{a_i} \|\omega \circ x - a_i\|, i \in \{1, \ldots, U\}, \tag{2.37}$$

where ω is a projection matrix and $\circ$ denotes the Hadamard product. Equation 2.37 indicates that the projection aims to find the closest discrete value for each continuous value x. Equation 2.37 is also equal to:

$$P_\Omega(\omega, x) = \arg\min_{a_i} \|x - \hat{\omega} \circ a_i\|, i \in \{1, \ldots, U\}, \tag{2.38}$$

where $\frac{1}{\omega} = \hat{\omega}$. During the following derivation of back propagation, we still use Eq. 2.37 as the basic equation, but in its implementation, one can also use Eq. 2.38 to achieve the optimization of PCNN.

2.7.4.2 Optimization

Minimizing $f(x)$ are restricted to discrete values, which becomes more challenging when training a large-scale problem on a huge dataset [13]. We solve the problem within the back propagation framework by considering (1) the inference process of the optimized model is based on the quantized variables, which means that the variable must be quantized in the forward pass (corresponding to the inference) during training, and the loss is calculated based on the quantized variables; the variable for back propagation process is not necessarily quantized, which however needs to fully consider the relationship between quantized variables and their counterparts. Based on the above considerations, we propose that in the kth iteration, based on the projection in Eq. 2.37, $x^{[k]}$ is quantized to $\hat{x}^{[k]}$ in the forward pass as:

$$\hat{x}^{[k]} = P_\Omega(\omega, x^{[k]}), \tag{2.39}$$

which is used to improve the back propagation process by defining an objective as:

$$\begin{aligned} \min \quad & f(\omega, x) \\ \text{s.t.} \quad & \hat{x}_j^{[k]} = P_\Omega^j(\omega_j, x), \end{aligned} \tag{2.40}$$

where ω_j, $j \in \{1, \dots, J\}$ is the jth projection matrix,[1] and J is the total number of projection matrices. To solve the problem in (2.40), we define our update rule as:

$$x \leftarrow x^{[k]} - \eta\delta_{\hat{x}}^{[k]}, \tag{2.41}$$

where the superscript $[k+1]$ is removed from x, $\delta_{\hat{x}}$ is the gradient of $f(\omega, x)$ with respect to $x = \hat{x}$, and η is the learning rate. The quantization process $\hat{x}^{[k]} \leftarrow x^{[k]}$, that is, $P_\Omega^j(\omega_j, x^{[k]})$, is equivalent to finding the projection of $\omega_j \circ (x + \eta\delta_{\hat{x}}^{[k]})$ onto Ω as:

$$\hat{x}^{[k]} = \arg\min_{\hat{x}}\{\|\hat{x} - \omega_j \circ (x + \eta\delta_{\hat{x}}^{[k]})\|^2, \hat{x} \in \Omega\}. \tag{2.42}$$

Obviously, $\hat{x}^{[k]}$ is the solution to the problem in (2.42). So, by incorporating (2.42) into $f(\omega, x)$, we obtain a new formulation for (2.40) based on the Lagrangian method as:

$$\min f(\omega, x) + \frac{\lambda}{2}\sum_{j}^{J}\|\hat{x}^{[k]} - \omega_j \circ (x + \eta\delta_{\hat{x}}^{[k]})\|^2. \tag{2.43}$$

The newly added part (right) shown in (2.43) is a quadratic function and is referred to as **projection loss**.

2.7.4.3 Theoretical Analysis

We closely examine the projection loss in Eq. 2.43 and have:

$$\hat{x}^{[k]} - \omega \circ (x + \eta\delta_{\hat{x}}^{[k]}) = \hat{x}^{[k]} - \omega \circ x - \omega \circ \eta\delta_{\hat{x}}^{[k]}. \tag{2.44}$$

We only consider one projection function in this case, so the subscript j of ω_j is omitted for simplicity. For multiple projections, the analysis is given after that. In the forward step, only the discrete values participate in the calculation, so their gradients can be obtained by:

$$\frac{\partial f(\omega, \hat{x}^{[k]})}{\partial \hat{x}^{[k]}} = \omega \circ \delta_{\hat{x}}^{[k]}, \tag{2.45}$$

as ω and $\hat{x}$ are bilinear with each other as $\omega \circ \hat{x}^{[k]}$. In our discrete optimization framework, the values of convolutional kernels are updated according to their gradients. Taking Eq. 2.45 into consideration, we derive the update rule for $\hat{x}^{[k+1]}$

[1] Since the kernel parameters x are represented as a matrix, ω_j denotes a matrix as ω.

as:

$$\hat{x}^{[k+1]} = \hat{x}^{[k]} - \eta \frac{\partial f(\omega, \hat{x}^{[k]})}{\partial \hat{x}^{[k]}} = \hat{x}^{[k]} - \omega \circ \eta \delta_{\hat{x}}^{[k]}. \tag{2.46}$$

By plugging Eq. 2.46 into Eq. 2.44, we achieve a new objective function or a loss function that minimizes:

$$||\hat{x}^{[k+1]} - \omega \circ x||, \tag{2.47}$$

to approximate:

$$\hat{x} = \omega \circ x, x = \omega^{-1} \circ \hat{x}. \tag{2.48}$$

We further discuss multiple projections, based on Eq. 2.48 and projection loss in (2.43), and have:

$$\min \frac{1}{2} \sum_{j}^{J} ||x - \omega_j^{-1} \circ \hat{x}_j||^2. \tag{2.49}$$

We set $g(x) = \frac{1}{2} \sum_j^J ||x - \omega_j^{-1} \circ \hat{x}_j||^2$ and calculate its derivative as $g'(x) = 0$, and we have:

$$x = \frac{1}{J} \sum_{j}^{J} \omega_j^{-1} \circ \hat{x}_j, \tag{2.50}$$

which shows that multiple projections can better reconstruct the full kernels based on binarized counterparts.

2.7.4.4 Projection Convolutional Neural Networks

Projection convolutional neural networks (PCNNs), shown in Fig. 2.9, work using DBPP for model quantization. We accomplish this by reformulating our projection loss shown in (2.43) into the deep learning paradigm as:

$$L_p = \frac{\lambda}{2} \sum_{l,i}^{L,I} \sum_{j}^{J} ||\hat{C}_{i,j}^{l,[k]} - \widetilde{W}_j^{l,[k]} \circ (C_i^{l,[k]} + \eta \delta_{\hat{C}_{i,j}^{l,[k]}})||^2, \tag{2.51}$$

where $C_i^{l,[k]}$, $l \in \{1, \dots, L\}$, $i \in \{1, \dots, I\}$ denotes the ith kernel tensor of the lth convolutional layer in the kth iteration. $\hat{C}_{i,j}^{l,[k]}$ is the quantized kernel of $C_i^{l,[k]}$ via projection $P_\Omega^{l,j}$, $j \in \{1, \dots, J\}$ as:

$$\hat{C}_{i,j}^{l,[k]} = P_\Omega^{l,j}(\widetilde{W}_j^{l,[k]}, C_i^{l,[k]}), \tag{2.52}$$

where $\widetilde{W}_j^{l,[k]}$ is a tensor, calculated by duplicating a learned projection matrix $W_j^{l,[k]}$ along the channels, which thus fits the dimension of $C_i^{l,[k]}$. $\delta_{\hat{C}_{i,j}^{l,[k]}}$ is the gradient at $\hat{C}_{i,j}^{l,[k]}$ calculated based on L_S, that is, $\delta_{\hat{C}_{i,j}^{l,[k]}} = \frac{\partial L_S}{\partial \hat{C}_{i,j}^{l,[k]}}$. The iteration index $[k]$ is omitted for simplicity.

In PCNNs, both the cross-entropy loss and projection loss are used to build the total loss as:

$$L = L_S + L_P. \tag{2.53}$$

The proposed projection loss regularizes the continuous values converging onto Ω^N while minimizing the cross-entropy loss, illustrated in Figs. 2.11 and 2.12.

2.7.4.5 Forward Propagation Based on Projection Convolution Layer

For each full-precision kernel C_i^l, the corresponding quantized kernels $\hat{C}_{i,j}^l$ are concatenated to construct the kernel D_i^l that actually participates in the convolution operation as:

$$D_i^l = \hat{C}_{i,1}^l \oplus \hat{C}_{i,2}^l \oplus \cdots \oplus \hat{C}_{i,J}^l, \tag{2.54}$$

where $\oplus$ denotes the concatenation operation on the tensors. In PCNNs, the projection convolution is implemented based on D^l and F^l to calculate the next layer's feature map F^{l+1}:

$$F^{l+1} = Conv2D(F^l, D^l), \tag{2.55}$$

where $Conv2D$ is the traditional 2D convolution. Although our convolutional kernels are 3D-shaped tensors, we design the following strategy to fit the traditional 2D convolution as:

$$F_{h,j}^{l+1} = \sum_{i,h} F_h^l \otimes D_{i,j}^l, \tag{2.56}$$

$$F_h^{l+1} = F_{h,1}^l \oplus \cdots \oplus F_{h,J}^l, \tag{2.57}$$

where $\otimes$ denotes the convolutional operation. $F_{h,j}^{l+1}$ is the jth channel of the hth feature map at the $(l+1)$th convolutional layer and F_h^l denotes the hth feature map at the lth convolutional layer. To be more precise, for example, when $h = 1$, for the jth channel of an output feature map, $F_{1,j}^{l+1}$ is the sum of the convolutions between all the h input feature maps and i corresponding quantized kernels. All channels of the output feature maps are obtained as $F_{h,1}^{l+1}, .., F_{h,j}^{l+1}, \ldots, F_{h,J}^{l+1}$, and they are concatenated to construct the hth output feature map F_h^{l+1}.

It should be emphasized that we can utilize multiple projections to increase the diversity of convolutional kernels D^l. However, the single projection can perform much better than the existing BNNs. The essential use of DBPP differs from [38] based on a single quantization scheme. Within our convolutional scheme, there is no dimensional disagreement on feature maps and kernels in two successive layers. Thus, we can replace the traditional convolutional layers with ours to binarize widely used networks, such as VGGs and ResNets. At inference time, we only store the set of quantized kernels D_i^l instead of the full-precision ones; that is, projection matrices W_j^l are not used for inference, achieving a reduction in storage.

2.7.4.6 Backward Propagation

According to Eq. 2.53, what should be learned and updated are the full-precision kernels C_i^l and the projection matrix W^l $(\widetilde{W}^l)$ using the updated equations described below.

Updating C_i^l We define δ_{C_i} as the gradient of the full-precision kernel C_i and have:

$$\delta_{C_i^l} = \frac{\partial L}{\partial C_i^l} = \frac{\partial L_S}{\partial C_i^l} + \frac{\partial L_P}{\partial C_i^l}, \tag{2.58}$$

$$C_i^l \leftarrow C_i^l - \eta_1 \delta_{C_i^l}, \tag{2.59}$$

where η_1 is the learning rate for the convolutional kernels. More specifically, for each item in Eq. 2.58, we have:

$$\begin{aligned} \frac{\partial L_S}{\partial C_i^l} &= \sum_j^J \frac{\partial L_S}{\partial \hat{C}_{i,j}^l} \frac{\partial P_{\Omega^N}^{l,j}(\widetilde{W}_j^l, C_i^l)}{\partial (\widetilde{W}_j^l \circ C_i^l)} \frac{\partial (\widetilde{W}_j^l \circ C_i^l)}{\partial C_i^l} \\ &= \sum_j^J \frac{\partial L_S}{\partial \hat{C}_{i,j}^l} \circ \mathbb{1}_{-1 \le \widetilde{W}_j^l \circ C_i^l \le 1} \circ \widetilde{W}_j^l, \end{aligned} \tag{2.60}$$

$$\frac{\partial L_P}{\partial C_i^l} = \lambda \sum_j^J \left[\widetilde{W}_j^l \circ \left(C_i^l + \eta \delta_{\hat{C}_{i,j}^l} \right) - \hat{C}_{i,j}^l \right] \circ \widetilde{W}_j^l, \tag{2.61}$$

where $\mathbb{1}$ is the indicator function [53] widely used to estimate the gradient of the nondifferentiable function. More specifically, the output of the indicator function is 1 only if the condition is satisfied; otherwise, 0.

Updating W_j^l Likewise, the gradient of the projection parameter $\delta_{W_j^l}$ consists of the following two parts:

$$\delta_{W_j^l} = \frac{\partial L}{\partial W_j^l} = \frac{\partial L_S}{\partial W_j^l} + \frac{\partial L_P}{\partial W_j^l}, \tag{2.62}$$

$$W_j^l \leftarrow W_j^l - \eta_2 \delta_{W_j^l}, \tag{2.63}$$

where η_2 is the learning rate for W_j^l. We also have the following:

$$\begin{aligned} \frac{\partial L_S}{\partial W_j^l} &= \sum_h^J \left(\frac{\partial L_S}{\partial \widetilde{W}_j^l} \right)_h \\ &= \sum_h^J \left(\sum_i^I \frac{\partial L_S}{\partial \hat{C}_{i,j}^l} \frac{\partial P_{\Omega^N}^{l,j}(\widetilde{W}_j^l, C_i^l)}{\partial (\widetilde{W}_j^l \circ C_i^l)} \frac{\partial (\widetilde{W}_j^l \circ C_i^l)}{\partial \widetilde{W}_j^l} \right)_h \\ &= \sum_h^J \left(\sum_i^I \frac{\partial L_S}{\partial \hat{C}_{i,j}^l} \circ \mathbb{1}_{-1 \le \widetilde{W}_j^l \circ C_i^l \le 1} \circ C_i^l \right)_h , \end{aligned} \tag{2.64}$$

$$\frac{\partial L_P}{\partial W_j^l} = \lambda \sum_h^J \left(\sum_i^I \left[\widetilde{W}_j^l \circ \left(C_i^l + \eta \delta_{\hat{C}_{i,j}^l} \right) - \hat{C}_{i,j}^l \right] \circ \left(C_i^l + \eta \delta_{\hat{C}_{i,j}^l} \right) \right)_h , \tag{2.65}$$

where h indicates the hth plane of the tensor along the channels. It shows that the proposed algorithm can be trained from end to end, and we summarize the training procedure in Algorithm 1. In the implementation, we use the mean of W in the forward process but keep the original W in the backward propagation.

Note that in PCNNs for BNNs, we set $U = 2$ and $a_2 = -a_1$. Two binarization processes are used in PCNNs. The first is the kernel binarization, which is done based on the projection onto Ω^N, whose elements are calculated based on the mean absolute values of all full-precision kernels per layer [53] as:

$$\frac{1}{I} \sum_i^I \left(\| C_i^l \|_1 \right), \tag{2.66}$$

where I is the total number of kernels.

Algorithm 1: Discrete back propagation via projection

```
Input:
    The training dataset; the full-precision kernels $C$; the projection matrix $W$; the learning
    rates $\eta_1$ and $\eta_2$.
Output:
    The binary or ternary PCNNs are based on the updated $C$ and $W$.
1:  Initialize $C$ and $W$ randomly;
2:  repeat
3:      // Forward propagation
4:      for $l = 1$ to $L$ do
5:          $\hat{C}^l_{i,j} \leftarrow P(W, C^l_i)$; // using Eq. 2.52 (binary) or Eq. 2.68 (ternary)
6:          $D^l_i \leftarrow Concatenate(\hat{C}_{i,j})$; // using Eq. 2.54
7:          Perform activation binarization; //using the sign function
8:          Traditional 2D convolution; // using Eqs. 2.55, 2.56 and 2.57
9:      end for
10:     Calculate cross-entropy loss $L_S$;
11:     // Backward propagation
12:     Compute $\delta_{\hat{C}^l_{i,j}} = \frac{\partial L_S}{\partial \hat{C}^l_{i,j}}$;
13:     for $l = L$ to 1 do
14:         // Calculate the gradients
15:         calculate $\delta_{C^l_i}$; // using Eqs. 2.58, 2.60 and 2.61
16:         calculate $\delta_{W^l_j}$; // using Eqs. 2.62, 2.64 and 2.65
17:         // Update the parameters
18:         $C^l_i \leftarrow C^l_i - \eta_1 \delta_{C^l_i}$; // Eq. 2.59
19:         $W^l_j \leftarrow W^l_j - \eta_2 \delta_{W^l_j}$; //Eq. 2.63
20:     end for
21:     Adjust the learning rates $\eta_1$ and $\eta_2$.
22: until the network converges
```

2.7.4.7 Progressive Optimization

Training 1-bit CNNs is a highly non-convex optimization problem, and initialization states will significantly impact the convergence. Unlike the method in [44] that a real-valued CNN model with the clip function pre-trained on ImageNet initializes the 1-bit CNN models, we propose applying a progressive optimization strategy in training 1-bit CNNs. However, a real-valued CNN model can achieve pretty high classification accuracy, we wonder if the converging states between real-value and 1-bit CNNs, which may mistakenly guide the converging process of 1-bit CNNs.

We believe that compressed ternary CNNs such as TTN [79] and TWN [36] have better initialization states for binary CNNs. Theoretically, the performance of models with ternary weights is slightly better than those with binary weights and far worse than those of real-valued ones. Still, they provide an excellent initialization state for 1-bit CNNs in our proposed progressive optimization framework. Subsequent experiments show that our PCNNs trained from a progressive optimization strategy perform better than those from scratch, even better than the ternary PCNNs from scratch.

The discrete set for ternary weights is a special case, defined as $\Omega := \{a_1, a_2, a_3\}$. We further require $a_1 = -a_3 = \Delta$ as Eq. 2.66 and $a_2 = 0$ to be hardware friendly [36]. Regarding the threshold for ternary weights, we follow the choice made in [58] as:

$$\Delta^l = \sigma \times E(|C^l|) \approx \frac{\sigma}{I} \sum_{i}^{I} \left(\|C_i^l\|_1 \right), \tag{2.67}$$

where σ is a constant factor for all layers. Note that [58] applies to Eq. 2.67 on convolutional inputs or feature maps; we find it appropriate in convolutional weights as well. Consequently, we redefine the projection in Eq. 2.37 as:

$$P_\Omega(\omega, x) = \arg\min_{a_i} \|\omega \circ x - 2a_i\|, i \in \{1, \ldots, U\}. \tag{2.68}$$

In our proposed progressive optimization framework, the PCNNs with ternary weights (ternary PCNNs) are first trained from scratch and then served as pre-trained models to progressively fine-tune the PCNNs with binary weights (binary PCNNs).

To alleviate the disturbance caused by the quantization process, intraclass compactness is further deployed based on the center loss function [65] to improve performance. Given the input features $x_i \in \mathbb{R}^d$ or Ω and the y_ith class center $c_{y_i} \in \mathbb{R}^d$ or Ω of the input features, we have:

$$L_C = \frac{\gamma}{2} \sum_{i=1}^{m} \|x_i - c_{y_i}\|_2^2, \tag{2.69}$$

where m denotes the total number of samples or batch size and γ is a hyperparameter to balance the center loss with other losses. More details on center loss can be found in [65]. By incorporating Eq. 2.69 into Eq. 2.53, the total loss is updated as:

$$L = L_S + L_P + L_C. \tag{2.70}$$

We note that the center loss is successfully deployed to handle feature variations in the training and will be omitted in the inference, so there is no additional memory storage and computational cost. More intuitive illustrations can be found in Fig. 2.10, and a more detailed training procedure is described in Algorithm 2.

2.7.4.8 Ablation Study

Parameter As mentioned above, the proposed projection loss, similar to clustering, can control quantization. We computed the distributions of the full-precision kernels and visualized the results in Figs. 2.11 and 2.12. The hyperparameter λ is designed to balance projection loss and cross-entropy loss. We vary it from $1e-3$

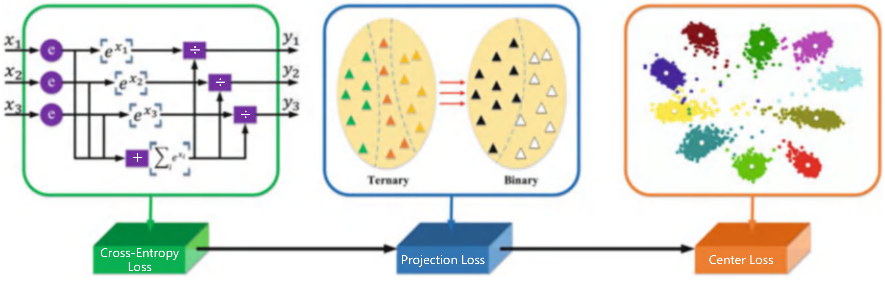


Fig. 2.10 In our proposed progressive optimization framework, the two additional losses, projection loss, and center loss are simultaneously optimized in continuous and discrete spaces, optimally combined by the projection approach in a theoretical framework. The subfigure on the left explains the softmax function in the cross-entropy loss. The subfigure in the middle illustrates the process of progressively turning ternary kernel weights into binary ones within our projection approach. The subfigure on the right shows the function of center loss to force the learned feature maps to cluster together, class by class. Best viewed in color

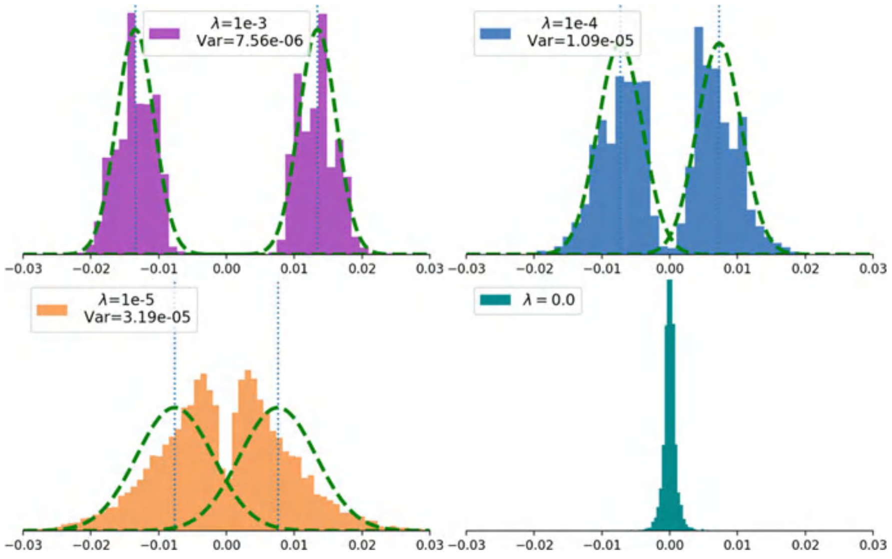


Fig. 2.11 We visualize the distribution of kernel weights of the first convolution layer of PCNN-22. The variance increases when the ratio decreases λ, which balances projection loss and cross-entropy loss. In particular, when $\lambda = 0$ (no projection loss), only one group is obtained, where the kernel weights are distributed around 0, which could result in instability during binarization. In contrast, two Gaussians (with projection loss, $\lambda > 0$) are more powerful than the single one (without projection loss), which thus results in better BNNs, as also validated in Table 2.4

to $1e - 5$ and finally set it to 0 in Fig. 2.11, where the variance increases as the number of λ. When $\lambda = 0$, only one cluster is obtained, where the kernel weights are tightly distributed around the threshold = 0. This could result in instability during

Algorithm 2: Progressive optimization with center loss

```
Input: The training dataset; the full-precision kernels $C$; the pre-trained kernels ${}^tC$ from
    ternary PCNNs; the projection matrix $W$; the learning rates $\eta_1$ and $\eta_2$.
Output: The binary PCNNs are based on the updated $C$ and $W$.
 1: Initialize $W$ randomly but $C$ from ${}^tC$;
 2: repeat
 3:    // Forward propagation
 4:    for $l = 1$ to $L$ do
 5:       $\hat{C}^l_{i,j} \leftarrow P(W, C^l_i)$; // using Eq. 2.52
 6:       $D^l_i \leftarrow Concatenate(\hat{C}_{i,j})$; // using Eq. 2.54
 7:       Perform activation binarization; //using the sign function
 8:       Traditional 2D convolution; // using Eqs. 2.55, 2.56 and 2.57
 9:    end for
10:    Calculate cross-entropy loss $L_S$;
11:    if using center loss then
12:       $L' = L_S + L_C$;
13:    else
14:       $L' = L_S$;
15:    end if
16:    // Backward propagation
17:    Compute $\delta_{\hat{C}^l_{i,j}} = \frac{\partial L'}{\partial \hat{C}^l_{i,j}}$;
18:    for $l = L$ to 1 do
19:       // Calculate the gradients
20:       calculate $\delta_{C^l_i}$; // using Eqs. 2.58, 2.60 and 2.61
21:       calculate $\delta_{W^l_j}$; // using Eqs. 2.62, 2.64 and 2.65
22:       // Update the parameters
23:       $C^l_i \leftarrow C^l_i - \eta_1 \delta_{C^l_i}$; // Eq. 2.59
24:       $W^l_j \leftarrow W^l_j - \eta_2 \delta_{W^l_j}$; // Eq. 2.63
25:    end for
26:    Adjust the learning rates $\eta_1$ and $\eta_2$.
27: until the network converges
```

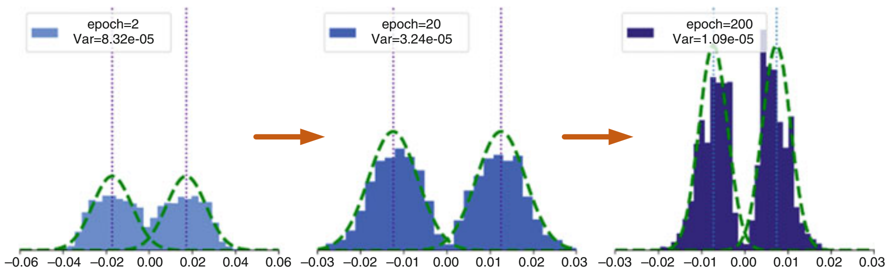


Fig. 2.12 With λ fixed to $1e-4$, the variance of the kernel weights decreases from the 2nd epoch to the 200th epoch, which confirms that the projection loss does not affect the convergence

Table 2.4 With different λ, the accuracy of PCNN-22 and PCNN-40 based on WRN-22 and WRN-40, respectively, on CIFAR10 dataset

	λ			
Model	$1e-3$	$1e-4$	$1e-5$	0
PCNN-22	91.92	92.79	92.24	91.52
PCNN-40	92.85	93.78	93.65	92.84

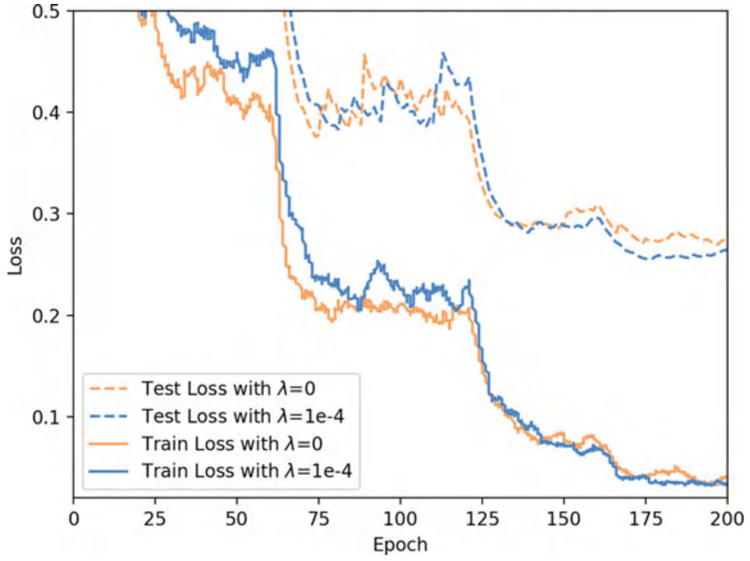


Fig. 2.13 Training and testing curves of PCNN-22 when $\lambda = 0$ and $1e-4$, which shows that the projection affects little on the convergence

binarization because little noise may cause a positive weight to be negative and vice versa.

We also show the evolution of the distribution of how projection loss works in the training process in Fig. 2.12. A natural question is: do we always need a large λ? As a discrete optimization problem, the answer is no. The experiment in Table 2.4 can verify it, i.e., both the projection and cross-entropy losses should be considered simultaneously with good balance. For example, when λ is set to $1e-4$, the accuracy is higher than those with other values. Thus, we fix λ to $1e-4$ in the following experiments.

Learning Convergence For PCNN-22 in Table 2.4, the PCNN model is trained for 200 epochs and then used to perform inference. In Fig. 2.13, we plot training and test loss with $\lambda = 0$ and $\lambda = 1e-4$, respectively. It clearly shows that PCNNs with $\lambda = 1e-4$ (blue curves) converge faster than PCNNs with $\lambda = 0$ (yellow curves) when the epoch number > 150.

Diversity Visualization In Fig. 2.14, we visualize four channels of the binary kernels D_i^l in the first row, the feature maps produced by D_i^l in the second row,

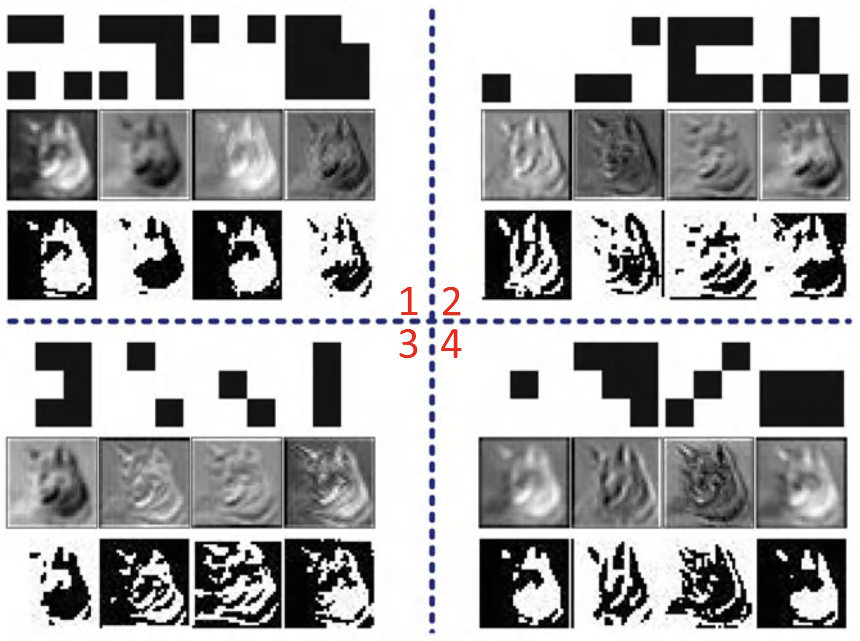


Fig. 2.14 Illustration of binary kernels D_i^l (first row), feature maps produced by D_i^l (second row), and corresponding feature maps after binarization (third row) when $J = 4$. This confirms the diversity in PCNNs

and the corresponding feature maps after binarization in the third row when $J = 4$. This way helps illustrate the diversity of kernels and feature maps in PCNNs. Thus, multiple projection functions can capture diverse information and perform highly based on compressed models.

References

1. Milad Alizadeh, Javier Fernández-Marqués, Nicholas D Lane, and Yarin Gal. An empirical study of binary neural networks' optimisation. In *Proceedings of the International Conference on Learning Representations*, 2018.
2. Yu Bai, Yu-Xiang Wang, and Edo Liberty. Proxquant: Quantized neural networks via proximal operators. *arXiv preprint arXiv:1810.00861*, 2018.
3. Chaim Baskin, Eli Schwartz, Evgenii Zheltonozhskii, Natan Liss, Raja Giryes, Alex M Bronstein, and Avi Mendelson. Uniq: Uniform noise injection for non-uniform quantization of neural networks. *arXiv preprint arXiv:1804.10969*, 2018.
4. Joseph Bethge, Christian Bartz, Haojin Yang, Ying Chen, and Christoph Meinel. Meliusnet: Can binary neural networks achieve mobilenet-level accuracy? *arXiv preprint arXiv:2001.05936*, 2020.

5. Joseph Bethge, Marvin Bornstein, Adrian Loy, Haojin Yang, and Christoph Meinel. Training competitive binary neural networks from scratch. *arXiv preprint arXiv:1812.01965*, 2018.
6. Joseph Bethge, Haojin Yang, Marvin Bornstein, and Christoph Meinel. Binarydensenet: developing an architecture for binary neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.
7. Leo Breiman. Bias, variance, and arcing classifiers. Technical report, Tech. Rep. 460, Statistics Department, University of California, Berkeley . . . , 1996.
8. Adrian Bulat, Jean Kossaifi, Georgios Tzimiropoulos, and Maja Pantic. Matrix and tensor decompositions for training binary neural networks. *arXiv preprint arXiv:1904.07852*, 2019.
9. Adrian Bulat and Georgios Tzimiropoulos. Binarized convolutional landmark localizers for human pose estimation and face alignment with limited resources. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3706–3714, 2017.
10. Adrian Bulat and Georgios Tzimiropoulos. XNOR-Net++: Improved binary neural networks. *arXiv preprint arXiv:1909.13863*, 2019.
11. John G Carney, Pádraig Cunningham, and Umesh Bhagwan. Confidence and prediction intervals for neural network ensembles. In *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339)*, volume 2, pages 1215–1218. IEEE, 1999.
12. Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.
13. Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.
14. Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11408–11417, 2019.
15. Ruizhou Ding, Zeye Liu, Rongye Shi, Diana Marculescu, and RD Blanton. LightNN: Filling the gap between conventional deep neural networks and binarized networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 35–40, 2017.
16. Pedro Felzenszwalb and Ramin Zabih. Discrete optimization algorithms in computer vision. *Tutorial at IEEE International Conference on Computer Vision*, 2007.
17. Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4852–4861, 2019.
18. Jiaxin Gu, Ce Li, Baochang Zhang, J. Han, Xianbin Cao, Jianzhuang Liu, and David S. Doermann. Projection convolutional neural networks for 1-bit CNNs via discrete back propagation. *ArXiv*, abs/1811.12755, 2018.
19. Jiaxin Gu, Ce Li, Baochang Zhang, Jungong Han, Xianbin Cao, Jianzhuang Liu, and David Doermann. Projection convolutional neural networks for 1-bit cnns via discrete back propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.
20. Jiaxin Gu, Junhe Zhao, Xiaolong Jiang, Baochang Zhang, Jianzhuang Liu, Guodong Guo, and Rongrong Ji. Bayesian optimized 1-bit cnns. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4909–4917, 2019.
21. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
22. Koen Helwegen, James Widdicombe, Lukas Geiger, Zechun Liu, Kwang-Ting Cheng, and Roeland Nusselder. Latent weights do not exist: Rethinking binarized neural network optimization. In *Advances in neural information processing systems*, pages 7531–7542, 2019.
23. Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

24. Kun Huang, Bingbing Ni, and Xiaokang Yang. Efficient quantization for neural networks with binary weights and low bitwidth activations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3854–3861, 2019.
25. Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.
26. Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.
27. Felix Juefei-Xu, Vishnu Naresh Boddeti, and Marios Savvides. Local binary convolutional neural networks. *CoRR*, abs/1608.06049, 2016.
28. Felix Juefei-Xu, Vishnu Naresh Boddeti, and Marios Savvides. Local binary convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 19–28, 2017.
29. Mohammad Emtiyaz Khan and Haavard Rue. Learningalgorithms from bayesian principles. *arXiv preprint arXiv:2002.10778*, 2(4), 2020.
30. Hyungjun Kim, Kyungsu Kim, Jinseok Kim, and Jae-Joon Kim. Binaryduo: Reducing gradient mismatch in binary activation network by coupling binary activations. In *International Conference on Learning Representations*.
31. Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.
32. Seungryong Kim, Dongbo Min, Stephen Lin, and Kwanghoon Sohn. DCTM: Discrete-continuous transformation matching for semantic flow. In *Proceedings of the IEEE International Conference on Computer Vision*, volume 6, 2017.
33. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
34. Emanuel Laude, Jan-Hendrik Lange, Jonas Sch pfer, Csaba Domokos, Leal-Taix? Laura, Frank R. Schmidt, Bjoern Andres, and Daniel Cremers. Discrete-continuous ADMM for transductive inference in higher-order MRFs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4539–4548, 2018.
35. Cong Leng, Zesheng Dou, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3466–3473, 2018.
36. Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.
37. Zefan Li, Bingbing Ni, Wenjun Zhang, Xiaokang Yang, and Wen Gao. Performance guaranteed network acceleration via high-order residual quantization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2584–2592, 2017.
38. Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.
39. Chunlei Liu, Peng Chen, Bohan Zhuang, Chunhua Shen, Baochang Zhang, and Wenrui Ding. SA-BNN: State-aware binary neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 2091–2099, 2021.
40. Chunlei Liu, Wenrui Ding, Yuan Hu, Baochang Zhang, Jianzhuang Liu, Guodong Guo, and David Doermann. Rectified binary convolutional networks with generative adversarial learning. *International Journal of Computer Vision*, 129:998–1012, 2021.
41. Chunlei Liu, Wenrui Ding, Xin Xia, Baochang Zhang, Jiaxin Gu, Jianzhuang Liu, Rongrong Ji, and David Doermann. Circulant binary convolutional networks: Enhancing the performance of 1-bit DCNNs with circulant back propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2691–2699, 2019.
42. Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. In *Proc. of ECCV*, 2016.
43. Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. *arXiv preprint arXiv:2003.03488*, 2020.

44. Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018.
45. Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. In *ICLR*. 2020.
46. Xiangming Meng, Roman Bachmann, and Mohammad Emtiyaz Khan. Training binary neural networks using the bayesian learning rule. In *International conference on machine learning*, pages 6852–6861. PMLR, 2020.
47. Luca Mocerino and Andrea Calimera. Tentaclenet: A pseudo-ensemble template for accurate binary convolutional neural networks. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 261–265. IEEE, 2020.
48. Jean-Jacques Moreau. Proximité et dualité dans un espace hilbertien. *Bulletin de la Société mathématique de France*, 93:273–299, 1965.
49. Nikunj C Oza and Stuart J Russell. Online bagging and boosting. In *International Workshop on Artificial Intelligence and Statistics*, pages 229–236. PMLR, 2001.
50. Haotong Qin, Zhongang Cai, Mingyuan Zhang, Yifu Ding, Haiyu Zhao, Shuai Yi, Xianglong Liu, and Hao Su. Bipointnet: Binary neural network for point clouds. In *Proceedings of the International Conference on Learning Representations*, 2021.
51. Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2250–2259, 2020.
52. Haotong Qin, Ruihao Gong, Xianglong Liu, Ziran Wei, Fengwei Yu, and Jingkuan Song. Ir-net: Forward and backward information retention for highly accurate binary neural networks. *arXiv preprint arXiv:1909.10788*, 2019.
53. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.
54. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NeurIPS*, 2015.
55. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.
56. VW-S Tseng, Sourav Bhattachara, Javier Fernández-Marqués, Milad Alizadeh, Catherine Tong, and Nicholas D Lane. Deterministic binary filters for convolutional neural networks. International Joint Conferences on Artificial Intelligence Organization, 2018.
57. Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. Tbn: Convolutional neural network with ternary inputs and binary weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 315–332, 2018.
58. Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. Tbn: Convolutional neural network with ternary inputs and binary weights. In *Proceedings of the European Conference on Computer Vision*, pages 315–332, 2018.
59. Peisong Wang, Qinghao Hu, Yifan Zhang, Chunjie Zhang, Yang Liu, and Jian Cheng. Two-step quantization for low-bit neural networks. In *Proceedings of the IEEE Conference on computer vision and pattern recognition*, pages 4376–4384, 2018.
60. Song Wang, Dongchun Ren, Li Chen, Wei Fan, Jun Sun, and Satoshi Naoi. On study of the binarized deep neural network for image classification. *arXiv preprint arXiv:1602.07373*, 2016.
61. Xiaodi Wang, Baochang Zhang, Ce Li, Rongrong Ji, Jungong Han, Xianbin Cao, and Jianzhuang Liu. Modulated convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 840–848, 2018.
62. Xiaodi Wang, Baochang Zhang, Ce Li, Rongrong Ji, Jungong Han, Xianbin Cao, and Jianzhuang Liu. Modulated convolutional networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 840–848, 2018.

63. Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. Learning channel-wise interactions for binary convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 568–577, 2019.
64. Ziwei Wang, Ziyi Wu, Jiwen Lu, and Jie Zhou. Bidet: An efficient binarized object detector. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2049–2058, 2020.
65. Yandong Wen, Kaipeng Zhang, Zhifeng Li, and Yu Qiao. A discriminative feature learning approach for deep face recognition. In *European Conference on Computer Vision (ECCV)*, pages 499–515, 2016.
66. Sheng Xu, Yanjing Li, Tiancheng Wang, Teli Ma, Baochang Zhang, Peng Gao, Yu Qiao, Jinhu Lü, and Guodong Guo. Recurrent bilinear optimization for binary neural networks. In *European Conference on Computer Vision*, pages 19–35. Springer, 2022.
67. Sheng Xu, Yanjing Li, Bohan Zeng, Teli Ma, Baochang Zhang, Xianbin Cao, Peng Gao, and Jinhu Lü. Ida-det: An information discrepancy-aware distillation for 1-bit detectors. In *European Conference on Computer Vision*, pages 346–361. Springer, 2022.
68. Sheng Xu, Yanjing Li, Junhe Zhao, Baochang Zhang, and Guodong Guo. Poem: 1-bit point-wise operations based on expectation-maximization for efficient point cloud processing. In *Proceedings of the British Machine Vision Conference*, 2021.
69. Sheng Xu, Chang Liu, Baochang Zhang, Jinhu Lü, Guodong Guo, and David Doermann. Bire-id: Binary neural network for efficient person re-id. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 18(1s):1–22, 2022.
70. Sheng Xu, Zhendong Liu, Xuan Gong, Chunlei Liu, Mingyuan Mao, and Baochang Zhang. Amplitude suppression and direction activation in networks for 1-bit faster r-cnn. In *Proc. of EMDL*, 2020.
71. Sheng Xu, Junhe Zhao, Jinhu Lu, Baochang Zhang, Shumin Han, and David Doermann. Layer-wise searching for 1-bit detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5682–5691, 2021.
72. Zhe Xu and Ray CC Cheung. Accurate and compact convolutional neural networks with trained binarization. In *30th British Machine Vision Conference*, 2019.
73. Zihan Xu, Mingbao Lin, Jianzhuang Liu, Jie Chen, Ling Shao, Yue Gao, Yonghong Tian, and Rongrong Ji. ReCU: Reviving the dead weights in binary neural networks. *arXiv preprint arXiv:2103.12369*, 2021.
74. Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. BMXNet: An open-source binary neural network implementation based on MXNet. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1209–1212, 2017.
75. Penghang Yin, Shuai Zhang, Jiancheng Lyu, Stanley Osher, Yingyong Qi, and Jack Xin. Binaryrelax: A relaxation approach for training deep neural networks with quantized weights. *SIAM Journal on Imaging Sciences*, 11(4):2205–2223, 2018.
76. Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. LQ-Nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.
77. Junhe Zhao, Sheng Xu, Baochang Zhang, Jiaxin Gu, David Doermann, and Guodong Guo. Towards compact 1-bit cnns via bayesian learning. *International Journal of Computer Vision*, pages 1–25, 2022.
78. Feng Zheng, Cheng Deng, and Heng Huang. Binarized neural networks for resource-efficient hashing with minimizing quantization loss. In *IJCAI*, pages 1032–1040, 2019.
79. Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.
80. Shilin Zhu, Xin Dong, and Hao Su. Binary ensemble neural network: More bits per network or more networks per bit? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4923–4932, 2019.
81. Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. Structured binary neural networks for accurate image classification and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 413–422, 2019.

Chapter 3
Binary Neural Architecture Search

3.1 Introduction

Deep convolutional neural networks (DCNNs) have achieved state-of-the-art performance in various computer vision tasks, including image classification, instance segmentation, and object detection. The success of DCNNs is attributed to effective architecture design. Neural architecture search (NAS) is an emerging approach that automates the process of designing neural architectures, replacing manual design.

NAS has enabled significant improvements in performance across a wide range of computer vision tasks. Traditionally, network architectures were manually designed, but NAS automates this process by generating sophisticated neural architectures. Existing NAS methods can be classified into three main categories: evolution-based, reinforcement learning-based, and one-shot-based approaches. These methods leverage different optimization strategies to search for the best neural architecture for a specific task. NAS has shown promising results in achieving competitive and even superior performance compared to manually designed networks.

To speed up the architecture search process, researchers have explored methods to reduce the evaluation cost of each candidate architecture. One early approach was to share weights between searched and newly generated networks [7]. Later, this idea evolved into a more efficient framework called one-shot architecture search.

In one-shot architecture search, an over-parameterized network or super-network that includes all candidate operations is trained only once. The final architecture is obtained by sampling from this super-network. Different approaches have been proposed to implement one-shot architecture search. For example, some methods use HyperNet to train the over-parameterized network [4], while others share parameters among child models to avoid retraining each candidate architecture from scratch [62].

Differentiable architecture search (DARTS) is a popular one-shot architecture search method that introduces a differentiable framework, combining the search and

B. Zhang et al., *Neural Networks with Model Compression*,
Computational Intelligence Methods and Applications,
https://doi.org/10.1007/978-981-99-5068-3_3

evaluation stages into one [56]. Despite its simplicity, DARTS has some limitations, leading researchers to propose improved approaches like PDARTS, which allows the depth of searched architectures to grow gradually during the training procedure, reducing search time [15].

ProxylessNAS is another notable method that adopts a differentiable framework and searches architectures on the target task instead of using a proxy-based framework [9]. These approaches have significantly accelerated the architecture search process and led to state-of-the-art neural network architectures.

Binary neural architecture search replaces the real-valued weights and activations with binarized ones, which consumes much less memory and computational resources to search binary networks and provides a more promising way to find network architectures efficiently. These methods can be categorized into *direct binary architecture search* and *auxiliary binary architecture search*. Direct binary architecture search yields binary architectures directly from well-designed binary search spaces. As the first art in this field, BNAS_1 [11] effectively reduces search time by channel sampling and search space pruning in the early training stages for a differentiable NAS. BNAS_2 [43] utilizes diversity in the early search to learn better performing binary architectures. BMES [63] learns an efficient binary MobileNet [40] architecture through evolution-based search. However, the accuracy of the direct binary architecture search can be improved by the auxiliary binary architecture search [6]. BATS [6] designs a new search space specially tailored for the binary network and incorporates it into the DARTS framework.

Unlike the methods above, our work is driven by the performance discrepancy between the 1-bit neural architecture and its real-valued counterpart. We introduce tangent propagation to explore the accuracy discrepancy and further accelerate the search process by applying the GGN to the Hessian matrix in optimization. Furthermore, we introduce a novel decoupled optimization to address asynchronous convergence in such a differentiable NAS process, leading to better-performed 1-bit CNNs. The overall framework leads to a novel and effective BNAS process.

To introduce the advances of the NAS area, we separately introduce the representative works in the NAS and binary NAS in the following.

3.2 Neural Architecture Search

3.2.1 ABanditNAS: Anti-bandit for Neural Architecture Search

Low search efficiency has prevented NAS from its practical use, and the introduction of adversarial optimization and a more extensive search space further exacerbates the issue. Early work directly regards network architecture search as a black-box optimization problem in a discrete search space and takes thousands of GPU days. To reduce the search space, a common idea is to adopt a cell-based search space [96].

However, when searching in a vast and complicated search space, prior cell-based works may still suffer from memory issues and are computationally intensive with the number of meta-architecture. For example, DARTS [56] can only optimize over a small subset of eight cells stacked to form a deep network of 20. To increase search efficiency, we reformulate NAS as a multi-armed bandit problem with a vast search space. The multi-armed bandit algorithm targets predicting the best arm in a sequence of trials to balance the result and its uncertainty. Likewise, NAS aims to get the best operation from an operation pool at each edge of the model with finite optimization steps, similar to the multi-armed bandit algorithm. They are both exploration and exploitation problems. Therefore, we tried to introduce the multi-armed bandit algorithm into NAS. In addition, the multi-armed bandit algorithm avoids the gradient descent process and provides good search speed for NAS. Unlike traditional upper confidence bound (UCB) bandit algorithms that prefer to sample using UCB and focus on exploration, we propose anti-bandit to exploit further both UCB and lower confidence bound (LCB) to balance exploration and exploitation. We achieve an accuracy-bias trade-off during the search process for the operation performance estimation. Using the test performance to identify the optimal architecture quickly is desirable. With the help of the anti-bandit algorithm, our anti-bandit NAS (ABanditNAS) [10] can handle the vast and complicated search space, where the number of operations that define the space can be 9^{60}!

Specifically, our proposed anti-bandit algorithm uses UCB to reduce search space, and LCB guarantees that every arm is thoroughly tested before abandoning it, as shown in Fig. 3.1. Based on the observation that the early optimal operation is not necessarily the optimal one in the end, and the worst operations in the early stage usually have worse performance in the end [89], we pruned the operations with the worst UCB, after enough trials selected by the worst LCB. This means that the operations we finally reserve are certainly a near-optimal solution. The more tests conducted, the closer UCB and LCB are to the average value. Therefore, LCB increases, and UCB decreases with increasing sampling times. Specifically, operations with poor performance in the early stages, such as parameterized operations, will receive more opportunities but are abandoned once they are confirmed to be wrong. Meanwhile, weight-free operations will be

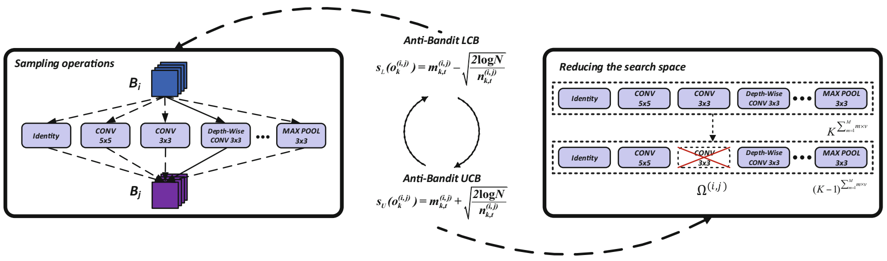

Anti-Bandit LCB
$s_L(o_k^{(i,j)}) = m_{k,t}^{(i,j)} - \sqrt{\frac{2\log N}{n_{k,t}^{(i,j)}}}$
Sampling operations
B_i
Identity
CONV 5x5
CONV 3x3
Depth-Wise CONV 3x3
MAX POOL 3x3
B_j
Anti-Bandit UCB
$s_U(o_k^{(i,j)}) = m_{k,t}^{(i,j)} + \sqrt{\frac{2\log N}{n_{k,t}^{(i,j)}}}$
Reducing the search space
$K^{\sum_{m=1}^{M} m\times v}$
$\Omega^{(i,j)}$
$(K-1)^{\sum_{m=1}^{M} m\times v}$

Fig. 3.1 ABanditNAS is divided into two steps: sampling using LCB and abandoning using UCB

compared only with parameterized operations when well-trained. On the other hand, with the operation pruning process, the search space becomes smaller and smaller, leading to an efficient search process.

3.2.1.1 Anti-Bandit Algorithm

Our goal is to search for network architectures effectively and efficiently. However, a dilemma exists for NAS about maintaining a network structure that offers significant rewards (exploitation) or further investigating other network structures (exploration). Based on probability theory, the multi-armed bandit can solve the aforementioned exploration-versus-exploitation dilemma, which makes decisions among competing choices to maximize their expected gain. Specifically, we propose an anti-bandit that chooses and discards the arm k in the trial based on:

$$\tilde{r}_k - \tilde{\delta}_k \leq r_k \leq \tilde{r}_k + \tilde{\delta}_k, \tag{3.1}$$

where r_k, $\tilde{r}_k$, and $\tilde{\delta}_k$ are the true reward, the average reward, and the estimated variance obtained from arm k. $\tilde{r}_k$ is the value term that favors actions that historically perform well, and $\tilde{\delta}_k$ is the exploration term that gives actions an exploration bonus. $\tilde{r}_k - \tilde{\delta}_k$ and $\tilde{r}_k + \tilde{\delta}_k$ can be interpreted as the lower and upper bounds of a confidence interval.

The traditional UCB algorithm, which optimistically substitutes $\tilde{r}_k + \tilde{\delta}$ for r_k, emphasizes exploration but ignores exploitation. Unlike the UCB bandit, we further exploited the LCB and UCB to balance exploration and exploitation. A smaller LCB usually has little expectations but a significant variance and should be given a larger chance to be sampled for more trials. Then, based on the observation that the worst operations in the early stage usually have worse performance at the end [89], we use UCB to prune the operation with the worst performance and reduce the search space. In summary, we adopt LCB, $\tilde{r}_k - \tilde{\delta}$, to sample the arm, which should be further optimized, and use UCB, $\tilde{r}_k + \tilde{\delta}$, to abandon the operation with the minimum value. Because the variance is bounded and converges, the operating estimate value is always close to the actual value and gradually approaches the true value as the number of trials increases. Our anti-bandit algorithm overcomes the limitations of an exploration-based strategy, including levels of understanding and suboptimal gaps. The definitions of the value and variance terms and the proof of our proposed method are shown below.

Definition 1 If an operation on arm k has been recommended n_k times, $reward_i$ is the reward on arm k on all trails. The value term of anti-bandit is defined as follows:

$$\tilde{r}_k = \frac{\sum reward_i}{n_k}. \tag{3.2}$$

The value of selecting an operation $\tilde{r}_k$ is the expected reward $\sum reward_i$ we receive when we take an operation from the possible set of operations. If n_k approaches infinity, $\tilde{r_k}$ approaches the actual value of the operation r_k. However, the number of operations n_k cannot be infinite. Therefore, we should approximate the actual value as closely as possible through the variance.

Definition 2 There exists a difference between the estimated probability $\tilde{r_k}$ and the actual probability r_k, and we can estimate the variance concerning the value:

$$\tilde{\delta}_k = \sqrt{\frac{2\ln N}{n}}, \tag{3.3}$$

where N is the total number of trails.

Proof Suppose $X \in [0, 1]$ represents the theoretical value of each independently distributed operation. n is the number of times the arm has been played up to trial, and p_i is the actual value of the operation in the ith trial. Furthermore, we define $p = \frac{\sum_i p_i}{n}$ and $q = 1 - p$. Since the variance boundary of independent operations can represent the global variance boundary (see the Appendix), based on Markov's inequality, we can arrive at below:

$$\begin{aligned} P[X > p + \delta] &= P[\sum_i (X_i - p_i) > \delta] \\ &= P[e^{\lambda \sum_i (X_i - p_i)} > e^{\lambda\delta}] \\ &\leq \frac{E[e^{\lambda \sum_i (X_i - p_i)}]}{e^{\lambda\delta}}. \end{aligned} \tag{3.4}$$

Since we can get $1 + x \leq e^x \leq 1 + x + x^2$ when $0 \leq |x| \leq 1$), $E[e^{\lambda \sum_i (X_i - p_i)}]$ in Eq. 3.4 can be further approximated as follows:

$$\begin{aligned} E[e^{\lambda \sum_i (X_i - p_i)}] &= \prod_i E[e^{\lambda (X_i - p_i)}] \\ &\leq \prod_i E[1 + \lambda(X_i - p_i) + \lambda^2 (X_i - p_i)^2] \\ &= \prod_i (1 + \lambda^2 v_i^2) \\ &\leq e^{\lambda^2 v^2}, \end{aligned} \tag{3.5}$$

where v denotes the variance of X. Combining Eqs. 3.4 and 3.5 gives $P[X > p + \delta] \leq \frac{e^{\lambda^2 v^2}}{e^{\lambda\delta}}$. Since λ is a positive constant, it can be obtained by the transformation of the values $P[X > p + \delta] \leq e^{-2n\delta^2}$. According to the symmetry of the distribution, we have $P[X < p - \delta] \leq e^{-2n\delta^2}$. Finally, we get the following inequality:

$$P[|X - p| \leq \delta] \geq 1 - 2e^{-2n\delta^2}. \tag{3.6}$$

We need to decrease δ as operating recommendations increase. Therefore, we choose $\sqrt{\frac{2\ln N}{n}}$ as $\tilde{\delta}$. That is to say, $p - \sqrt{\frac{2\ln N}{n}} \leq X \leq p + \sqrt{\frac{2\ln N}{n}}$ is implemented at least with probability $1 - \frac{2}{N^4}$. The variance value will gradually decrease as the trial progresses, and $\tilde{r_k}$ will gradually approach r_k. Equation 3.7 shows that we can achieve a probability of 0.992 when the number of the trial gets 4:

$$\sqrt{1 - \frac{2}{N^4}} = \begin{cases} 0.857 & N = 2 \\ 0.975 & N = 3 \\ 0.992 & N = 4. \end{cases} \tag{3.7}$$

According to Eq. 3.6, the variance in the anti-bandit algorithm is bounded, and the lower/upper confidence bounds can be estimated as:

$$\tilde{r_k} - \sqrt{\frac{2\ln N}{n}} \leq r_k \leq \tilde{r_k} + \sqrt{\frac{2\ln N}{n}}. \tag{3.8}$$

3.2.1.2 Search Space

Following [56, 89, 96], we search for computation cells as the building blocks of the final architecture. A cell is a fully connected directed acyclic graph (DAG) of M nodes, i.e., $\{B_1, B_2, \ldots, B_M\}$ as shown in Fig. 3.2a. Here, each node is a specific tensor (e.g., a feature map in convolutional neural networks), and each directed edge (i, j) between B_i and B_j denotes an operation $o^{(i,j)}(.)$, which is sampled from

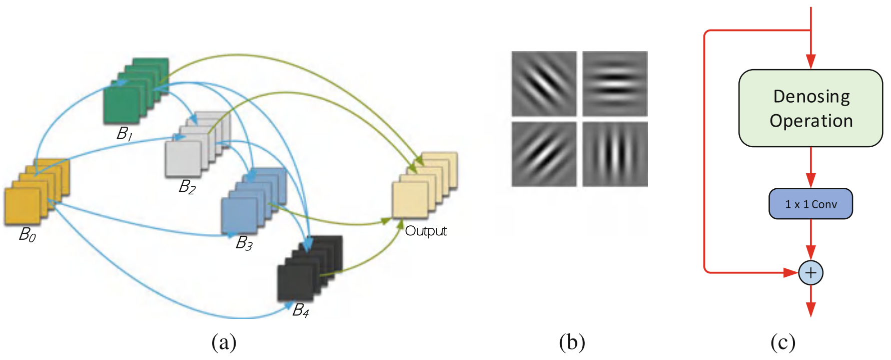


Fig. 3.2 (**a**) A cell containing four intermediate nodes, namely, B_1, B_2, B_3, B_4, which apply sampled operations on the input node B_0. B_0 is from the output of the last cell. The output node concatenates the outputs of the four intermediate nodes. (**b**) Gabor filter. (**c**) A generic denoising block. Following [81], it wraps the denoising operation with a 1 × 1 convolution and an identity skip connection [36]

$\Omega^{(i,j)} = \{o_1^{(i,j)}, \dots, o_K^{(i,j)}\}$. $\{\Omega^{(i,j)}\}$ is the search space of a cell. Each node B_j takes its dependent nodes as input and can be obtained by $B_j = \Sigma_{i<j} o^{(i,j)}(B_i)$. The constraint $i < j$ here is to avoid cycles in a cell. Each cell takes the output of the last cell as input. For brevity, we denote by B_0 the last node of the previous cell and the first node of the current cell. Unlike existing approaches that use only normal and reduction cells, we search for v ($v > 2$) cells instead. For general NAS search, we follow [56] and take seven normal operations, i.e., 3×3 max pooling, 3×3 average pooling, skip connection (identity), 3×3 convolution with rate 2, 5×5 convolution with rate 2, 3×3 depth-wise separable convolution, and 5×5 depth-wise separable convolution. Considering adversarially robust optimization for NAS, we introduce two additional operations, the 3×3 Gabor filter and denoising block, for model defense. Therefore, the size of the entire search space is $K^{|\mathcal{E}_\mathcal{M}| \times v}$, where $\mathcal{E}_\mathcal{M}$ is the set of possible edges with M intermediate nodes in the fully connected DAG. In the case with $M = 4$ and $v = 6$, together with the input node, the total number of cell structures in the search space is $9^{(1+2+3+4)\times 6} = 9^{10\times 6}$. Here, we briefly introduce the two additional operations.

Gabor filter Gabor filters [24, 25] containing frequency and orientation representations can characterize the spatial frequency structure in images while preserving spatial relationships. This operation provides superb robustness for the network [64]. Gabor filters are defined as $\exp(-\frac{x'^2+\gamma^2 y'^2}{2\sigma^2})\cos(2\pi\frac{x'}{\lambda} + \psi)$. Here, $x' = x\cos\theta + y\sin\theta$ and $y' = -x\sin\theta + y\cos\theta$. σ, γ, λ, ψ, and θ are learnable parameters. Note that the symbols used here apply only to the Gabor filter and are different from the symbols used in the rest of this paper. Figure 3.2b shows an example of Gabor filters.

Denoising block As described in [81], adversarial perturbations on images will introduce noise in the features. Therefore, denoising blocks can improve adversarial robustness by denoising features. Following this, we add the nonlocal mean denoising block [5] as shown in Fig. 3.2c to the search space to denoise the features. Calculate a denoised feature map z of an input feature map x by taking a weighted mean of the spatial locations of the features in general $\mathcal{L}$ as $z_p = \frac{1}{C(x)} \sum_{\forall q \in \mathcal{L}} f(x_p, x_q) \cdot x_q$, where $f(x_p, x_q)$ is a feature-dependent weighting function and $C(x)$ is a normalization function.

3.2.1.3 Anti-bandit Strategy for NAS

As described in [85, 89], the validation accuracy ranking of different network architectures is not a reliable indicator of the final quality of the architecture. However, the experimental results suggest that if an architecture performs poorly at the beginning of training, there is little hope that it can be part of the final optimal model [89]. As training progresses, this observation becomes more and more specific. Based on this observation, we derive a simple but effective training strategy. During training and the increasing epochs, we progressively abandon the worst-performing operation and sample the operations with little expectations but a significant variance for each edge. Unlike [89], which uses the performance as the evaluation metric to decide which operation should be pruned, we use the anti-bandit algorithm described in Sect. 3.2.1.1 to decide.

Following UCB in the bandit algorithm, we obtain the initial performance for each operation on every edge. Specifically, we sample one of the K operations in $\Omega^{(i,j)}$ for every edge, and then obtain the validation accuracy a, which is the initial performance $m_{k,0}^{(i,j)}$ by adversarially training the sampled network for one epoch and finally assigning this accuracy to all the sampled operations.

By considering the confidence of the kth operation using Eq. 3.8, the LCB is calculated by:

$$s_L(o_k^{(i,j)}) = m_{k,t}^{(i,j)} - \sqrt{\frac{2\log N}{n_{k,t}^{(i,j)}}}, \tag{3.9}$$

where N is the total number of samples, $n_{k,t}^{(i,j)}$ refers to the number of times the kth operation of the edge (i, j) has been selected, and t is the epoch index. The first item in Eq. 3.9 is the value term (see Eq. 3.2) which favors the operations that look good historically. The second is the exploration term (see Eq. 3.3), which allows operations to get an exploration bonus that grows with $\log N$. The selection probability for each operation is defined as:

$$p(o_k^{(i,j)}) = \frac{\exp\{-s_L(o_k^{(i,j)})\}}{\sum_m \exp\{-s_L(o_m^{(i,j)})\}}. \tag{3.10}$$

The minus sign in Eq. 3.10 means we prefer to sample operations with smaller confidence. After sampling one operation for every edge based on $p(o_k^{(i,j)})$, we obtain the validation accuracy a by training adversarially the sampled network for one epoch and then update the performance $m_{k,t}^{(i,j)}$ that historically indicates the validation accuracy of all the sampled operations $o_k^{(i,j)}$ as:

$$m_{k,t}^{(i,j)} = (1-\lambda)m_{k,t-1}^{(i,j)} + \lambda * a, \tag{3.11}$$

where λ is a hyperparameter.

Finally, after $K * T$ samples where T is a hyperparameter, we calculate the confidence with the UCB according to Eq. 3.8 as:

$$s_U(o_k^{(i,j)}) = m_{k,t}^{(i,j)} + \sqrt{\frac{2\log N}{n_{k,t}^{(i,j)}}}. \tag{3.12}$$

The operation with minimal UCB for every edge is abandoned. This means that operations that are given more opportunities but result in poor performance are removed. With this pruning strategy, the search space is significantly reduced from $|\Omega^{(i,j)}|^{10\times 6}$ to $(|\Omega^{(i,j)}|-1)^{10\times 6}$, and the reduced space becomes:

$$\Omega^{(i,j)} \leftarrow \Omega^{(i,j)} - \{\arg\min_{o_k^{(i,j)}} s_U(o_k^{(i,j)})\}, \forall (i,j). \tag{3.13}$$

The reduction procedure is repeated until the optimal structure is obtained, where only one operation is left on each edge.

Complexity Analysis There are $\mathcal{O}(K^{|\mathcal{E}_\mathcal{M}|\times v})$ combinations in the search space discovery process with v types of different cells. In contrast, ABanditNAS reduces the search space for every $K * T$ epoch. Therefore, the complexity of the proposed method is the following:

$$\mathcal{O}(T \times \sum_{k=2}^{K} k) = \mathcal{O}(TK^2). \tag{3.14}$$

3.2.1.4 Adversarial Optimization

The goal of adversarial training [58] is to learn networks that are robust to adversarial attacks. Given a network f_θ parameterized by θ, a dataset (x_e, y_e), a loss function l, and a threat model Δ, the learning problem can be formulated as the following optimization problem: $\min_\theta \sum_e \max_{\delta\in\Delta} l\big(f_\theta(x_e+\delta), y_e\big)$, where δ is the adversarial perturbation. In this paper, we consider the typical l_∞ threat model [58], $\Delta = \{\delta : \|\delta\|_\infty \le \epsilon\}$ for some $\epsilon > 0$. Here, $\|\cdot\|_\infty$ is the l_∞ norm distance metric and ϵ is the adversarial manipulation budget. The adversarial training procedure uses attacks to approximate inner maximization over Δ, followed by some variation of gradient descent on model parameters θ. For example, one of the earliest versions of adversarial training uses the fast gradient sign method (FGSM) [29] to approximate the inner maximization. This could be seen as a relatively inaccurate approximation of inner maximization for l_∞ perturbations, and it has the closed-form solution: $\theta = \epsilon \cdot \text{sign}\Big(\nabla_x l\big(f(x), y\big)\Big)$. A better approximation of inner maximization is to take multiple smaller FGSM steps of size α instead. However, the number of gradient computations caused by the multiple steps is proportional to $\mathcal{O}(EF)$ in a single epoch, where E is the size of the dataset and F is the number of steps taken by the adversary PGD. This is F times higher than standard training with $\mathcal{O}(E)$ gradient computations per epoch, and adversarial training is typically F times slower. To accelerate adversarial training, we combine FGSM with random initialization [77] for our ABanditNAS. Our ABanditNAS with adversarial training is summarized in Algorithm 3.

Algorithm 3: ABanditNAS with adversarial training

```
Input: Training data, validation data, searching hyper-graph, adversarial perturbation $\delta$,
       adversarial manipulation budget $\epsilon$, $K = 9$, hyper-parameters $\alpha$, $\lambda = 0.7$, $T = 3$.
Output: The remaining optimal structure;
$t = 0; c = 0;$
Get initial performance $m_{k,0}^{(i,j)}$;
while ($K > 1$) do
    $c \leftarrow c + 1;$
    $t \leftarrow t + 1;$
    Calculate $s_L(o_k^{(i,j)})$ using Eq. 3.9;
    Calculate $p(o_k^{(i,j)})$ using Eq. 3.10;
    Select an architecture by sampling one operation based on $p(o_k^{(i,j)})$ from $\Omega^{(i,j)}$ for
    every edge;
    # Train the selected architecture adversarially:
    for $e = 1, \ldots, E$ do
        $\delta = \text{Uniform}(-\epsilon, \epsilon);$
        $\delta \leftarrow \delta + \alpha \cdot \text{sign}\left(\nabla_x l\big(f(x_e + \delta), y_e\big)\right);$
        $\delta = \max\big(\min(\delta, \epsilon), -\epsilon\big);$
        $\theta \leftarrow \theta - \nabla_\theta l\big(f_\theta(x_e + \delta), y_e\big).$
    end
    Get the accuracy $a$ on the validation data;
    Update the performance $m_{k,t}^{(i,j)}$ using Eq. 3.11;
    if $c = K * T$ then
        Calculate $s_U(o_k^{(i,j)})$ using Eq. 3.12;
        Update the search space $\{\Omega^{(i,j)}\}$ using Eq. 3.13;
        $c = 0;$
        $K \leftarrow K - 1.$
    end
end
```

3.2.1.5 Analysis

Effect on the hyperparameter λ The hyperparameter λ balances the performance between the past and the current. Different values of λ result in similar search costs. The performance of the structures searched by ABanditNAS with different values of λ is used to find the best λ. We train the structures in the same setting. From Fig. 3.3, we can see that when $\lambda = 0.7$, ABanditNAS is most robust.

Effect on the search space We test the performance of ABanditNAS with different search spaces. We adopt the same experimental setting as the general NAS in this part. The search space of the general NAS has seven operations. We incrementally add the Gabor filter, denoising block, 1×1 dilated convolution with rate 2, and 7×7 dilated convolution with rate 2 until the number of operations in the search space reaches 11. In Table 3.1, # Search Space represents the number of operations in the search space. Although the search difficulty increases with increasing search space, ABanditNAS can effectively select the appropriate operations. Each additional

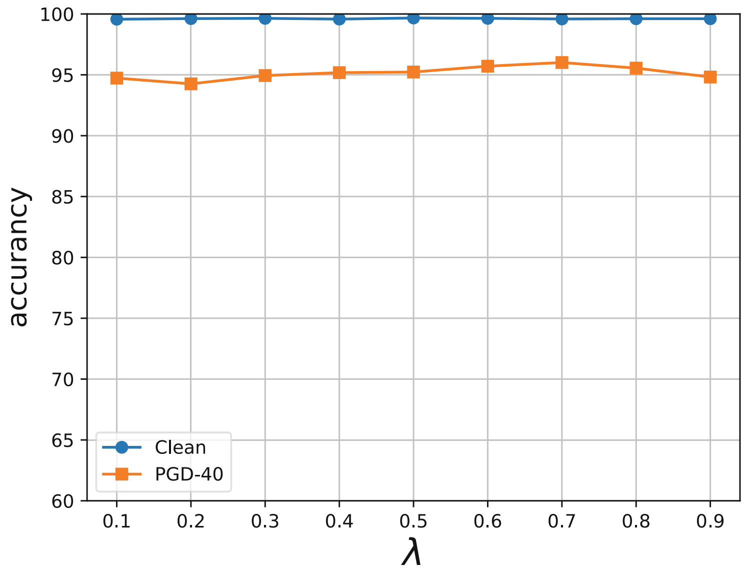


Fig. 3.3 Performances of structures searched by ABanditNAS with different hyperparameter values λ

Table 3.1 The performance of ABanditNAS with different search spaces on CIFAR10

Architecture	# Search space	Accuracy (%)	# Params (M)	Search cost (GPU days)	Search method
ABanditNAS	7	97.13	3.0	**0.09**	Anti-bandit
ABanditNAS	8	97.47	3.3	0.11	Anti-bandit
ABanditNAS	9	97.52	4.1	0.13	Anti-bandit
ABanditNAS	10	97.53	**2.7**	0.15	Anti-bandit
ABanditNAS	11	**97.66**	3.7	0.16	Anti-bandit

operation has little effect on search efficiency, demonstrating the efficiency of our search method. When the number of operations in the search space is 9, the classification accuracy of the model searched by ABanditNAS exceeds all the methods with the same level of search cost.

3.2.2 IDARTS: Interactive Differentiable Architecture Search

In part, NAS has significantly impacted computer vision by reducing the need for manual work. Recently, Liu et al. [56] proposed differentiable architecture search (DARTS) as an alternative that makes architecture search more efficient. DARTS relaxes the search space to be continuous and differentiable. DARTS learns the weight of each operation with gradient descent so that the architecture can be optimized concerning its validation set performance by gradient descent. Despite its sophisticated design, DARTS is still subject to an ample yet redundant space of network architectures and thus suffers from significant memory and computation overhead. To address the problems of DARTS, researchers have proposed alternative formulations, including PDARTS [16], DARTS+ [49], PC-DARTS [82], ProxylessNAS [9], CDARTS [86], Fair DARTS [20], and SGAS [47]. Among them, PC-DARTS [82] reduces redundancy in the network space by performing a more efficient search without compromising performance. PC-DARTS only samples a subset of channels in a super-net during the search to reduce computation and introduces edge normalization to stabilize the search for network connectivity by explicitly learning an extra set of edge-selection parameters.

However, these DARTS alternatives need to pay more attention to the intrinsic relationship between different parameters, and as a result, the selected architecture could be more robust due to an insufficient training process. The reason is that the coupling relationship will affect the training of the network architecture to its limit before it is selected or left out. To address this issue, we introduce a bilinear model into DARTS and develop a new backpropagation method to decouple the hidden relationships among variables to facilitate the optimization process. To the best of our knowledge, few works have formulated DARTS as a bilinear problem.

We address these issues by formulating DARTS as a bilinear optimization problem and developing the efficient interactive differentiable architecture search. Figure 3.4 shows the framework of IDARTS [84]. Figure 3.4b shows that the dotted line results are inefficient compared with IDARTS shown in the solid line. t_1, t_2 marks the results where the architecture parameter α is backtracked. IDARTS coordinates the training of different parameters and fully explores their interaction based on the backtracking method. Our method allows operations to be selected only

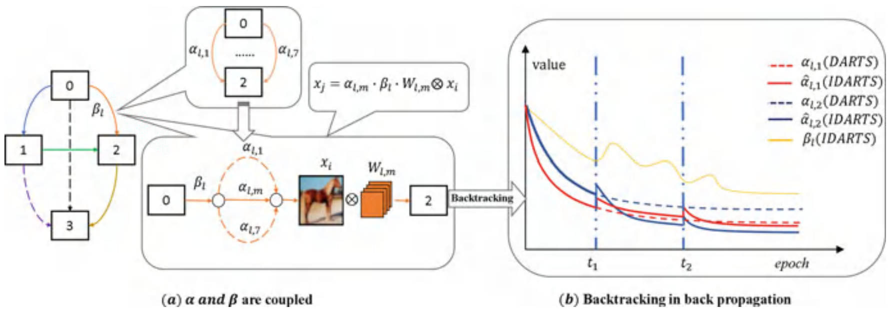


Fig. 3.4 An overview of IDARTS. (**a**) α and β are coupled in IDARTS. The edge and operation (β_l and α_l) are coupled during the neural architecture search. x_i and x_j represent node 0 and node 2, respectively. $x_j = \alpha_{l,m} \cdot \beta l \cdot W_{l,m} \otimes x_i$ is specifically described in Eq. 3.16. (**b**) A backtracking method is introduced to coordinate the training of different parameters, which can fully explore their interaction during training. The dotted line results indicate that the lack of backtracking leads to the inadequate training of α, and the solid line indicates an efficient training of IDARTS

when they are sufficiently trained. We evaluate our IDARTS on image classification and conduct experiments on the CIFAR10 and ImageNet datasets. The experimental results show that IDARTS achieves superior performance compared to existing DARTS approaches, including PC-DARTS [82], CDARTS [86], and FairDARTS [20].

3.2.2.1 Bilinear Models for DARTS

We first show how DARTS can be formulated as a bilinear optimization problem. Assume that there are L edges in a cell, and the edge between node N_i and node N_j is the lth edge. Following [56, 82], we take the lth edge, which is formulated as:

$$f_l(\mathbf{W}_{l,m}, \mathbf{x_i}) = \sum_{o_{l,m} \in O_{(l)}} \alpha_{l,m} \cdot o_{l,m}(\mathbf{W}_{l,m} \otimes \mathbf{x_i}), \tag{3.15}$$

where $\mathbf{W}_{l,m}$ denotes the kernels of the mth convolution operation. We assume that there are M operations on one edge. M refers to the number of all operations. $\mathbf{x_i}$ denotes the feature map of N_i, O_l denotes the set of operations, and $\alpha_{l,m}$ is the parameter of operation $o_{l,m}$ on lth edge processed by softmax operation:

$$\begin{aligned}\mathbf{x_j} &= \sum_{i<j} \{\beta_l\} \cdot f_l(x_i) \\ &= \sum_{i<j} \sum_{o_{l,m} \in O_l} \beta_l \alpha_{l,m} \cdot o_{l,m}(\mathbf{W_{l,m}} \otimes \mathbf{x_i}),\end{aligned} \tag{3.16}$$

where β_l denotes the parameter of lth edge. To calculate the final architecture, the softmax is defined on β and α. For each intermediate node, we will choose two edges, which are jointly determined by α and β. In Fig. 3.4, we see α and β are coupled in the inference process as shown in Eq. 3.16. x_j is linearly dependent on both α and β, a classic bilinear problem. If an improper operation is selected, it will affect the selection of the edge and vice versa. It suggests that we should consider their relationship for better optimization. A basic bilinear optimization problem attempts to optimize the following objective function in the architecture search:

$$\arg\min_{\boldsymbol{\beta},\boldsymbol{\alpha}} G(\mathbf{W}, \boldsymbol{\beta}, \boldsymbol{\alpha}) = \arg\min_{\boldsymbol{\beta},\boldsymbol{\alpha}} (\mathcal{L}(\mathbf{W}, \boldsymbol{\beta}, \boldsymbol{\alpha}) + R(\boldsymbol{\beta})), \tag{3.17}$$

where $\alpha \in \mathbb{R}^{L\times M}$ and $\beta \in \mathbb{R}^{L\times 1}$ are variables to be optimized, α denotes the matrix, L is the number of edges, M is the number of operations at each edge, and $R(\cdot)$ represents the constraint, which determines where the backtracking occurs. $\mathcal{L}(\cdot)$ denotes the loss function in the original DARTS models.

Following [56, 82], the weights of the kernels **W** and the architectural parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ are optimized sequentially. The learning procedure for the architectural parameters involves an optimization as:

$$\begin{aligned} \mathbf{W^{t+1}} &= \arg\min_{\mathbf{W}} \mathcal{L}_{train}(\mathbf{W^t}, \alpha^t, \beta^t), \\ \alpha^{t+1} &= \arg\min_{\alpha} \mathcal{L}_{val}(\mathbf{W^{t+1}}, \alpha^t, \beta^t), \\ \beta^{t+1} &= \arg\min_{\beta} \mathcal{L}_{val}(\mathbf{W^{t+1}}, \alpha^t, \beta^t), \end{aligned} \tag{3.18}$$

where α^{t+1} and β^{t+1} denote the parameters of operation and edge in the $(t+1)$th step, and $\mathbf{W^{t+1}}$ denotes the kernel of the convolution at the $(t+1)$th step.

In Eq. 3.18, α and β are updated independently. However, optimizing α and β independently is improper due to their coupling relationship. We consider the search process of differentiable architecture search as a bilinear optimization problem and solve the problem using a new backtracking method. The details will be shown in Sect. 3.2.2.3.

3.2.2.2 Search Space

By simplifying the architecture search to find the best cell structure, cell-based NAS methods try to learn a scalable and transferable architecture. Following [56, 82], we search for normal and reduction computation cells to build the final architecture. The reduction cells are located at 1/3 and 2/3 of the total network depth; the rest are normal cells. A normal cell uses operations with a stride of 1 to keep the size of the input feature map unchanged. The number of output channels is identical to the number of input channels. A reduction cell uses operations with a stride of 2 to reduce the spatial resolution of feature maps, and the number of output channels is twice the number of input channels. The set of operations includes 3×3 and 5×5 separable convolution, 3×3 and 5×5 dilated separable convolution, 3×3 max pooling, 3×3 average pooling, a zero(none), and a skip connection. A cell (Fig. 3.5) is a fully connected directed acyclic graph (DAG) of *seven* nodes. Each $\mathbf{x_i}$ is a latent representation (e.g., a feature map in convolutional networks). Each directed edge (i, j) between node N_i and node N_j denotes the set of operations $\mathcal{O}_l = \{o_{l,1}, \ldots, o_{l,M}\}$. Following [56], there are 2 input nodes, 4 intermediate nodes, 1 output node, and 14 edges per cell during the search. Each cell takes the outputs of the two previous cells as the input. The output node of a cell is the depth-wise concatenation of all of the intermediate nodes.

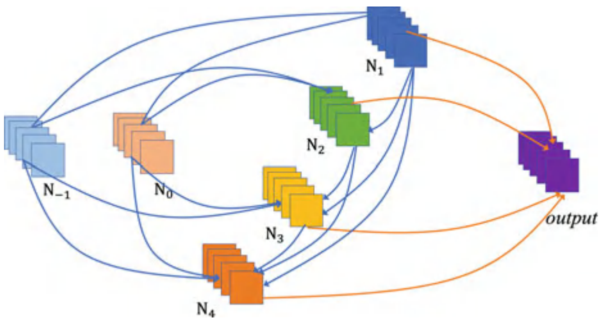


Fig. 3.5 A cell contains seven nodes, which are two input nodes N_{-1} and N_0; four intermediate nodes N_1, N_2, N_3, and N_4; and one output node

3.2.2.3 Backtracking Back Propagation

We consider the problem from a new perspective where the $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$ are coupled in Eq. 3.17. We note that the calculation of the derivative of α should consider its coupling relationship with β. Based on the chain rule [61] and its notations, we have:

$$\hat{\boldsymbol{\alpha}}^{t+1} = \boldsymbol{\alpha}^{\boldsymbol{t}} + \eta_1 \left(\frac{\partial G}{\partial \boldsymbol{\alpha}} + \eta_2 Tr \left(\left(\frac{\partial G}{\partial \boldsymbol{\beta}} \right)^T \frac{\partial \boldsymbol{\beta}}{\partial \boldsymbol{\alpha}} \right) \right), \tag{3.19}$$

where η_1 represents the learning rate, η_2 represents the coefficient of backtracking, $\hat{\alpha}_{t+1}$ denotes the value backtracked from α_{t+1}, and $Tr(\cdot)$ represents the trace of the matrix. $Tr(\cdot)$ means that each element in the matrix $\frac{\partial G}{\partial \boldsymbol{\alpha}}$ adds the trace of the corresponding matrix related to $\boldsymbol{\alpha}$. Here, $\mathbf{W}$ is omitted for simplicity, and only structure parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$ are considered during the back propagation process. We further define:

$$\hat{G}(\boldsymbol{\beta}, \boldsymbol{\alpha}) = \left(\frac{\partial G}{\partial \boldsymbol{\beta}} \right)^T / \boldsymbol{\alpha}, \tag{3.20}$$

where $\hat{G}$ is defined by considering the bilinear optimization problem as in Eq. 3.17. Note that $R(\cdot)$ is only considered when backtracking. Then we have:

$$\frac{\partial G(\beta, \alpha)}{\partial \boldsymbol{\alpha}} = Tr \left[\alpha \hat{G} \frac{\partial \boldsymbol{\beta}}{\partial \alpha} \right]. \tag{3.21}$$

We denote $\hat{G} = [\hat{g}_1, \ldots, \hat{g}_L]$. Assuming that $\boldsymbol{\beta}_l$ and α_m are independent when $l \neq m$, α_m denotes a column vector, and $\alpha_{1,m}$ denotes an element in matrix α, we have:

$$\frac{\partial \boldsymbol{\beta}}{\partial \alpha} = \begin{bmatrix} 0 \ldots \frac{\partial \boldsymbol{\beta}_m}{\partial \alpha_{1,m}} \ldots 0 \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ 0 \ldots \frac{\partial \boldsymbol{\beta}_m}{\partial \alpha_{L,m}} \ldots 0 \end{bmatrix}, \tag{3.22}$$

and

$$\boldsymbol{\alpha}\hat{G} = \begin{bmatrix} \alpha_1\hat{g}_1 \ldots \alpha_1\hat{g}_l \ldots \alpha_1\hat{g}_L \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \alpha_L\hat{g}_1 \ldots \alpha_L\hat{g}_l \ldots \alpha_L\hat{g}_L \end{bmatrix}. \tag{3.23}$$

We combine Eqs. 3.22 and 3.23 and get:

$$\boldsymbol{\alpha}\hat{G}\frac{\partial \boldsymbol{\beta}}{\partial \alpha} = \begin{bmatrix} 0 \ldots \alpha_1 \sum_{l=1}^{L} \hat{g}_l \frac{\partial \boldsymbol{\beta}_m}{\partial \alpha_{l,m}} \ldots 0 \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ 0 \ldots \alpha_L \sum_{l=1}^{L} \hat{g}_l \frac{\partial \boldsymbol{\beta}_m}{\partial \alpha_{l,m}} \ldots 0 \end{bmatrix}. \tag{3.24}$$

After that, the trace of Eq. 3.19 is then calculated by:

$$Tr\left[\boldsymbol{\alpha}^t\hat{G}\frac{\partial \boldsymbol{\beta}}{\partial \alpha_m}\right] = \alpha_m \sum_{l=1}^{L} \hat{g}_l \frac{\partial \boldsymbol{\beta}_m}{\partial \alpha_{l,m}}. \tag{3.25}$$

Remembering that $\boldsymbol{\alpha}^{t+1} = \boldsymbol{\alpha}^t + \eta_1 \frac{\partial G(\beta,\alpha)}{\partial \boldsymbol{\alpha}}$, IDARTS combines Eqs. 3.19 and 3.25:

$$
\begin{aligned}
\hat{\boldsymbol{\alpha}}^{t+1} &= \boldsymbol{\alpha}^{t+1} + \eta \begin{bmatrix} \sum_{l=1}^{L} \hat{g}_l \frac{\partial \boldsymbol{\beta}_1}{\partial \alpha_{l,1}} \\ \cdot \\ \cdot \\ \cdot \\ \sum_{l=1}^{L} \hat{g}_l \frac{\partial \boldsymbol{\beta}_L}{\partial \alpha_{l,L}} \end{bmatrix} \odot \begin{bmatrix} \alpha_1 \\ \cdot \\ \cdot \\ \cdot \\ \alpha_L \end{bmatrix} \\
&= \boldsymbol{\alpha}^{t+1} + \eta \begin{bmatrix} < \hat{G}, \frac{\partial \boldsymbol{\beta}_1}{\partial \alpha_1} > \\ \cdot \\ \cdot \\ \cdot \\ < \hat{G}, \frac{\partial \boldsymbol{\beta}_L}{\partial \alpha_L} > \end{bmatrix} \odot \begin{bmatrix} \alpha_1 \\ \cdot \\ \cdot \\ \cdot \\ \alpha_L \end{bmatrix} \\
&= \boldsymbol{\alpha}^{t+1} + \eta \boldsymbol{\gamma} \odot \boldsymbol{\alpha}^t,
\end{aligned} \tag{3.26}
$$

where $\odot$ represents the Hadamard product and $\eta = \eta_1 \eta_2$. To simplify the calculation, $\frac{\partial \boldsymbol{\beta}}{\partial \alpha}$ can be approximated by $\frac{\Delta \boldsymbol{\beta}}{\Delta \alpha}$. Equation 3.26 shows our method is based on a projection function to solve the coupling problem of the bilinear optimization by γ. In this method, we consider the influence of $\boldsymbol{\alpha}^t$ and backtrack the optimized state at the $(t+1)th$ step to form $\hat{\boldsymbol{\alpha}}^{t+1}$. We first decide when the optimization should be backtracked, and the update rule of the proposed IDARTS is defined as:

$$
\hat{\boldsymbol{\alpha}}^{t+1} = \begin{cases} P(\boldsymbol{\alpha}^{t+1}, \boldsymbol{\alpha}^t) & if\ R(\beta) < \zeta, \\ \boldsymbol{\alpha}^{t+1} & otherwise, \end{cases} \tag{3.27}
$$

where $P(\boldsymbol{\alpha}^{t+1}, \boldsymbol{\alpha}^t) = \boldsymbol{\alpha}^{t+1} + \eta \boldsymbol{\gamma} \odot \boldsymbol{\alpha}^t$. $R(\beta)$ represents the ranking of $|\beta_l|$ and ζ represents the threshold. We then have:

$$
\zeta = \lfloor (S - T) \cdot \lambda \cdot L \rfloor, \tag{3.28}
$$

where T and S denote the beginning and ending epoch of backtracking, λ denotes the coefficient, and L denotes the number of edges in a cell. As shown in Eq. 3.28, ζ will be increased during searching. By doing so, α will be backtracked, according to β.

3.2.2.4 Comparison of Searching Methods

Figure 3.6 illustrates the comparison of α for IDARTS and PC-DARTS in the shallowest edge. The label of the x-axis is the epoch in searching, and the label of the y-axis is the value of α. We freeze the hyperparameters, α and β, in the first

Algorithm 4: IDARTS interactive differentiable architecture search

```
Input: Training data, validation data, searching hyper-graph, hyper-parameters $K = 0$,
       $T = 25$, $S = 50$;
Create architectural parameters $\alpha = \alpha_l$, edge level parameters $\beta = \beta_l$ and supernet weights $W$
Create a mixed operation $o_l$ parameterized by $\alpha_l$ and $\beta_l$ for each edge $l$;
Output: The structure;
Search for an architecture for S epochs; while $(K \leq S)$ do
    Update parameters $\alpha$ and $\beta$;
    if $(K \geq T)$ then
        According to Eq. 3.27, we select $\alpha$ that should be backtracked;
        backtracking $\alpha$ by Eq. 3.26;
    end
    Update weights $W$;
    $K \leftarrow K + 1$;
    Find the final architecture based on the learned $\alpha$ and $\beta$;
end
```

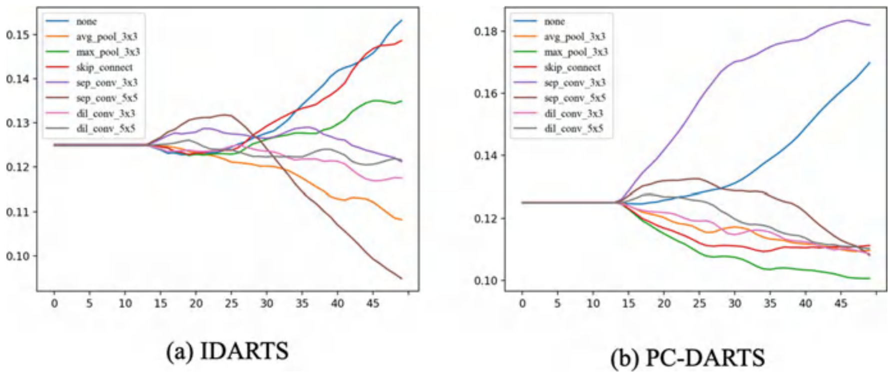


Fig. 3.6 Comparison of α values in the shallowest edge of IDARTS and PC-DARTS on CIFAR10

15 epochs (only network parameters are updated), α remains unchanged. As the shortage of interaction between α and β in PC-DARTS, α and β might easily fall into the local minima. However, we backtrack the insufficiently trained operations on this edge to escape from the local minima to select a better operation and, thus, a better architecture by considering the intrinsic relationship between α and β. Due to the backtracking of α, the competition between different operations is intensified in the IDARTS search process, as shown in Fig. 3.6. As a result, it is more conducive to choosing the most valuable operation than PC-DARTS. In Fig. 3.7, the label of y-axis is $\mathcal{L}_{val}$. We also show that the convergence of IDARTS is similar to that of PC-DARTS. Although the two have the same convergence rate, we can see that the final loss of IDARTS converges to a smaller value. The main reason is that IDARTS has explored the relationship between different parameters and used our backtracking method to train the architecture parameter α entirely. We theoretically

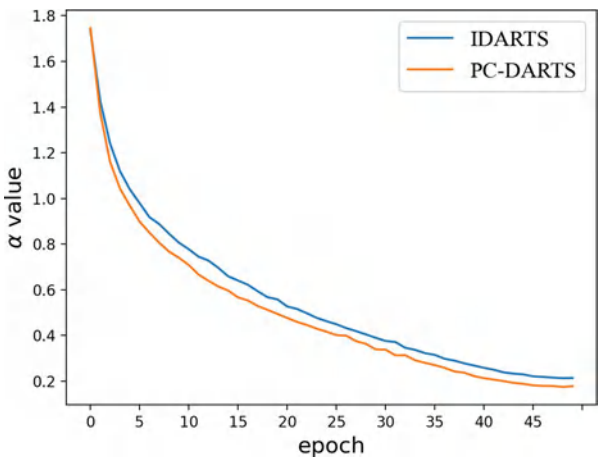


Fig. 3.7 Comparison of searching loss on CIFAR10 with IDARTS and PC-DARTS

derive our method under the framework of gradient descent, which provides a solid foundation for the convergence analysis of our method.

3.2.3 Fast and Unsupervised Neural Architecture Evolution for Visual Representation Learning

Learning high-level representations from labeled data and deep learning models in an end-to-end manner is one of the biggest successes in computer vision in recent history. These techniques make manually specified features redundant and significantly improve the state of the art for many real-world applications. Many challenges remain, however, such as the cost of annotating large datasets and an insufficient ability to generalize the model. For example, a learned representation from supervised learning for image classification may lack information such as texture, which matters little for classification but can be more relevant for later tasks. Yet adding it makes the representation less general and might be irrelevant for tasks such as image captioning. Thus, improving representation learning requires features to be focused on solving a specific task. Unsupervised learning is an important stepping stone towards robust and generic representation learning [34]. The main challenge is a significant performance gap compared with supervised learning.

In a scenario where we cannot obtain sufficient annotation, self-supervised learning is a popular approach to leverage the mutual information of unlabeled data for training. However, its performance still needs improvement compared with the supervised methods. One obstacle is that only parameters are learned in conventional self-supervised methods. To break the performance bottleneck, a natural idea is to explore NAE to optimize the architectures along with parameter training. Specifically, we can initialize with an architecture found using NAS on a small supervised dataset and then evolve the architecture on a larger dataset using unsupervised learning. Currently, existing architecture evolution methods [7, 94] could be more efficient and cannot deal effectively with the challenging unsupervised representation learning. Our approach is highly efficient with a complexity of $O(n^2)$ where n is the size of the operation space.

Here we propose our fast and unsupervised neural architecture evolution (FaU-NAE) [83] method to search architectures for representation learning. Although UnNAS [54] discusses the value of a label and discovers that labels are not necessary for NAS, it cannot solve the problems above because it is computationally expensive and is trained using supervised learning for real applications. FaUNAE is introduced to evolve an architecture from an existing architecture manually designed or searched from one small-scale dataset on another large-scale dataset. This partial optimization can utilize the existing models to reduce the search cost and improve search efficiency. The strategy is more practical for real applications, as it can efficiently adapt to new scenarios with minimal requirements for data labeling.

First, we adopt a trial-and-test method to evolve the initial architecture, which is more efficient than the traditional evolution methods, which are computationally expensive and require large amounts of labeled data. Second, we note that the quality of the architecture could be better estimated due to the absence of labeled data. To address this, we explore contrastive loss [34] as the evaluation metric for the operation evaluation. Although our method is built based on contrastive loss [34], we model our method on the teacher-student framework to mimic the supervised learning and then estimate the operation performance even without annotations. Then the architecture can be evolved based on the estimated performance. Third, we address that one bottleneck in NAS is its explosive search space of up to 14^8. The search space issue is even more challenging for unsupervised NAS built on an ambiguous performance estimation that further deteriorates the training process. To address this issue, we follow the principle of survival of the fittest and eliminating the inferior to build our search algorithm. This significantly improves search efficiency. Our framework is shown in Fig. 3.8.

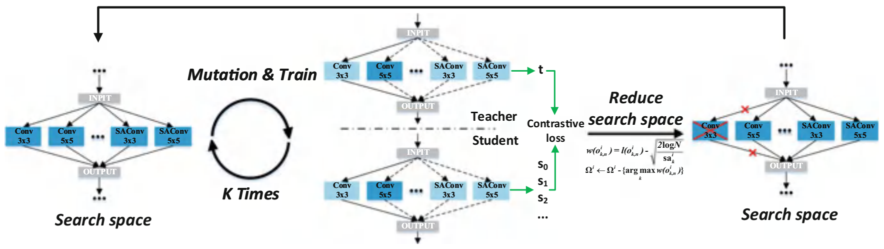


Fig. 3.8 The main framework of the proposed teacher-student search strategy

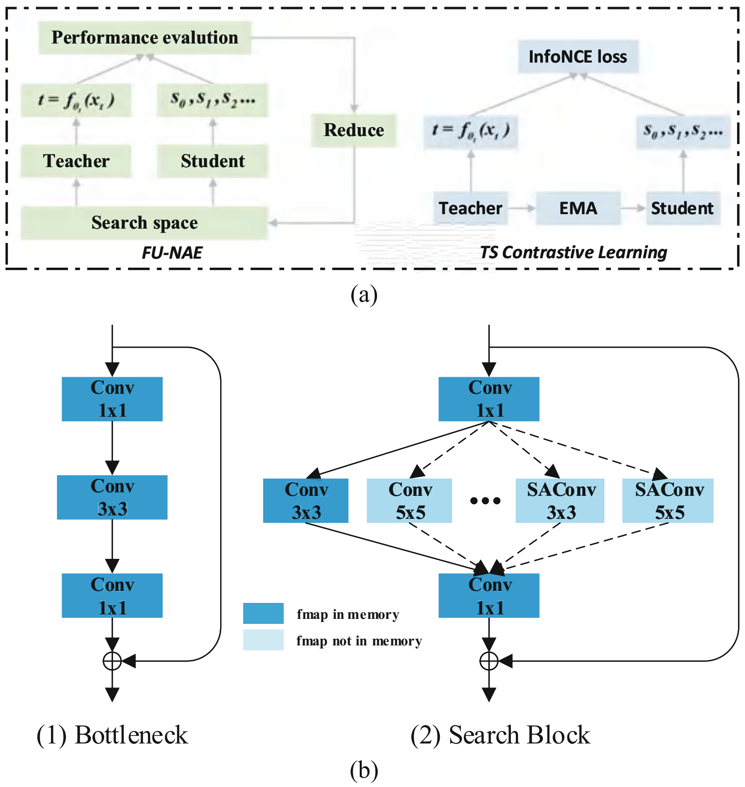


Fig. 3.9 (**a**) The main framework of teacher-student model, which focuses on both the unsupervised neural architecture evolution (left) and contrastive learning (right). (**b**) Compared with the original bottleneck (1) in ResNet, a new search block is designed for FaUNAE (2)

3.2.3.1 Search Space

We have experimentally determined that for unsupervised learning, ResNet [36] is better than cell-based methods for building an architectural space. We denote this space as $\{\Omega^i\}$, where i represents given block. Rather than repeating the bottleneck (building block in ResNet) with various operations, however, we allow a set of search blocks shown in Fig. 3.9b with various operations including traditional convolution with kernel sizes $\{3, 5\}$ and split-attention convolution [87] with kernel sizes $\{3, 5\}$ and radixes $\{2, 4\}$. This reduces the model size by sharing the 1×1 convolution to improve the efficiency. To enable a direct trade-off between depth and block size (indicated by the parameters of the selected operations), we initiate a deeper over-parameterized network and allow a block to be skipped by adding the identity operation to the candidate set of its mixed operation. So the set of the operations Ω^i in the ith block consists of $M = 7$ operations. With a limited model size, the network can either be shallower by skipping more blocks and using larger ones or choose to be deeper by keeping smaller ones. To accelerate the evolution process and make use of prior knowledge, the initial structure α_0 is first manually designed (e.g., ResNet-50, without weight parameters) or searched for by another NAS (e.g., ProxylessNAS [9]) on different datasets in supervised manner[1], which are then remapping to the search space.

3.2.3.2 Evolution

The evolutionary strategy is summarized in Algorithm 5. Unlike AmoebaNet [66] that evaluates the performance of sub-networks sampled from the search space in a population, our method targets evolving the operation in each block using a trial-and-test manner. We first mutate the operation based on its mutation probability, followed by an evaluation step to make sure the mutation is ultimately used.

Mutation An initial structure α_0 is manually designed (e.g., ResNet-50)[2] or searched by another NAS (e.g., ProxylessNAS [9]) on a different dataset using supervised learning. The initial sub-network f_{θ_s}, which is generated by searching over-parameterized network based on α_0, is then trained using Eq. 3.34 for k steps to obtain the evaluation metric $l(o_k^i)$. A new architecture α_n $(o_{k,n})$ is then constructed from the old architecture α_{n-1} $(o_{k,n-1})$ by a transformation or a mutation. The mutation probability p_{mt} is defined as:

$$p_{\text{mt}}(o_{k,n}^i) = \begin{cases} 1-\epsilon, o_{k,n}^i = o_{k,n-1}^i \\ \dfrac{1}{K-1}(1 - \dfrac{\text{sa}_k^i}{\sum_{k'} \text{sa}_{k'}^i})\epsilon, otherwise \end{cases} \tag{3.29}$$

where sa_k^i represent the sampling times of the operation o_k^i. In general, the operation in each block is kept constant with a probability $1-\epsilon$ in the beginning. For the mutation with the probability of ϵ, younger (less sample time) operations are more likely to be selected. Intuitively, keeping the operation constant can be considered to provide exploitation, while mutations provide exploration [66]. We use two main mutations, the depth mutation and the op mutation, as in AmoebaNet [66], to modify the structure generated by the search space described above.

The operation mutation pays attention to the selection of operations in each block. Once the operation in a block is chosen to mutate, the mutation picks one of the other operations based on Eq. 3.29. The depth mutation can change the depth of the sub-network from the over-parameterized network by setting the operation of one block to the "Identity" operation. We limit the model size as a restriction metric to search more efficiently and evaluate more reasonable operations. The

[1] The evolution is based on unsupervised learning.

[2] No weight parameters

structure can then evolve into a sub-network with the same computational burden. The probability of the restriction metric p^i_{rm} is defined as:

$$p_{\text{rm}}(o^i_{k,n}) = \frac{-\exp \text{MS}(o^i_{k,n})}{\sum_{k'} \exp \text{MS}(o^i_{k',n})}, \tag{3.30}$$

where $\text{MS}(o^i_{k,n})$ represents number of parameters of the kth operation in the ith block. The final evolution probability p that combines p_{mt} and p_{rm} is defined as:

$$p(o^i_{k,n}) = \lambda_1 * p_{\text{mt}}(o^i_{k,n}) + (1 - \lambda_1) * p_{\text{rm}}(o^i_{k,n}), \tag{3.31}$$

where λ_1 is hyperparameter.

Mutation validation After each evolution, the sub-network is trained using Eq. 3.34, and the loss is used as the evaluation metric. We observe the current validation loss a and accordingly update the loss $l(o^i_{k,n})$, which historically indicates the validation loss of all the sampled operations $o^{(i,j)}_k$ as:

$$l(o^i_{k,n}) = \lambda_2 * l(o^i_{k,n-1}) + (1 - \lambda_2) * a, \tag{3.32}$$

where λ_2 is a hyperparameter. If the operation which is mutated performs better (less loss), we apply it as the base of the next evolution; otherwise, we use the original operation as the base of the next evolution:

$$o^i_{k,n} = \begin{cases} o^i_{k,n}, \, l(o^i_{k,n}) > l(o^i_{k,n-1}) \\ o^i_{k,n-1}, \, else \end{cases} \tag{3.33}$$

3.2.3.3 Contrastive Learning

Contrastive learning [31] can significantly improve the performance of unsupervised visual representation learning. The goal is to make positive sample pairs close and negative sample pairs far away in the latent space. Prior works [34, 74] usually investigate contrastive learning by exploring the sample pairs calculated from the encoder and the momentum encoder [34]. Based on the investigation, we reformulate the unsupervised/self-supervised NAS as a teacher-student model, as shown in Fig. 3.9a. Following [34], we build dynamic dictionaries, and the "keys" (e.g., tokens) t in the dictionary are sampled from data (e.g., images or patches) and are represented by the teacher network. In general, the keys representation is $t = f_{\theta_t}(x_t)$, where $f_{\theta_t}(.) = f(o^i_{k,n}; \theta_t; .)$ is a teacher network and x_t is a key sample. Likewise, the "query" x_s is represented by $s = f_{\theta_s}(x_s)$, where $f_{\theta_s} = f(o^i_{k,n}; \theta_s; .)$ is a student network. Unsupervised learning trains the student network to perform

Algorithm 5: FaUNAE

```
Input: Training data, Validation data, initial structure $\alpha_0$
Parameter: Searching hyper-graph
Output: Optimal structure $\alpha$
1: Let $n = 1$.
2: while $(K > 1)$ do
3:     for $t = 1, \ldots, T$ epoch do
4:         Evolve architecture $\alpha_n$ from the old architecture $\alpha_{n-1}$ based on the evolution
           probability $p$ using Eq. 3.31;
5:         Construct the Teacher model and the Student model with the same architecture $\alpha_n$,
           and then train Student models by gradient descent and update the Teacher model by
           EMA using Eq. 3.35;
6:         Get the evaluation loss on the validation data using Eq. 3.34;
           Use Eq. 3.32 to compute the performance and assign that to all the sampled
           operations;
           Update $\alpha_n$ using Eq. 3.33;
7:     end for
8:     if $t == K * E$ then
9:         Update $w(o^i_{k,n})$ using Eq. 3.36;
10:        Reduce the search space: $\Omega^i \leftarrow \Omega^i - \arg\max_k w(o^i_{k,n})$ ;
11:        $K \leftarrow K - 1$;
12:    end if
13: end while
14: return $\alpha$
```

dictionary lookups. An encoded "query" s should be similar to its matching key and dissimilar to others. The student and teacher models are NAE sub-networks from the over-parameterized network described in Sect. 3.2.3.1.

Using a contrastive loss, we train a visual representation student model by matching an encoded query s to a dictionary of encoded keys. The value of the contrastive loss is lower when s and t are from the same (positive) sample and higher when s and t are from different (negative) samples. The contrastive loss is also deployed in FaUNAE to guide structure evolution to obtain the optimal structure based on the unlabeled dataset. InfoNCE [60] shown in Fig. 3.9a measures the similarity using the dot product and is used as our evaluation metric:

$$\mathcal{L} = -\log \frac{\exp(s \cdot t_+/\tau)}{\sum_{n=0}^{N} \exp(s \cdot t_n/\tau)}, \tag{3.34}$$

where τ is a temperature hyperparameter per [80] and t_+ represents the feature calculated from the same sample with s. InfoNCE is over one positive and M negative sample. Intuitively, it is a log loss of a $(M+1)$-way softmax-based classifier that tries to classify s as t_+. Our method is general and can be based on other contrastive loss functions [31, 39, 76, 80], such as margin-based losses and variants of NCE losses.

Following [34, 72], the teacher model is updated as an exponential moving average (EMA) of the student model:

$$\theta_t = m * \theta_t + (1 - m) * \theta_s, \tag{3.35}$$

where θ_s and θ_t are the weight of the student model and teacher model, respectively, updated by back propagation in contrastive learning, and $m \in [0, 1)$ is a smoothing coefficient hyperparameter.

3.2.3.4 Fast Evolution by Eliminating Operations

One of the most challenging aspects of NAS lies in the inefficient search process, and we address this issue by eliminating the least potential operations. After $|\Omega^i| * E$ epochs, we remove the operations in each block based on performances (loss) and the sampling times. We define the combination of the two as:

$$w(o^i_{k,n}) = l(o^i_{k,n}) - \sqrt{\frac{2\log N}{\mathrm{sa}^i_k}}, \tag{3.36}$$

where N is the total number of evolutions and mutations and sa^i_k refers to the number of times the kth operation of the ith block has been selected. The first item $l(o^i_{k,n})$ in Eq. 3.36 is calculated based on an accumulation of the validation loss, which favors the operations that look good historically, and the second term is the exploration term which allows operations to get an exploration bonus that grows as $\log N$. The operation with the minimal w for every block is abandoned. This means that the operations that are given more opportunities, but result in poor performance, are removed. With this strategy, the search space which has v blocks is significantly reduced from $|\Omega^i|^v$ to $(|\Omega^i| - 1)^v$, and the reduced space becomes:

$$\Omega^i \leftarrow \Omega^i - \{\arg\max_k w(o^i_{k,n})\}. \tag{3.37}$$

The reduction procedure is repeated until the optimal structure is obtained when only one operation is left in each block.

3.2.3.5 Experiments

This section compares FaUNAE with human-designed networks and state-of-the-art NAS methods for classification on the ImageNet and CIFAR10 datasets. The evolved architecture on ImageNet is also applied as the backbone of object detection on the PASCAL VOC and COCO datasets. Due to page limitations, the experimental results on CIFAR10 and PASCAL VOC are shown in the supplemental material.

Evolution and Training Protocol The evolution and training protocol used in our experiments is described in this section. We first set global average pooling and a two-layer MLP [14] head (hidden layer 2048-d, with ReLU) which has a fixed-dimensional output (128-d [80]) after the hypernet and searched network. The output vector is normalized by its L2 norm [80], representing the query or key. The temperature τ in Eq. 3.34 is set as 0.2 [80], and the smoothing coefficient hyperparameter m in Eq. 3.35 is set as 0.999. The data augmentation setting follows MoCoV2 [17]. A 224×224-pixel cropped patch is taken from a randomly resized image and is then subjected to random color jittering, random horizontal flip, random grayscale conversion, and blur augmentation [14]. We use the SGD optimizer with an initial learning rate of 0.03 (annealed down to zero following a cosine schedule without restart), a momentum of 0.9, a weight decay of 0.0001, and batch size of 256 in 8 GPUs.

In experiments, we first evolve the initial structure α_0 on an over-parameterized network that uses ResNet50 as the backbone to build the architecture space (details can be found in Sect. 3.2.3.1) on ImageNet. We set initial structure α_0 as a random structure, ResNet50, and structure searched by Proexyless on ImageNet100, respectively, to show the importance of prior knowledge. During the architecture search, the 128M training samples of ImageNet are divided into two subsets, 80% for the training set for training the network weights and the remainder as a validation set for mutation validation and search space reduction. We set the channel as half of that of ResNet50 for efficiency and attention to the evolution of operation rather than the channel. So the model size of search space can be reduced to a quarter, and we set hyperparameter $E = 3$, so the total number of epochs is $\sum_{m=2}^{M} k * E$. The hyperparameter λ_1 and λ_2 are set to 0.9 and 0.3. After evolution, we train the searched network on ImageNet unsupervised for 200 epochs. We run the experiment multiple times and find that the resulting architectures only show slight variation in performance, demonstrating the proposed method's stability.

Results for Classification Following a common protocol, we verify our method by linearly classifying frozen features. In this subsection, we perform unsupervised pre-training on ImageNet, and then we freeze the features and train a supervised linear classifier (a fully connected layer followed by softmax). We train this classifier on the global average pooling features of the evaluated network for 100 epochs. We report Top-1 classification accuracy on the ImageNet validation set. For this experiment, we set the initial learning rate as 30 and weight decay 0 same with [34].

The results for different architectures on ImageNet are summarized in Table 3.2. We use 8 Tesla V100 GPUs to search for about 46 hours. Table 3.2 shows that FaUNAE outperforms ResNet50, ResNet101, ResNet170, and AMDIM$_{\text{small/large}}$ with higher accuracy. FaUNAE also performs better than the structure sampled randomly from the search space described in Sect. 3.2.3.1 on Top-1 accuracy (68.3 vs. 66.2), demonstrating our method's effectiveness. When compared with other NAS methods like Proxyless, which uses the same search space as FaUNAE, our

Table 3.2 Comparisons under the linear classification protocol on ImageNet

Architecture	Method	Accuracy (%)	**Params** (M)	Search cost (GPU days)	Search method
ResNet50	InstDisc [80]	54.0	24	–	Manual
ResNet50	LocalAgg [93]	58.8	24	–	Manual
ResNet101	CPC v1 [39]	48.7	28	–	Manual
$ResNet170_{\text{wider}}$	CPC v2 [37]	65.9	303	–	Manual
$ResNet50_{\text{L+ab}}$	CMC [73]	64.1	47	–	Manual
$AMDIM_{\text{small}}$	AMDIM [3]	63.5	194	–	Manual
$AMDIM_{\text{large}}$	AMDIM [3]	68.1	626	–	Manual
ResNet50	MoCo v1 [34]	60.6	24	–	Manual
ResNet50	MoCo v2 [17]	67.5	24	–	Manual
ResNet50	SimCLR [14]	66.6	24	–	Manual
Random	MoCo v2	66.2	23	–	Random
Proxyless	MoCo v2	67.8	23	23.1	Gradient-base
FaUNAE (Random)	MoCo v2	67.4	24	**15.3**	Evolution
FaUNAE (ResNet50)	MoCo v2	67.8	24	**15.3**	Evolution
FaUNAE (Proxyless)	MoCo v2	68.3	30	**15.3**	Evolution

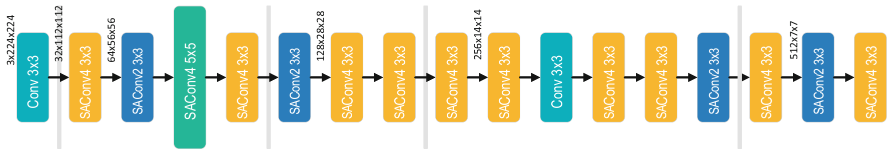


Fig. 3.10 Detailed structures of the best structure discovered on ImageNet. “SAConv2” and “SAConv4” denote split-attention bottleneck convolution layer with radixes of 2 and 4, respectively

method obtains a better performance with higher accuracy (67.8 vs. 68.3) and with a much faster search speed (23.1 vs. 15.3 GPU days).

We also set different initial structures α_0 including random structure, ResNet50, and structure searched by Proxyless on ImageNet100. As shown in Table 3.2, we find that the better the initial structure, the better the performance, which shows the importance of prior knowledge. For the structure (Fig. 3.10) obtained by ABanditNAS on ImageNet, we find that the structure on unsupervised learning prefers a small kernel size and a split-attention convolution [87], which also shows the effective of split-attention convolution and the rationality of FaUNAE.

Results on Object Detection and Segmentation Learning transferable features is the primary goal of unsupervised learning. ImageNet supervised pre-training is most influential when initializing fine-tuning in object detection and segmentation (e.g., [26, 27, 67]). Next, we compare FaUNAE with ImageNet supervised pre-training, transferred to various tasks including PASCAL VOC [22] (in the attached files), COCO [52].

Table 3.3 Object detection and instance segmentation results on COCO with Mask R-CNN. AP^{bb} means bounding-box AP and AP^{mk} means mask AP

Architecture	Method	AP^{bb}	AP_{50}^{bb}	AP_{75}^{bb}	AP^{mk}	AP_{50}^{mk}	AP_{75}^{mk}
ResNet50	super.	40.0	59.9	43.1	34.7	56.5	36.9
ResNet50	MoCo v1[34]	40.7	60.5	44.1	35.4	57.3	37.6
FaUNAE	MoCo v2	43.1	63.0	47.2	37.7	60.2	40.6

We apply Mask R-CNN [35] with the C4 backbone as the detector, with batch normalization tuned and implemented as in [79]. All layers are fine-tuned end-to-end, and the image scale is 480x800 pixels during training and 800x800 at inference. We fine-tune all layers end-to-end. We fine-tune on the train2017 set ($\sim$ 118k images) and evaluate it on val2017. The schedule is the default $2\times$ [28].

Table 3.3 shows the results on the COCO dataset with the C4 backbones. With the $2\times$ schedule, FaUNAE is better than its ImageNet-supervised counterpart on all metrics. Due to the absent result of MoCo v2 [17], we do not compare it with our FaUNAE. We run their code for this comparison, which is even worse than v1. Also, FaUNAE is better than ResNet50 trained with unsupervised MoCo v1 [34].

3.3 Binary Neural Architecture Search

3.3.1 BNAS: Binarized Neural Architecture Search for Efficient Object Recognition

Efficient computing has become one of the hottest topics in academia and industry. It will be vital for the 5G networks to provide hardware-friendly and efficient solutions for practical and wild applications [59]. Edge computing is computing resources that are closer to the end user. This makes applications faster and more user-friendly [13]. It enables mobile or embedded devices to provide real-time intelligent analysis of big data, reducing the pressure on the cloud computing center and improving availability [33]. However, edge computing is still challenged by its limited computational ability, memory and storage, and severe performance loss, making edge computing models inefficient for feature calculation and inference [46].

A possible solution for efficient edge computing can be achieved based on compressed deep models, which fall mainly into network pruning, knowledge distillation, and model quantization. Network pruning [32] aims to remove network connections with less significance, and knowledge distillation [38] introduces a teacher-student model, which uses the soft targets generated by the teacher model to guide the student model with a much smaller model size, to achieve knowledge transfer. Differently, model quantization [42] calculates neural networks with low-bit weights and activations to compress a model more efficiently, which is also orthogonal to the other two. The binarized model is widely considered one of the most efficient ways to perform computing on embedded devices with extremely low computational cost. Binarized filters have been used in traditional convolutional neural networks (CNNs) to compress deep models [42, 57, 65], showing up to 58-time speedup and 32-time memory saving. In [65], the XNOR network is presented where the weights and inputs attached to the convolution are approximated with binary values. This efficiently implements convolutional operations by reconstructing the unbinarized filters with a single scaling factor. [92] introduces $2 \sim 4$-bit quantization based on a two-stage approach to quantizing weights and activations, significantly improving the efficiency and performance of quantized models. Furthermore, WAGE [78] is proposed to discretize both the training and inference processes and quantizes not only weights and activations but also gradients and errors. In [30], a projection convolutional neural network (PCNN) is proposed to realize binarized neural networks (BNNs) based on a simple back propagation algorithm. In our previous work [88], we propose a novel approach called Bayesian-optimized 1-bit CNNs (denoted BONNs), taking advantage of Bayesian learning to significantly improve the performance of extreme 1-bit CNNs. Other practices in [1, 21, 71] with improvements over previous work. Binarized models show the advantages of computational cost reduction and memory savings but, unfortunately, suffer from performance loss when handling wild data in practical applications. The main reasons are twofold. On the one hand, there is still a gap between low-bit weights/activations and full-precision weights/activations on feature representation, which should be investigated from new perspectives. On the other hand, traditional binarized networks are based on the neural architecture manually designed for full-precision networks, which means that the design of binarized architecture still needs to be explored.

Traditional neural architecture search (NAS) has attracted significant attention with remarkable performance in various deep learning tasks. For example, impressive results have been shown for reinforcement learning (RL)-based methods [95, 96], which train and evaluate more than 20,000 neural networks across 500 GPUs over 4 days. Recent methods like differentiable architecture search (DARTS) reduce search time by formulating the task differently [56]. DARTS relaxes the search space to be continuous so that the architecture can be optimized concerning its validation set performance by gradient descent, which provides a fast solution for an effective network architecture search. To reduce redundancy in the network space, partially connected DARTS (PC-DARTS) was recently introduced to perform a more efficient search without compromising the performance of DARTS [82].

Although DARTS or its variants have a smaller model size than traditional light models, the searched network still needs to improve its inference process due to the complicated architectures generated by multiple stacked full-precision convolution operations. Consequently, the searched network for embedded devices must still be more computationally expensive and efficient. At the same time, existing gradient-based approaches select operations without meaningful guidance. The search process is inefficient, and the selected operation might exhibit significant

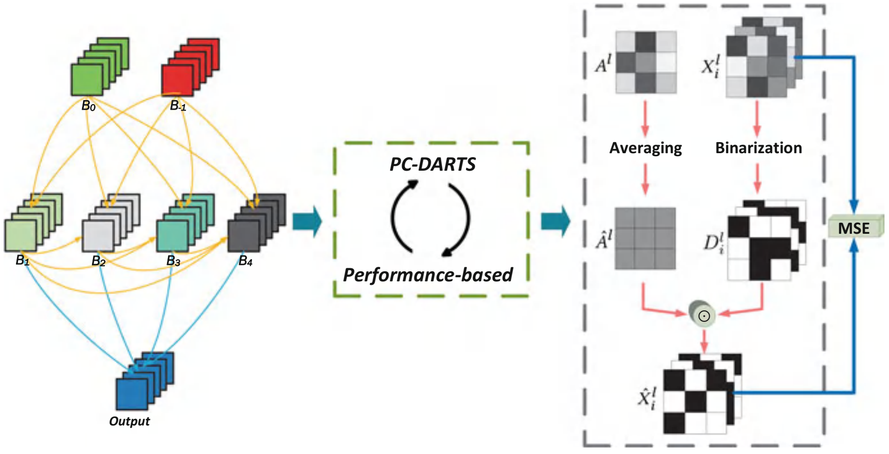


Fig. 3.11 The proposed binarized neural architecture search (BNAS) framework. In BNAS, the search cell is a fully connected directed acyclic graph with four nodes calculated based on PC-DARTS and a performance-based method. We also reformulate the optimization of binarization of CNNs in the same framework

vulnerability to model attacks based on gradient information [29, 58], also for wild data. These problems require further exploration to overcome these challenges.

To address these challenges, we transfer the NAS to a binarized neural architecture search (BNAS) [12], exploring the advantages of binarized neural networks (BNNs) on memory saving and computational cost reduction. In our BNAS framework, as shown in Fig. 3.11, we use PC-DARTS as a warm-up step, followed by the performance-based method to improve the robustness of the resulting BNNs for the wild data. Furthermore, based on observation, the early optimal operation is not necessarily optimal at the end, and the worst operation at the early stage usually performs worse at the end [90]. We take advantage of PC-DARTS and performance evaluation to reduce operating space. This means that the operations we finally reserve are certainly a near-optimal solution. On the other hand, with the operation pruning process, the search space becomes smaller and smaller, leading to an efficient search process. We show that the BNNs obtained by BNAS can outperform conventional BNN models by a large margin. It is a significant contribution to the field of BNNs considering that the performance of conventional BNNs is not yet comparable with those of their corresponding full-precision models in terms of accuracy. To further validate the performance of our method, we also implemented a 1-bit BNAS in the same framework. Unlike BNNs (only kernels are binarized), 1-bit CNNs suffer from a poor performance evaluation problem for binarized operations with binarized activations in the beginning due to insufficient training. We assume BNAS as a multi-armed bandit problem and introduce an exploration term based on the upper confidence bound (UCB) [2] to improve the search performance.

The exploration term handles the exploration-exploitation dilemma in the multi-armed bandit problem. We lead a new performance measure based on UCB by

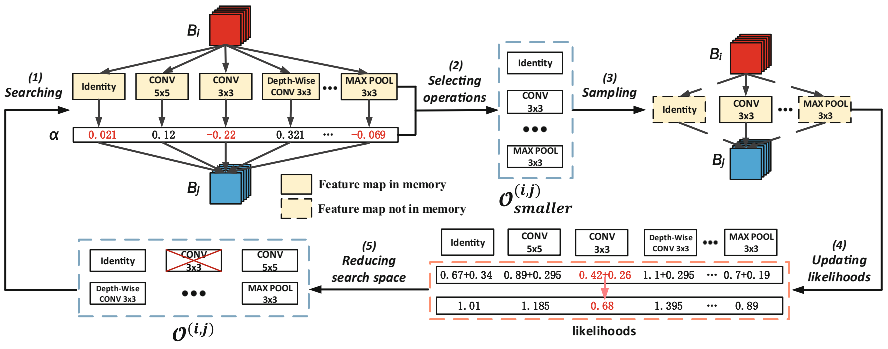


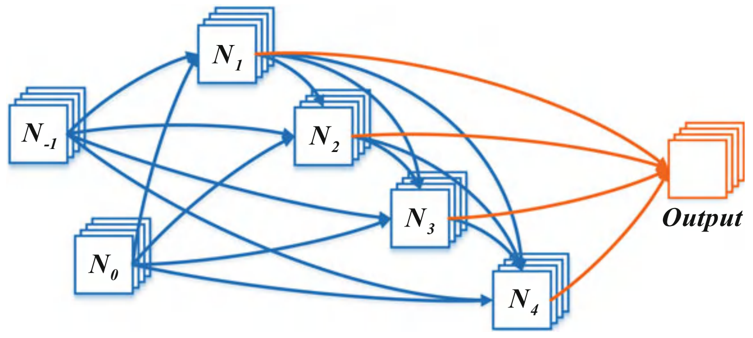
Fig. 3.12 The main steps of our BNAS: (1) Search for an architecture based on $O^{(i,j)}$ using PC-DARTS. (2) Select half of the operations with less potential from $O^{(i,j)}$ for each edge, resulting in $O^{(i,j)}_{smaller}$. (3) Select an architecture by sampling (without replacement) one operation from $O^{(i,j)}_{smaller}$ for every edge and then train the selected architecture. (4) Update the likelihood of selection of the operation $s(o_k^{(i,j)})$ based on the accuracy obtained from the selected architecture on the validation data. (5) Abandon the operation with the minimal likelihood of selection of the search space $\{O^{(i,j)}\}$ for every edge

considering both the performance evaluation and the number of trials for operation pruning in the same framework. This means the operation is ultimately abandoned only when sufficiently evaluated (Fig. 3.12).

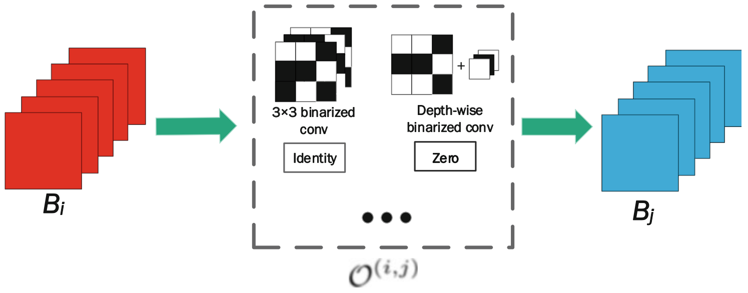
The search process for our BNAS consists of two steps. One is the potential operation ordering based on partially connected DARTS (PC-DARTS) [82], which also serves as a baseline for our BNAS. It is further improved with a second operation reduction step guided by a performance-based strategy. In the operation reduction step, we prune one operation at each iteration from one-half of the operations with less potential, as calculated by PC-DARTS. As such, the optimization of the two steps becomes faster and faster because the search space is reduced due to the operation pruning. We can take advantage of the differential framework of DARTS, where search and performance evaluation are in the same setting. We also enrich the DARTS search strategy. The gradient is used to determine which operation is better, and the proposed performance evaluation is included to reduce the search space further.

3.3.1.1 Search Space

Following [56, 95, 96], we search for a computing cell as the building block of the final architecture. A network consists of a predefined number of cells [95], which can be normal cells or reduction cells. Each cell takes the outputs of the two previous cells as input. A cell is a fully connected directed acyclic graph (DAG) of M nodes, i.e., $\{B_1, B_2, \ldots, B_M\}$, as illustrated in Fig. 3.13a. Each node B_i takes



(a) Cell



(b) Operation Set

Fig. 3.13 (**a**) A cell contains seven nodes; two input nodes B_{-1} and B_0; four intermediate nodes B_1, B_2, B_3, B_4 that apply sampled operations on the input nodes and upper nodes; and an output node that concatenates the outputs of the four intermediate nodes. (**b**) The set of operations $O^{(i,j)}$ between B_i and B_j, including binarized convolutions

its dependent nodes as input and generates an output through a sum operation $B_j = \sum_{i<j} o^{(i,j)}(B_i)$. Here each node is a specific tensor.

Unlike conventional convolutions, our BNAS is achieved by transforming all convolutions in O into binarized convolutions. We denote the full-precision and binarized kernels as X and $\hat{X}$, respectively. A convolution operation in O is represented as $B_j = B_i \otimes \hat{X}$ as shown in Fig. 3.13b, where $\otimes$ denotes convolution. To build BNAS, one critical step is how to binarize the kernels from X to $\hat{X}$, which can be implemented based on state-of-the-art BNNs, such as XNOR or PCNN. Optimizing BNNs is more challenging than conventional CNNs [30, 65], adding an additional burden to NAS. To solve it, we introduce channel sampling and reduction in operating space in differentiable NAS to significantly reduce the cost of GPU hours, leading to efficient BNAS.

3.3.1.2 Binarized Optimization for BNAS

The inference process of a BNN model is based on binarized kernels, which means that the kernels must be binarized in the forward step (corresponding to inference) during training. Contrary to the forward process, the resulting kernels are not binarized during back propagation and can be full-precision.

To achieve binarized weights, we first divide each convolutional kernel into two parts (amplitude and direction) and formulate the current binarized methods in a unified framework. We elaborate D, A, and $\hat{A}$: D_i^l are the directions of the full-precision kernels X_i^l of the lth convolutional layer, $l \in \{1, \cdots, N\}$; A^l shared by all D_i^l represents the amplitude of the lth convolutional layer; $\hat{A}^l$ and A^l are of the same size; and all elements of $\hat{A}^l$ are equal to the average of the elements of A^l. In the forward pass, $\hat{A}^l$ is used instead of the full-precision A^l. In this case, $\hat{A}^l$ can be considered a scalar. Full-precision A^l is only used for back propagation during training. Note that our formulation can represent both XNOR based on the scalar and simplified PCNN [30] whose scalar is learnable as a projection matrix.

We represent $\hat{X}$ by the amplitude and direction as

$$\hat{X} = \hat{A} \odot D, \tag{3.38}$$

where $\odot$ denotes the element-wise multiplication between matrices. We then define an amplitude loss function to reconstruct the full-precision kernels as:

$$L_{\hat{A}} = \frac{\theta}{2} \sum_{i,l} \|X_i^l - \hat{X}_i^l\|^2 = \frac{\theta}{2} \sum_{i,l} \|X_i^l - \hat{A}^l \odot D_i^l\|^2, \tag{3.39}$$

where $D_i^l = sign(X_i^l)$ represents the binarized kernel. X_i^l is the full-precision model updated during the backpropagation process in PCNNs, while $\hat{A}^l$ is calculated based on a closed-form solution in XNOR. Element-wise multiplication combines binarized kernels and amplitude matrices to approximate full-precision kernels. The final loss function is defined by considering:

$$L_S = \frac{1}{2S} \sum_{s} \|\hat{Y}_s - Y_s\|_2^2, \tag{3.40}$$

where $\hat{Y}_s$ is the label of the sth example and Y_s is the corresponding classification results. Finally, the overall loss function L is applied to supervise the training of BNAS in back propagation as:

$$L = L_S + L_{\hat{A}}. \tag{3.41}$$

Binarized optimization is used to optimize neural architecture search, leading to our binarized neural architecture search (BNAS). To this end, we use partially connected DARTS (PC-DARTS) to achieve operation potential ordering, which serves as a warm-up step for our BNAS. Denote by L_{train} and L_{val} the training and validation losses, respectively. Both losses are determined by the architecture α and the binarized weights $\hat{X}$ in the network. The goal of the warm-up step is to find $\hat{X}^*$ and α^* that minimize the validation loss $L_{val}(\hat{X}^*, \alpha^*)$, where the weights $\hat{X}^*$ associated with the architecture are obtained by minimizing the training loss $\hat{X}^* = \arg\min_{\hat{X}} \ L_{train}(\hat{X}, \alpha^*)$.

This implies a bilevel optimization problem with α as the upper-level variable and $\hat{X}$ as the lower-level variable:

$$\begin{aligned} &\arg\min_{\alpha} L_{val}(\hat{X}^*, \alpha) \\ &\qquad s.t.\ \hat{X}^* = \arg\min_{\hat{X}} \ L_{train}(\hat{X}, \alpha). \end{aligned} \tag{3.42}$$

To better understand our method, we also review the core idea of PC-DARTS, which can take advantage of partial channel connections to improve memory efficiency. For example, the connection from B_i to B_j involves defining a channel sampling mask $S^{(i,j)}$, which assigns 1 to selected channels and 0 to masked ones. The selected channels are sent to a mixed computation of $|O^{(i,j)}|$ operations, while the masked ones bypass these operations. They are copied directly to the output, which is formulated as:

$$\begin{aligned} &f^{(i,j)}(B_i, S^{(i,j)}) \\ &= \sum_{o_k^{i,j} \in O^{(i,j)}} \frac{exp\{\alpha_{o_k^{(i,j)}}\}}{\sum_{o_{k'}^{(i,j)} \in O^{(i,j)}} exp\{\alpha_{o_{k'}^{(i,j)}}\}} \cdot o_k^{(i,j)}(S^{(i,j)} * B_i) \\ &+ (1 - S^{(i,j)}) * B_i, \end{aligned} \tag{3.43}$$

where $S^{(i,j)} * B_i$ and $(1 - S^{(i,j)}) * B_i$ denote the selected and masked channels, respectively, and $\alpha_{o_k^{(i,j)}}$ is the parameter of operation $o_k^{(i,j)}$ between B_i and B_j.

PC-DARTS sets the proportion of selected channels to $1/C$ by considering C as a hyperparameter. In this case, the computational cost can be reduced by C. However, the size of the entire search space is $2 \times K^{|\mathcal{E}_{\mathcal{M}}|}$, where $\mathcal{E}_{\mathcal{M}}$ is the set of possible edges with M intermediate nodes in the fully connected DAG, and the "2" comes from the two types of cells. In our case with $M = 4$, together with the two input nodes, the total number of cell structures in the search space is $2 \times 8^{2+3+4+5} = 2 \times 8^{14}$. This is a vast space to search for binarized neural architectures, which need more time than a full-precision NAS. Therefore, efficient optimization strategies are required for BNAS.

3.3.1.3 Performance-Based Strategy for BNAS

Reinforcement learning could be more efficient in architecture search due to delayed rewards in network training. That is, the evaluation of a structure is usually done after the network training converges. On the other hand, we can evaluate a cell when training the network. Inspired by [85], we use a performance-based strategy to increase search efficiency by a large margin. [85] did a series of experiments showing that in the early stage of training, the validation accuracy ranking of different network architectures is not a reliable indicator of the quality of the final architecture. However, we observe that the results of the experiments suggest that if an architecture performs poorly at the beginning of training, there is little hope that it can be part of the final optimal model. As training progresses, this observation shows less uncertainty. Based on this observation, we derive a simple yet effective operation-abandoning process. We progressively abandon the worst-performing operation on each edge during training and increasing epochs.

To this end, we reduce the search space $\{\mathcal{O}^{(i,j)}\}$ after the warm-up step achieved by PC-DARTS to increase search efficiency. According to $\{\alpha_{o_k^{(i,j)}}\}$, we can select half of the operations with less potential than $\mathcal{O}^{(i,j)}$ for each edge, resulting in $\mathcal{O}^{(i,j)}_{smaller}$. After that, we randomly sample one operation from the $K/2$ operations in $\mathcal{O}^{(i,j)}_{smaller}$ for every edge, then obtain the validation accuracy by training the sampled network for one epoch, and finally assign this accuracy to all the sampled operations. These three steps are performed $K/2$ times by sampling without replacement, giving each operation exactly one accuracy for every edge.

We repeat it T times. Thus, each operation for every edge has T accuracies $\{y^{(i,j)}_{k,1}, y^{(i,j)}_{k,2}, \ldots, y^{(i,j)}_{k,T}\}$. Then we define the selection likelihood of the kth operation in $\mathcal{O}^{(i,j)}_{smaller}$ for each edge as:

$$s_{smaller}(o_k^{(i,j)}) = \frac{exp\{\bar{y}_k^{(i,j)}\}}{\sum_m exp\{\bar{y}_m^{(i,j)}\}}, \tag{3.44}$$

where $\bar{y}_k^{(i,j)} = \frac{1}{T}\sum_t y^{(i,j)}_{k,t}$. And the selection likelihoods of the other operations not in $\mathcal{O}^{(i,j)}_{smaller}$ are defined as:

$$s_{larger}(o_k^{(i,j)}) = \frac{1}{2}(\max_{o_k^{(i,j)}}\{s_{smaller}(o_k^{(i,j)})\} + \frac{1}{\lceil K/2 \rceil}\sum_{o_k^{(i,j)}} s_{smaller}(o_k^{(i,j)})), \tag{3.45}$$

where $\lceil K/2 \rceil$ denotes the smallest integer $\geq K/2$. It is used because K can be an odd integer during iteration in the proposed Algorithm 6. Equation 3.45 is an estimate for the remaining operations using a value balanced between the maximum and the average of $s_{smaller}(o_k^{(i,j)})$. Then, $s(o_k^{(i,j)})$ is updated by:

$$s(o_k^{(i,j)}) \leftarrow \frac{1}{2} s(o_k^{(i,j)}) + q_k^{(i,j)} s_{smaller}(o_k^{(i,j)}) + \quad (1 - q_k^{(i,j)}) s_{larger}(o_k^{(i,j)}), \tag{3.46}$$

where $q_k^{(i,j)}$ is a mask, which is 1 for the operations in $O_{smaller}^{(i,j)}$ and 0 for the others.

When searching for BNAS, we do not use PC-DARTS as a warm-up to consider efficiency because quantizing feature maps is slower. Therefore, $O_{smaller}^{(i,j)}$ is $O^{(i,j)}$. Also, we introduce an exploration term into Eq. 3.46 based on bandit [2]. In machine learning, the multi-armed bandit problem is a classic reinforcement learning problem that exemplifies the exploration-exploitation trade-off dilemma: shall we stick to an arm that has given high reward so far (exploitation) or rather probe other arms further (exploration)? The upper confidence bound (UCB) is widely used for dealing with the exploration-exploitation dilemma in the multi-armed bandit problem. Then, with the above analysis, Eq. 3.46 becomes:

$$s(o_k^{(i,j)}) \leftarrow s(o_k^{(i,j)}) + \delta * \sqrt{\frac{2 \log N}{n_{k,t}^{(i,j)}}} \tag{3.47}$$

where N is the total number of samples, $n_{k,t}^{(i,j)}$ refers to the number of times the kth operation of the edge (i, j) has been selected, and t is the epoch index. The first item in Eq. 3.47 is the value term, which favors historically good operations. The second is the exploration term, which allows operations to get an exploration bonus that grows with $\log N$. And this work uses $\delta = 2$ to balance the value and exploration terms. We also test other values, which achieve slightly worse results. Thus, 1-bit convolutions, which misbehave in sufficient trials, are prone to be abandoned.

Finally, we abandon the operation with a minimal likelihood of selection for each edge. The size of the search space is significantly reduced from $2 \times |O^{(i,j)}|^{14}$ to $2 \times (|O^{(i,j)}| - 1)^{14}$. We have the following:

$$O^{(i,j)} \leftarrow O^{(i,j)} - \{\arg\min_{o_k^{(i,j)}} s(o_k^{(i,j)})\}. \tag{3.48}$$

The optimal structure is obtained when only one operation is left on each edge. Our performance-based search algorithm is presented in Algorithm 6. Note that in line 1, PC-DARTS is performed for L epochs as a warm-up to find an initial architecture, and line 14 is used to update the architecture parameters $\alpha_{o_k^{(i,j)}}$ for all edges due to reduction of the search space $\{O^{(i,j)}\}$.

Algorithm 6: Performance-based search

```
Input: Training data, Validation data, Searching hyper-graph: $\mathcal{G}$, $K = 8$, $T = 3$, $V = 1$,
$L = 5$, $s(o_k^{(i,j)}) = 0$ for all edges
Output: Optimal structure $\alpha$
 1: Search an architecture for $L$ epochs based on $O^{(i,j)}$ using PC-DARTS
 2: while ($K > 1$) do
 3:     Select $O_{smaller}^{(i,j)}$ consisting of $\lceil K/2 \rceil$ operations with smallest $\alpha_{o_k^{(i,j)}}$ from $O^{(i,j)}$ for
        every edge;
 4:     for $t = 1, \ldots, T$ epoch do
 5:         $O'^{(i,j)}_{smaller} \leftarrow O^{(i,j)}_{smaller}$;
 6:         for $e = 1, \ldots, \lceil K/2 \rceil$ epoch do
 7:             Select an architecture by sampling (without replacement) one operation from
                $O'^{(i,j)}_{smaller}$ for every edge
 8:             Train the selected architecture and get the accuracy on the validation data
 9:             Assign this accuracy to all the sampled operations;
10:         end for
11:     end for
12:     Update $s(o_k^{(i,j)})$ using Eq. 3.46;
13:     if 1 bit then
14:         Update $s(o_k^{(i,j)})$ using Eq. 3.47;
15:     end if
16:     Update the search space $\{O^{(i,j)}\}$ using Eq. 3.48;
17:     Search the architecture for $V$ epochs based on $O^{(i,j)}$ using PC-DARTS;
18:     $K = K - 1$;
19: end while
```

3.3.1.4 Gradient Update for BNAS

In BNAS, $\hat{X}^l$ in the lth layer is used to calculate the output feature maps F^{l+1} as:

$$F^{l+1} = ACconv(F^l, \hat{X}^l), \tag{3.49}$$

where $ACconv$ denotes the amplitude convolution operation designed in Eq. 3.50. In ACconv, the output feature map channels are generated as follows:

$$F_h^{l+1} = \sum_{i,g} F_g^l \otimes \hat{X}_i^l, \tag{3.50}$$

where $\otimes$ denotes the convolution operation; F_h^{l+1} is the hth feature map in the $(l+1)$th convolutional layer; and F_g^l denotes the gth feature map in the lth convolutional layer. Note that the BNAS kernels are binarized, whereas for 1-bit BNAS, both the kernels and the activations are binarized. Similar to previous work [30, 57, 65], the 1-bit BNAS is obtained by binarizing the kernels and activations simultaneously. In addition, we replace ReLU with PReLU to reserve harmful elements generated by a 1-bit convolution.

In BNAS, full-precision kernels X_i and amplitude matrices A need to be learned and updated. The kernels and the matrices are jointly learned. BNAS updates the full-precision kernels and amplitude matrices in each convolutional layer. In what follows, the layer index l is omitted for simplicity.

We denote δ_{X_i} as the gradient of the full-precision kernel X_i, and we have:

$$\delta_{X_i} = \frac{\partial L}{\partial X_i} = \frac{\partial L_S}{\partial X_i} + \frac{\partial L_{\hat{A}}}{\partial X_i}, \tag{3.51}$$

$$X_i \leftarrow X_i - \eta_1 \delta_{X_i}, \tag{3.52}$$

where η_1 is a learning rate. Then we have:

$$\frac{\partial L_S}{\partial X_i} = \frac{\partial L_S}{\partial \hat{X}_i} \cdot \frac{\partial \hat{X}_i}{\partial X_i} = \frac{\partial L_S}{\partial \hat{X}_i} \cdot \hat{A} \cdot \mathbb{1}, \tag{3.53}$$

$$\frac{\partial L_{\hat{A}}}{\partial X_i} = \theta \cdot (X_i - \hat{A} \odot D_i), \tag{3.54}$$

where X_i is the full-precision convolutional kernel corresponding to D_i and $\mathbb{1}$ is the indicator function [65] widely used to estimate the gradient of the nondifferentiable function.

After updating X, we update the amplitude matrix A. Let δ_A be the gradient of A. According to Eq. 3.41, we have:

$$\delta_A = \frac{\partial L}{\partial A} = \frac{\partial L_S}{\partial A} + \frac{\partial L_{\hat{A}}}{\partial A}, \tag{3.55}$$

$$A \leftarrow |A - \eta_2 \delta_A|, \tag{3.56}$$

where η_2 is another learning rate. Note that the amplitudes are always set to be nonnegative. Then we have:

$$\frac{\partial L_S}{\partial A} = \sum_i \frac{\partial L_S}{\partial \hat{X}_i} \cdot \frac{\partial \hat{X}_i}{\partial \hat{A}} \cdot \frac{\partial \hat{A}}{\partial A} = \sum_i \frac{\partial L_S}{\partial \hat{X}_i} \cdot D_i, \tag{3.57}$$

$$\frac{\partial L_{\hat{A}}}{\partial A} = \frac{\partial L_{\hat{A}}}{\partial \hat{A}} \cdot \frac{\partial \hat{A}}{\partial A} = -\theta \cdot (X_i - \hat{A} \odot D_i) \cdot D_i, \tag{3.58}$$

where $\frac{\partial \hat{A}}{\partial A}$ is set to 1 for an easy implementation of the algorithm. Note that $\hat{A}$ and A are used in the forward-pass and back propagation asynchronously. The above derivations show that BNAS is learnable with the new BP algorithm.

3.3.1.5 Ablation Study

We use the same datasets and evaluation metrics as the existing NAS works [8, 55, 56, 96]. First, most experiments are conducted on CIFAR-10 [44], and the color intensities of all images are normalized to $[-1, +1]$. During the architecture search, the 50K training samples of CIFAR-10 are divided into two subsets of equal size, one to train the network weights and the other to search the architecture hyperparameters. When reducing the search space, we randomly select 5K images from the training set as a validation set (used on line 8 of Algorithm 6). Especially for 1-bit BNAS, we replace ReLU with PReLU to avoid the disappearance of negative numbers generated by a 1-bit convolution. The bandit strategy is introduced to solve the insufficient training problem caused by the binarization of both kernels and activations. To further show the efficiency of our method, we also search for the architecture on ImageNet directly.

In the search process, we consider a total of six cells in the network, where the reduction cell is inserted in the second and fourth layers, and the others are normal cells. There are $M = 4$ intermediate nodes in each cell. Our experiments follow PC-DARTS. We set the hyperparameter C in PC-DARTS to 2 for CIFAR-10 so that only $1/2$ features are sampled for each edge. The batch size is set to 128 during the search for an architecture for $L = 5$ epochs based on $O^{(i,j)}$ (line 1 of Algorithm 6). Note that for $5 \leq L \leq 10$, a larger L has little effect on the final performance but costs more search time, as shown in Table 3.4. We freeze network hyperparameters, such as α, and allow only network parameters, such as filter weights, to be tuned in the first $three$ epochs. Then in the next two epochs, we train both the network hyperparameters and the network parameters. This is done to provide initialization for the network parameters, thus alleviating the drawback of parameterized operations compared to free-parameter operations. We also set $T = 3$ (line 4 in Algorithm 6) and $V = 1$ (line 14), so the network is trained in fewer than 60 epochs, with a larger batch size of 400 (due to few operation samplings) during the reduction of the search space. The initial number of channels is 16. We use momentum-based SGD to optimize network weights, with an initial learning rate of 0.025 (annealed to zero following a cosine schedule), a momentum of 0.9, and a weight decay of 5×10^{-4}. The learning rate to find the hyperparameters is set to 0.01. When we search for the architecture directly on ImageNet, we use the same parameters as when searching on CIFAR-10, except that the initial learning rate is set to 0.05

Table 3.4 With different L, the accuracy and search cost of BNAS based on PCNN on the CIFAR10 dataset

	L				
Model	3	5	7	9	11
Accuracy (%)	95.8	96.06	95.94	96.01	96.03
Search cost	0.0664	0.09375	0.1109	0.1321	0.1687

3.3.2 *BDetNAS: A Fast Binarized Detection Neural Architecture Search*

3.3.2.1 Search Space

We follow the same settings as previous NAS works [56, 91] to search for a computation cell as the building block of the final architecture. As plotted in Fig. 3.14, the search space is related to three main elements: node, cell, and network. We will describe the binarized architecture search space and the method to build the binarized network as below.

Node As the fundamental elements that compose cells, each node F_i is a set of specific feature maps. To formulate a directed acyclic graph (DAG), each node has its connections. Possible operation set between nodes (i, j) is denoted as $\mathcal{O}_{i,j}$, where a practicable operation is selected to transform F_i to F_j as shown in Fig. 3.14c.

In a cell, nodes can be divided into three categories: input node, intermediate node, and output node. Each cell takes the output of previous two cells as the input node. And each intermediate node takes the input node and previous intermediate nodes as the input. Then we concatenate all intermediate nodes to formulate the final output node. Following this guideline as [56], we form a possible operation set, denoted as $\mathcal{O}$, consisting of $|\mathcal{O}| = 8$ operations: (1) 3×3 max pooling, (2) no

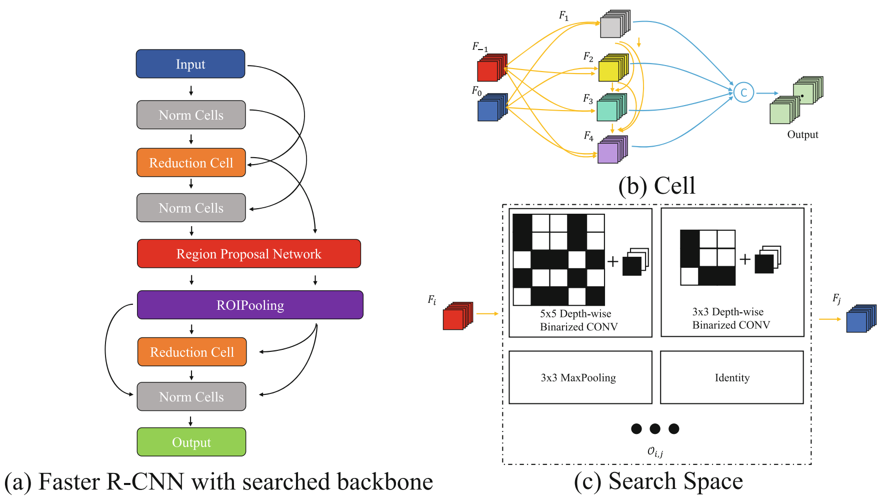


Fig. 3.14 (**a**) The Faster R-CNN detector with searched network consisting of stacked cell. (**b**) A cell contains seven nodes; two input nodes F_{-1}, F_0; four intermediate nodes F_1, F_2, F_3, F_4 that apply sampled operations on the input nodes and upper nodes; and an output node that concatenates the outputs of the four intermediate nodes. © denotes the concatenating operation. (**c**) The search space of BDetNAS, link operation between input, and intermediate nodes will be selected among the possible operations $\mathcal{O}_{i,j}$

connection (zero), (3) 3×3 average pooling, (4) skip connection (identity), (5) 3×3 dilated convolution with rate 2, (6) 5×5 dilated convolution with rate 2, (7) 3×3 depth-wise separable convolution, and (8) 5×5 depth-wise separable convolution. Moreover, a binarized NAS is achieved by transforming all the convolutions in O to binarized convolutions as shown in Fig. 3.14c.

Cell A cell is defined as a tiny convolutional network with complex connections and multiple operation layers. Cells can be categorized into two classes, i.e., normal cell and reduction cell. We define the input shape of cells as $K \times W \times C$. A normal cell uses the operations with stride 1, so its input and output shape are identical, i.e., $K \times W \times C$. Following the guideline of common heuristic in most human designed convolutional neural networks [36, 41, 70], C is doubled when the stride is 2. Hence, a reduction cell uses the operations with the stride set to 2, and the output shape is $K/2 \times W/2 \times 2C$.

We set the cell according to [56], which is formed by $seven$ nodes and correspondingly $2+3+4+5 = 14$ possible connections as illustrated in Fig. 3.14b. The edge between two nodes denotes a possible operation which will be selected according to the performance-based strategy. In training, we form an architecture every epoch by sampling operations without replacement. And we optimize the search space according to our search space reduction algorithm. In addition, we should cut the 14 possible connections down to 8. Thus, we select the top $eight$ probabilities to generate the final cells in testing. Therefore, the size of the whole search space is $2 \times 8^{2+3+4+5} = 2 \times 8^{14}$, which is an extremely large space to search. Hence, efficient optimization methods are required.

Network A backbone network consists of a predefined number of stacked cells, which take the output of two previous cells as the input. Among the cells, there are either normal cells with stride set as 1 or reduction cells with the stride set as 2. Following [56], we employ two stem cells with the total stride set as 8 to preprocess the raw image. Hence, only two kinds of cells are generated. Based on the performance ranking hypothesis [91], we train a small stacked network with six cells (two reduction cells and $four$ normal cells) for search. And then we employ the corresponding 20-cell network of the optimal 6-cell network for pre-training and fine-tuning. A Faster R-CNN detector [68] with the searched backbone is plotted as shown in Fig. 3.14a.

3.3.2.2 Performance-Based Strategy for BDetNAS

The core idea of our search algorithm is to sample randomly and reduce the search space step by step according to the testing accuracy. In general, we select an edge between specific nodes (i, j) from operation sets and test the network compose of the selected edges without replacement. We record the performance information according to the test accuracy and accordingly optimize the search space.

To accomplish this, we implement the operation sampling on $\mathcal{O}_{i,j}$. We randomly sample an edge $o_{i,j}^k$ from $\mathcal{O}_{i,j}$ to form a network for test. After that, we update the performance of each operation between nodes i and j as:

$$s(o_{i,j}^k) = \frac{\sum_{e=1}^{N} y^e \cdot m_{i,j}^{k,e}}{\sum_{e=1}^{N} m_{i,j}^{k,e}}, \tag{3.59}$$

where N denotes the current training epoch and $1 \le e \le N$. y^e denotes the test accuracy of the e-th epoch. And $m_{i,j}^{k,e}$ denotes an indicative variable defined as:

$$m_{i,j}^{k,e} = \begin{cases} 1, & o_{i,j}^k \text{ is selected in } e\text{−th epoch} \\ 0, & \text{else} \end{cases} \tag{3.60}$$

Equation 3.59 indicates taking the average test accuracy of the epochs, where $o_{i,j}^k$ is selected, as the performance of $s(o_{i,j}^k)$.

We define the iteration times $\mathcal{T} = 3$ for sampling without replacement. We first repeat sampling $|\mathcal{O}_{i,j}| \times \mathcal{T} = 8 \times 3 = 24$ epochs and then reduce the search space as:

$$\mathcal{O}_{i,j} \leftarrow \mathcal{O}_{i,j} - \arg\min_{o_{i,j}^k} s(o_{i,j}^k). \tag{3.61}$$

After Eq. 3.61, the search space size of every edge is reduced to $|\mathcal{O}_{i,j}| - 1$ for one cell. As a result, the whole search space size is significantly reduced from $2 \times (|\mathcal{O}_{i,j}|)^L$ to $2 \times (|\mathcal{O}_{i,j}| - 1)^L$, where L is the number of cells. Then we repeat the search space reduction process until $|\mathcal{O}_{i,j}| = 1$ to achieve the final architecture. The number of total epochs is $(8 + 7 + \cdots + 2) * 3 = 108$, which is efficient.

3.3.2.3 Optimization for BDetNAS

To achieve a binarized NAS, kernel weights are binarized by decomposing the full-precision kernel X into amplitude and direction as:

$$\hat{X} = A \cdot D, \tag{3.62}$$

where A and D respectively denote the amplitude and the direction of X. D is the ℓ_1-normalized matrix, which is element-wisely calculated by sign(X) as $\frac{-1}{|X|}$ for negative X and $\frac{1}{|X|}$ for positive X. $|X|$ denotes the number of elements in X. A is a scalar. Then a binarized convolution can be formulated as:

$$F_j = F_i \circ \hat{X}_{i,j}, \tag{3.63}$$

where $\circ$ denotes convolution.

Algorithm 7: BDetNAS framework

```
Input: Training data, validation data, $O_{i,j}$ and $s(o_{i,j}^k) = 0$ for all edges, supernet and $e = 0$;
Output: Optimal backbone architecture $a^*$, optimal $w_D^*$ for object detection;
 1: Initialize $w$ randomly;
 2: repeat
 3:     Sample $o_{i,j}^k$ randomly with no replacement.
 4:     for $t = 1$ to $T$ do
 5:         Train the selected architecture according to Eq. 3.65.
 6:     end for
 7:     Test the trained network and calculate the test accuracy $y^e$.
 8:     Update the $s(o_{i,j}^k)$ via Eq. 3.59.
 9:     if $e == |O_{i,j}| \times \mathcal{T}$ then
10:         Update $O_{i,j}$ via Eq. 3.61.
11:     end if
12:     $e \leftarrow e + 1$
13: until $|O_{i,j}| = 1$
14: Pre-train searched $a^*$ and get $w_P^*$ on ImageNet.
15: Initialize $w \leftarrow w_P^*$ and fine-tune on VOC/COCO.
16: Get $a^*$ and $w_D^*$.
```

We then define an amplitude loss function to reconstruct the full-precision kernels as:

$$\begin{aligned}\mathcal{L}_A &= \sum_{l}^{L} \sum_{F_i, F_j \in C_l} \|X_{i,j} - \hat{X}_{i,j}\|_2^2 \\ &= \sum_{l}^{L} \sum_{F_i, F_j \in C_l} \|X_{i,j} - A_{i,j} \cdot D_{i,j}\|_2^2,\end{aligned} \tag{3.64}$$

where C_l denotes the l-th cell. $X_{i,j}$ denotes the full-precision kernel and $\hat{X}_{i,j}$ denotes a reconstructed one. The total loss for optimization in search process is:

$$\mathcal{L} = \mathcal{L}_{Cls} + \frac{\alpha}{2}\mathcal{L}_A, \tag{3.65}$$

where $\mathcal{L}_{Cls}$ is the conventional loss function, e.g., cross entropy. α is a hyperparameter.

3.3.2.4 Experiments

In this section, we compare our BDetNAS with state-of-the-art manually designed and other NAS object detectors. Moreover, we also compare the BNNs obtained by our BDetNAS based on XNOR [65] to validate effectiveness of our method. More experimental results are also provided in the supplementary material. GPU days are counted according to NVIDIA GTX 1080Ti, which is the same as DetNAS [18]. All the experiments and models are implemented with PyTorch.

Experimental Settings

Search on ImageNet+VOC/COCO For search process, we use the commonly used 1.28M ImageNet `ILSVRC2012` [45] and Cropped&Resized detection dataset for training images, as plotted in Fig. 5.5. The Cropped&Resized VOC `trainval07+12` [23] has 46.9k images over 20 classes. Likewise, the Cropped&Resized COCO `trainval35k` [53] has 0.86M images over 80 classes. Hence, we get an augmented dataset of 1.33M images for search on VOC `trainval07+12` and of 2.14M images for search on COCO `trainval35k`. When calculating the accuracy, we randomly select 5K images from the training set as a validation set (in line 7 of Algorithm 7). As illustrated in Sect. 3.3.2.2, 108 epochs are needed for search. And we use a batch size of 512 on 4 NVIDIA GTX 1080Ti GPUs for 280k iterations for ImageNet `ILSVRC2012` + VOC `trainval07+12` and 450k iterations on ImageNet `ILSVRC2012` + COCO `trainval35k`.

Pre-training on ImageNet For ImageNet classification dataset, we use the commonly used 1.28M ImageNet `ILSVRC2012` [45]. To get a pre-trained backbone on ImageNet, the network is trained from scratch for 250 epochs with a batch size of 512. We use the SGD optimizer with a momentum of 0.9, an initial learning rate of 0.05 (decayed down to zero following a cosine schedule), and a weight decay of 3×10^{-5}. Additional enhancements are adopted including label smoothing and an auxiliary loss tower during training.

Fine-tuning on VOC/COCO We validate our method with Faster R-CNN [68] detector. The training images are randomly flipped for augmentation. Then a superposition of the original data and the augmented data is used for training. 40k input images are employed for VOC `trainval07+12` and 230k input images are employed for COCO `trainval35k`. Note that COCO `trainval35k` used here is the left part with 5k COCO `minival` taken away. We train on 4 GPUs with a total of 4 images per mini-batch for 27k iterations on VOC and 150k iterations on COCO. The weights of backbone are initialized with ImageNet pre-training. The parameters of region proposal network (RPN) are randomly initialized. We set the weight decay as 1×10^{-4} and momentum as 0.9. Initial learning rate is 4×10^{-3} and we decay the rate at the 8th epoch of the total ten epochs.

Results on VOC `test2007`

Hyperparameter α is set as 2×10^{-5} for search on VOC `trainval07+12`. Relevant ablation study is attached in supplementary material. We compare our method with manually designed networks with a similar model size, state-of-the-art quantization methods, and networks searched by NAS. The manually designed backbones include ResNet [36] and VGG [70]. Binarized ResNet-34 implemented

Table 3.5 Comparison with the state-of-the-art object detectors on VOC test2007

Detector	Backbone	mAP	Backbone Params (M)	Search Cost (GPU days)
Faster R-CNN [68]	ResNet-18 [36]	73.2	10.67 (32 bits)	–
Faster R-CNN [68]	ResNet-34[36]	75.6	20.27 (32 bits)	–
Faster R-CNN [68]	VGG-16 [70]	73.5	15.21 (32 bits)	–
Faster R-CNN [68]	ResNet-34 [75]	59.0	20.27 (1 bit)	–
Faster R-CNN [68]	ResNet-34 [65]	54.7	20.27 (1 bit)	–
FPN [50]	DetNAS [18]	81.5	4.34 (32 bits)	35
RetinaNet [51]	DetNAS [18]	80.1	5.07 (32 bits)	35
Faster R-CNN [68]	FairNAS [19]	67.3	6.72 (1 bit)	8.1
Faster R-CNN [68]	BDetNAS (XNOR[65])	68.8	6.23 (1 bit)	**8.3**
Faster R-CNN[68]	BDetNAS	**70.8**	**6.51** (1 bit)	**8.1**

by TBN [75] and XNOR [65] are considered in our comparison. In addition, we compare our BDetNAS with state-of-the-art DetNAS [18].

As illustrated in Table 3.5, Faster R-CNN [68] with full-precision ResNet-18, VGG-16 and ResNet-34 achieves 73.2, 73.5, and75.6 mAP on VOC test2007, respectively, while BDetNAS incurs only 1.4%, 1.7%, and 4.8% mAP loss with a compressed model size by 52×, 74×, and 99×. For binarized ResNet-34 implemented via XNOR [65] and TBN [75], our BDetNAS achieve 16.1% and 11.8% mAP higher as well as compress the memory usage by 3.2×.

Compared with the full-precision detectors obtained by DetNAS [18], the binarized networks with our BDetNAS have acceptable mAP loss but with much more compressed models. Note that the numbers of parameters of backbones searched by DetNAS [18] are less than 5M. However, the binarized networks only need 1 bit to save one parameter, while the full-precision networks need 32 bits. Hence, our BDetNAS saves about 21× and 25× memory, which is an obviously superior trade-off for real applications. In terms of search efficiency, our framework searches directly on image classification task from scratch and needs no advanced pre-training or fine-tuning compared to DetNAS. Hence, our BDetNAS is more than 4× faster compared with DetNAS. The superiority is attributed to the proposed scheme of search space reduction and novel search framework.

In addition, we reimplement FairNAS [19], i.e., random search under the same setup as ours for fair comparison. As illustrated in the last rows of Table 3.5, BDetNAS outperforms FairNAS [19] with 3.5% mAP higher after searching for same epochs. This demonstrates that our BDetNAS can effectively improve the performance of the backbone. Compared to BDetNAS implemented by XNOR [65], the BDetNAS with our novel quantization method achieves higher mAP with similar memory usage. This demonstrates our novel quantization framework is of great effect (Fig. 3.15).

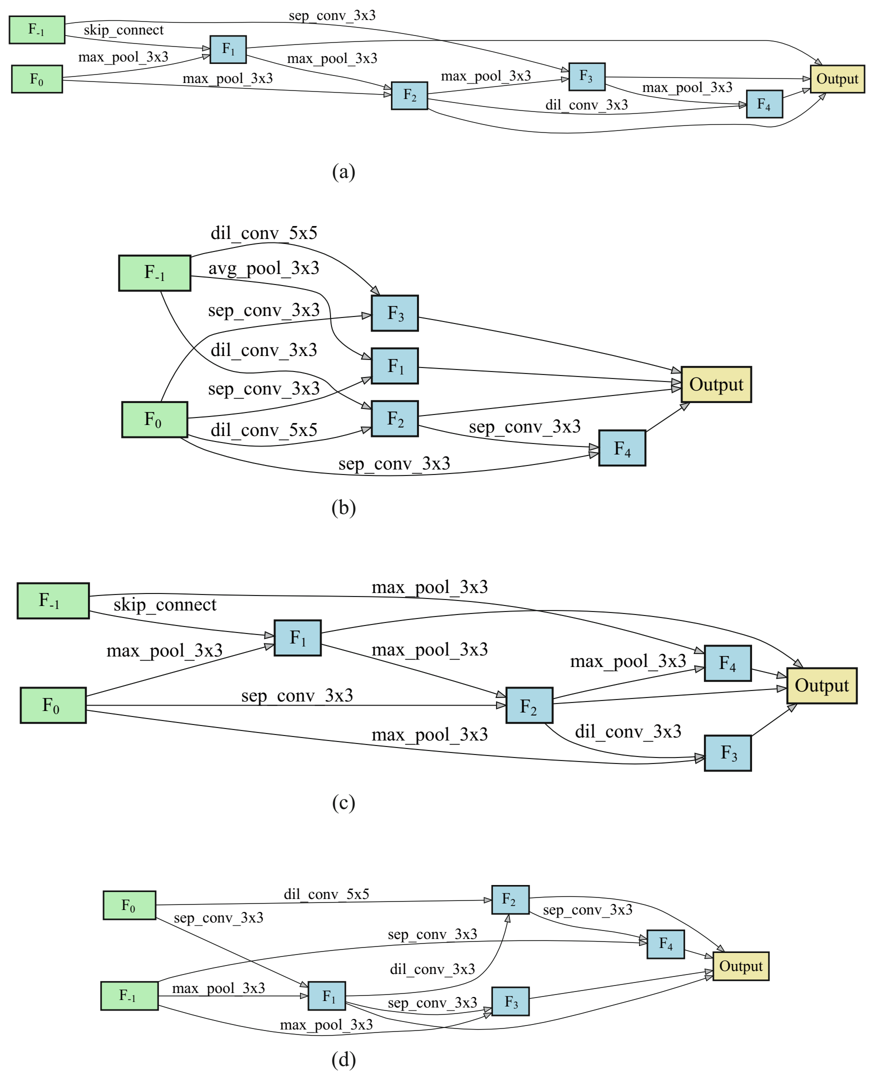


Fig. 3.15 Detailed structures of the best cells discovered using BDetNAS based on our quantization methods. In the normal cell, the stride of the operations on *two* input nodes is 1, and in the reduction cell, the stride is 2. (**a**) Normal cell on VOC. (**b**) Reduction cell on VOC. (**c**) Normal cell on COCO. (**d**) Reduction cell on COCO

Results on COCO `minival`

We further compare BDetNAS with other state of the arts on COCO `minival`. Hyperparameter α is set as 1×10^{-5} for search on COCO `trainval35k`. Relevant ablation study is attached in supplementary material. The backbones for comparison

Table 3.6 Comparison with the state-of-the-art object detectors on COCO `minival`

Detector	Backbone	mAP			Backbone params	Search cost
		AP	$AP^{0.5}$	$AP^{0.75}$	(M)	(GPU days)
Faster R-CNN [68]	ResNet-18 [36]	32.2	53.8	34.0	10.67 (32 bits)	–
Faster R-CNN [68]	MobileNetV2 [69]	29.0	49.7	29.5	3.4(32 bits)	–
Faster R-CNN [68]	ResNet-18 [48]	28.1	48.4	29.3	10.67 (1 bit)	–
Faster R-CNN [68]	MobleNetV2 [48]	25.5	45.3	25.7	3.4 (1 bit)	–
RetinaNet [51]	DetNAS [18]	34.1	–	–	5.07 (32 bits)	44
Faster R-CNN [68]	BDetNAS	**29.0**	**49.2**	**29.7**	**6.30** (1 bit)	**13.4**

consist of manually designed full-precision ones such as MobileNetV2 [69] and ResNet-18 [36], binarized one such as FQN [48], and searched one by DetNAS [18]. From the results in Table 3.6, we have the following observations: (1) BDetNAS performs equally to human-designed light full-precision networks MobileNetV2 (29.0 vs. 29.0) as well as save memory usage by 17× on the same detector. (2) Compared with binarized ResNet-18 by FQN [48], BDetNAS achieves 0.4% mAP higher as well as compress the model by 1.7×. And BDetNAS achieves 3.5% mAP higher compared with binarized MobileNetV2 by FQN[48]. (3) BDetNAS saves memory usage by 26× with only 5.1%mAP lower (29.0 vs. 34.1) compared with DetNAS [18] on RetinaNet [51]. Moreover, our search cost is only 30.4% of DetNAS.

References

1. Milad Alizadeh, Javier Fernández-Marqués, Nicholas D Lane, and Yarin Gal. An empirical study of binary neural networks' optimisation. In *Proceedings of the International Conference on Learning Representations*, 2018.
2. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. In *Machine learning*, 2002.
3. Philip Bachman, R Devon Hjelm, and William Buchwalter. Learning representations by maximizing mutual information across views. In *NeurIPS*, 2019.
4. Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.
5. A. Buades, B. Coll, and J. Morel. A non-local algorithm for image denoising. In *CVPR*, 2005.
6. Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. Bats: Binary architecture search. In *Proc. of ECCV*, pages 309–325, 2020.
7. Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, 2018.
8. Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *International Conference on Machine Learning*, pages 678–687. PMLR, 2018.

9. Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019.
10. Hanlin Chen, Baochang Zhang, Shenjun Xue, Xuan Gong, Hong Liu, Rongrong Ji, and David S. Doermann. Anti-bandit neural architecture search for model defense. *ArXiv*, abs/2008.00698, 2020.
11. Hanlin Chen, Li'an Zhuo, Baochang Zhang, Xiawu Zheng, Jianzhuang Liu, Rongrong Ji, David Doermann, and Guodong Guo. Binarized neural architecture search for efficient object recognition. *International Journal of Computer Vision*, 129(2):501–516, 2021.
12. Hanlin Chen, Li'an Zhuo, Baochang Zhang, Xiawu Zheng, Jianzhuang Liu, Rongrong Ji, David S. Doermann, and Guodong Guo. Binarized neural architecture search for efficient object recognition. *International Journal of Computer Vision*, 129:501–516, 2020.
13. Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. In *Proceedings of the IEEE*, 2019.
14. Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. *arXiv:2002.05709*, 2020.
15. Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1294–1303, 2019.
16. Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proc. of ICCV*, 2019.
17. Xinlei Chen, Haoqi Fan, Ross Girshick, and Kaiming He. Improved baselines with momentum contrastive learning. *arXiv:2003.04297*, 2020.
18. Yukang Chen, Tong Yang, Xiangyu Zhang, Gaofeng Meng, Xinyu Xiao, and Jian Sun. Detnas: Backbone search for object detection. In *NIPS*, pages 6638–6648, 2019.
19. Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Jixiang Li. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv preprint arXiv:1907.01845*, 2019.
20. Xiangxiang Chu, Tianbao Zhou, Bo Zhang, and Jixiang Li. Fair darts: Eliminating unfair advantages in differentiable architecture search. In *Proc. of ECCV*, 2020.
21. Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11408–11417, 2019.
22. Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *IJCV*, 2010.
23. Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 2010.
24. D. Gabor. Electrical engineers part iii: Radio and communication engineering, j. *Journal of the Institution of Electrical Engineers - Part III: Radio and Communication Engineering 1945-1948*, 1946.
25. D. Gabor. Theory of communication. part 1: The analysis of information. *Journal of the Institution of Electrical Engineers-Part III: Radio and Communication Engineering*, 1946.
26. Ross Girshick. Fast r-cnn. In *ICCV*, 2015.
27. Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.
28. Ross Girshick, Ilija Radosavovic, Georgia Gkioxari, Piotr Dollár, and Kaiming He. Detectron. https://github.com/facebookresearch/detectron, 2018.
29. I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv*, 2014.
30. Jiaxin Gu, Ce Li, Baochang Zhang, Jungong Han, Xianbin Cao, Jianzhuang Liu, and David Doermann. Projection convolutional neural networks for 1-bit cnns via discrete back propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.
31. Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *CVPR*, 2006.

32. Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NIPS*, pages 1135–1143, 2015.
33. Yiwen Han, Xiaofei Wang, Victor Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. Convergence of edge computing and deep learning: A comprehensive survey. In *arXiv*, 2019.
34. Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *CVPR*, 2020.
35. Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *ICCV*, 2017.
36. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
37. Olivier J Hénaff, Ali Razavi, Carl Doersch, SM Eslami, and Aaron van den Oord. Data-efficient image recognition with contrastive predictive coding. *arXiv:1905.09272*, 2019.
38. Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *Computer Science*, 14(7):38–39, 2015.
39. R Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Phil Bachman, Adam Trischler, and Yoshua Bengio. Learning deep representations by mutual information estimation and maximization. In *ICLR*, 2019.
40. Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
41. Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
42. Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.
43. Dahyun Kim, Kunal Pratap Singh, and Jonghyun Choi. Learning architectures for binary networks. In *Proc. of ECCV*, pages 575–591, 2020.
44. Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
45. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.
46. En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. In *IEEE Transactions on Wireless Communications*, 2019.
47. Guohao Li, Guocheng Qian, Itzel C Delgadillo, Matthias Muller, Ali Thabet, and Bernard Ghanem. Sgas: Sequential greedy architecture search. In *Proc. of CVPR*, 2020.
48. Rundong Li, Yan Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. Fully quantized network for object detection. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.
49. Hanwen Liang, Shifeng Zhang, Jiacheng Sun, Xingqiu He, Weiran Huang, Kechen Zhuang, and Zhenguo Li. DARTS+: improved differentiable architecture search with early stopping. *arXiv*, 2019.
50. Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.
51. Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.
52. Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *ECCV*, 2014.

53. Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision (ECCV)*, pages 740–755, 2014.
54. Chenxi Liu, Piotr Dollár, Kaiming He, Ross Girshick, Alan Yuille, and Saining Xie. Are labels necessary for neural architecture search? *arXiv:2003.12056*, 2020.
55. Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision*, pages 19–34, 2018.
56. H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. In *ICLR*, 2019.
57. Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018.
58. A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2017.
59. Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. Mobile edge computing: Survey and research outlook. In *arXiv*, 2017.
60. Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv:1807.03748*, 2018.
61. Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7(15):510, 2008.
62. Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.
63. Hai Phan, Zechun Liu, Dang Huynh, Marios Savvides, Kwang-Ting Cheng, and Zhiqiang Shen. Binarizing mobilenet via evolution-based searching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13420–13429, 2020.
64. Juan C. Pérez, Motasem Alfarra, Guillaume Jeanneret, Adel Bibi, Ali Kassem Thabet, Bernard Ghanem, and Pablo Arbeláez. Robust gabor networks. *arXiv*, 2019.
65. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.
66. Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.
67. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NeurIPS*, 2015.
68. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016.
69. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
70. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
71. Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *Thirty-First AAAI conference on artificial intelligence*, 2017.
72. Antti Tarvainen and Harri Valpola. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *NIPS*, 2017.
73. Yonglong Tian, Dilip Krishnan, and Phillip Isola. Contrastive multiview coding. *arXiv:1906.05849*, 2019.
74. Yonglong Tian, Dilip Krishnan, and Phillip Isola. Contrastive representation distillation. In *ICLR*, 2020.
75. Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. Tbn: Convolutional neural network with ternary inputs and binary weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 315–332, 2018.

76. Xiaolong Wang and Abhinav Gupta. Unsupervised learning of visual representations using videos. In *CVPR*, 2015.
77. Eric Wong, Leslie Rice, and J. Zico Kolter. Fast is better than free: Revisiting adversarial training. In *ICLR*, 2020.
78. Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. In *Proceedings of the International Conference on Learning Representationss*, pages 1–14, 2018.
79. Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019.
80. Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. Unsupervised feature learning via non-parametric instance discrimination. In *CVPR*, 2018.
81. C. Xie, Y. Wu, L. V. D. Maaten, A. L. Yuille, and K. He. Feature denoising for improving adversarial robustness. In *CVPR*, 2019.
82. Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. Pc-darts: Partial channel connections for memory-efficient architecture search. *arXiv preprint arXiv:1907.05737*, 2019.
83. Shenjun Xue, Hanlin Chen, Chunyu Xie, Baochang Zhang, Xuan Gong, and David S. Doermann. Fast and unsupervised neural architecture evolution for visual representation learning. *IEEE Computational Intelligence Magazine*, 16:22–32, 2021.
84. Shenjun Xue, Runqi Wang, Baochang Zhang, Tian Wang, Guodong Guo, and David S. Doermann. Idarts: Interactive differentiable architecture search. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1143–1152, 2021.
85. C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *ICML*, 2019.
86. Hongyuan Yu and Houwen Peng. Cyclic differentiable architecture search. *arXiv*, 2020.
87. Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R Manmatha, et al. Resnest: Split-attention networks. *arXiv:2004.08955*, 2020.
88. Junhe Zhao, Sheng Xu, Baochang Zhang, Jiaxin Gu, David Doermann, and Guodong Guo. Towards compact 1-bit cnns via bayesian learning. *International Journal of Computer Vision*, pages 1–25, 2022.
89. Xiawu Zheng, Rongrong Ji, Lang Tang, Yan Wan, Baochang Zhang, Yongjian Wu, Yunsheng Wu, and Ling Shao. Dynamic distribution pruning for efficient network architecture search. *arXiv preprint arXiv:1905.13543*, 2019.
90. Xiawu Zheng, Rongrong Ji, Lang Tang, Baochang Zhang, Jianzhuang Liu, and Qi Tian. Multinomial distribution learning for effective neural architecture search. In *CVPR*, 2019.
91. Xiawu Zheng, Rongrong Ji, Lang Tang, Baochang Zhang, Jianzhuang Liu, and Qi Tian. Multinomial distribution learning for effective neural architecture search. In *ICCV*, October 2019.
92. Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. Towards effective low-bitwidth convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7920–7928, 2018.
93. Chengxu Zhuang, Alex Lin Zhai, and Daniel Yamins. Local aggregation for unsupervised learning of visual embeddings. In *ICCV*, 2019.
94. B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.
95. Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, pages 1–16, 2017.
96. Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.

Chapter 4
Quantization of Neural Networks

4.1 Introduction

Quantization has emerged as a highly successful strategy for both training and inference of neural networks (NN). While the challenges of numerical representation and quantization have been long-standing in digital computing, NNs offer unique opportunities for advancements in this area. Although this survey primarily focuses on quantization for inference, it is important to acknowledge that quantization has also shown promise in NN training [2, 7, 15, 25].

In particular, innovations in half-precision and mixed precision training have played a crucial role in achieving higher throughput in AI accelerators [9, 20]. However, pushing beyond half-precision without extensive tuning has proven to be a significant challenge, and recent research on quantization has mainly centered around the inference stage of neural networks.

4.2 Quantitative Arithmetic Principles

Given a neural network (NN) model with N layers, we represent the set of weights as $\mathbf{W} = \mathbf{w}^n n = 1^N$ and the set of input features as $\mathbf{A} = \mathbf{a}^n inn = 1^N$. Here, $\mathbf{w}^n$ is the convolutional weight matrix for the n-th layer, with dimensions $C^n out \times C_{in}^n$, where C_{in}^n and C_{out}^n are the input and output channel numbers, respectively. Similarly, $\mathbf{a}_{in}^n$ is the input feature map for the n-th layer, with dimensions C_{in}^n.

The output feature map $\mathbf{a}_{out}^n$ of the n-th layer can be technically formulated as:

$$\mathbf{a}_{out}^n = \mathbf{w}^n \cdot \mathbf{a}_{in}^n, \tag{4.1}$$

B. Zhang et al., *Neural Networks with Model Compression*,
Computational Intelligence Methods and Applications,
https://doi.org/10.1007/978-981-99-5068-3_4

where $\cdot$ represents matrix multiplication. For simplicity, we omit the nonlinear activation function in this formulation. Following the prior works [24], quantized neural network (QNN) intends to represent $\mathbf{w}^n$ and $\mathbf{a}^n$ in a low-bit format as:

$$\mathbb{Q} := \{q_1, \cdots, q_U\},$$

where $q_i, i = 1, \cdots, U$ satisfying $q_1 < \cdots < q_U$, are defined as quantized values of the original variable. Note that x can be the input feature $\mathbf{a}^n$ or the weights $\mathbf{w}^n$. In this way, $\mathbf{q}^{\mathbf{w}^n} \in \mathbb{Q}^{C^n_{out} \times C^n_{in}}$ and $\mathbf{q}^{\mathbf{a}^n_{in}} \in \mathbb{Q}^{C^n_{in}}$ such that the float-point convolutional outputs can be approximated by the efficient XNOR and bit-count instructions as:

$$\mathbf{a}^n_{out} \approx \mathbf{q}^{\mathbf{w}^n} \odot \mathbf{q}^{\mathbf{a}^n_{in}}. \tag{4.2}$$

The key challenge in QNNs is how to define the quantization set $\mathbb{Q}$, and the methods to achieve this are further described in the following sections.

4.3 Uniform and Nonuniform Quantization

In quantized neural networks (QNNs), we need to define a function that can quantize the weights and activations of the neural network to a finite set of values. One popular choice for this quantization function is the uniform quantization function, which is defined as follows:

$$\mathbf{q}^x = \text{INT}\left(\frac{x}{S}\right) - Z, \tag{4.3}$$

where x is a real-valued input (activation or weight), S is a real-valued scaling factor, and Z is an integer zero point. The function INT converts a real number to an integer value using a rounding technique such as rounding to the nearest integer or truncation. In other words, the quantization function maps real values x to some integer value, allowing us to represent the original continuous values with a finite set of discrete values. This method of quantization is also known as uniform quantization.

Besides, nonuniform quantization methods produce quantized values that are not necessarily uniformly spaced. The formal definition of nonuniform quantization is shown as:

$$\mathbf{q}^x = \begin{cases} q_1, & \text{if } x \leq \Delta_1, \\ \cdots \\ q_i, & \text{if } \Delta_{i-1} < x \leq \Delta_i, \\ \cdots \\ q_U, & \text{if } x > \Delta_U. \end{cases} \tag{4.4}$$

where q_i represents the discrete quantization levels and Δ_i denotes the quantization steps. When the value of a real number x falls between the quantization steps $\Delta_i - 1$ and $i + 1$, the quantizer Q projects it to the associated quantization level q_i. It should be noted that neither q_i nor Δ_i are evenly spaced.

Nonuniform quantization techniques offer the potential to achieve higher accuracy for a fixed bit width compared to uniform quantization. This is because nonuniform quantization allows for better representation of data distributions by focusing on essential value regions and determining appropriate dynamic ranges. One common scenario where nonuniform quantization is beneficial is when dealing with bell-shaped distributions of weights and activations, which often have long tails. In such cases, various nonuniform quantization methods have been developed to accommodate these specific distributions. One popular approach is the rule-based nonuniform quantization using a logarithmic distribution. In this method, the quantization steps and levels are increased exponentially instead of linearly.

Recent advances in quantization techniques have treated quantization as an optimization problem to enhance performance. The objective is to minimize the difference between the original tensor and its quantized version by adjusting the quantization steps or levels in the quantizer $\mathbf{q}^x$. This can be formulated as an optimization problem:

$$\min_{\mathbf{q}} |\mathbf{q}^x - x|_2^2 \tag{4.5}$$

Nonuniform quantization can also benefit from learnable quantizers, where the quantization steps are optimized through an iterative process or gradient descent along with the model parameters.

Overall, nonuniform quantization offers the advantage of better representing data by distributing bits and discretizing the parameter range unevenly. However, implementing nonuniform quantization effectively on standard computer hardware, such as GPUs and CPUs, can be challenging. As a result, uniform quantization remains the predominant method due to its straightforward implementation and efficient mapping to hardware, making it more suitable for practical deployment in various computing platforms.

4.4 Symmetric and Asymmetric Quantization

The choice of the scaling factor, S, in uniform quantization (Eq. 4.2) is critical as it determines the granularity of quantization and ultimately impacts the accuracy of the quantized representation. The value of S affects how the range of real values, x, is divided into a specified number of segments, and it directly influences the size of each partition. The clip range $[\alpha, \beta]$ defines the range of real values that should be quantized, and the bit width b determines the number of bits used for quantization. The formula for S is given by:

$$S = \frac{\beta - \alpha}{2^b - 1}, \tag{4.6}$$

where $[\alpha, \beta]$ is the clip range and b is the bit width. Choosing an appropriate clip range is crucial as it directly affects the quantization precision and the overall quality of the quantized model. This process is known as calibration, an essential step in uniform quantization.

Asymmetric quantization may use a tighter clip range compared to symmetric quantization. This is especially useful for signals with imbalanced values, such as activations after ReLU, which always have nonnegative values.

Symmetric quantization, on the other hand, simplifies the quantization function by centering the zero point at $Z = 0$, resulting in the following expression:

$$\mathbf{q}^x = \mathrm{INT}\left(\frac{x}{S}\right). \tag{4.7}$$

In practice, using the whole-range approach often leads to greater accuracy. Symmetric quantization is commonly employed for quantizing weights due to its simplicity and reduced computational cost during inference. However, for quantizing activations, asymmetric quantization may be more effective as the offset in asymmetric activations can be absorbed into the bias or used to initialize the accumulator, leading to improved performance.

4.5 Comparison of Different Quantization Methods

4.5.1 LSQ: Learned Step Size Quantization

Fixed quantization methods that rely on user-defined settings do not guarantee optimal network performance and may still produce suboptimal results even if they minimize quantization error. An alternative approach is learning the quantization mapping by minimizing task loss, directly improving the desired metric. However, this method is challenging because the quantizer is discontinuous and requires an accurate approximation of its gradient, which existing methods [8] have done roughly that overlooks the effects of transitions between quantized states.

This section introduces a new method for learning the quantization mapping for each layer in a deep network called learned step size quantization (LSQ) [13]. LSQ improves on previous methods with two key innovations. First, we offer a simple way to estimate the gradient of the quantizer step size, considering the impact of transitions between quantized states. This results in more advanced optimization when learning the step size as a model parameter. Second, we introduce a heuristic to balance the magnitude of step size updates with weight updates, leading to improved convergence. Our approach can be used to quantize both activations and weights and is compatible with existing techniques for back propagation and stochastic gradient descent.

4.5.1.1 Notations

The goal of quantization in deep networks is to reduce the precision of the weights and the activations during the inference time to increase computational efficiency. Given the data to quantize v, the quantizer step size s, and the number of positive and negative quantization levels (Q_P and Q_N), a quantizer is used to compute $\hat{v}$, a quantized representation on the whole scale of the data, and $\hat{v}$, a quantized representation of the data at the same scale as v:

$$\bar{v} = \lfloor clip(v/s, -Q_N, Q_P) \rceil \tag{4.8}$$

$$\hat{v} = \bar{v} \times s \tag{4.9}$$

This technique uses low-precision inputs, represented by $\bar{w}$ and $\bar{x}$, in matrix multiplication units for convolutional or fully connected layers in deep learning networks. The low-precision integer matrix multiplication units can be computed efficiently, and a step size then scale the output with a relatively low-cost high-precision scalar-tensor multiplication. This scaling step has the potential to be combined with other operations, such as batch normalization, through algebraic merging, as shown in Fig. 4.1. This approach aims to minimize the memory and computational costs associated with matrix multiplication.

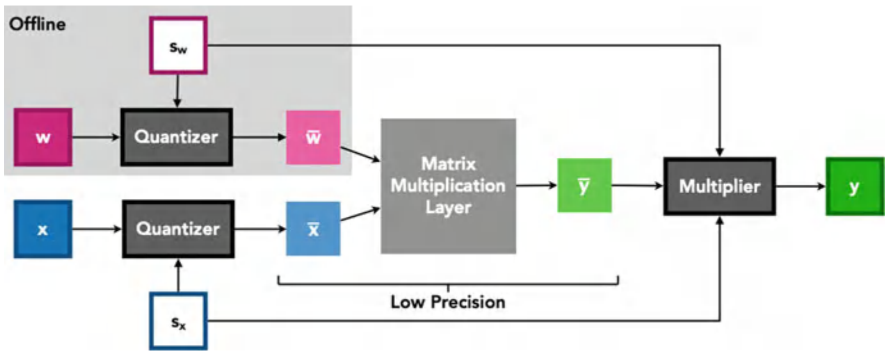


Fig. 4.1 Computation of a low-precision convolution or fully connected layer, as envisioned here

4.5.1.2 Step Size Gradient

LSQ offers a way of determining s based on the training loss through the incorporation of a gradient into the step size parameter of the quantizer as:

$$\frac{\partial \hat{v}}{\partial s} = \begin{cases} -v/s + \lfloor v/s \rceil, & \text{if } -Q_N < v/s < Q_p, \\ -Q_N, & \text{if } v/s \leq x, \\ Q_P, & \text{if } v/s \geq Q_p. \end{cases} \tag{4.10}$$

The gradient is calculated using the straight-through estimator, as proposed by [4], to approximate the gradient through the round function as a direct pass. The round function remains unchanged to differentiate downstream operations, while all other operations are differentiated conventionally.

The gradient calculated by LSQ is different from other similar approximations (Fig. 4.2) in that it does not transform the data before quantization (Jung et al., 2018) or estimate the gradient by algebraically canceling terms after removing the round operation from the forward equation, resulting in $\partial\hat{v}/\partial s = 0$ when $-Q_N < v/s < Q_P$ [8]. In these previous methods, the proximity of v to the transition point between quantized states does not impact the gradient of the quantization parameters. However, it is intuitive that the closer a value of v is to a quantization transition point, the more likely it is to change its quantization bin $\hat{v}$ with a slight change in s, resulting in a significant jump in $\hat{v}$. This means that $\partial\hat{v}/\partial s$ should increase as the distance from v to a transition point decreases, as observed in the LSQ gradient. Notably, this gradient emerges naturally from the simple quantizer formulation and the use of the straight-through estimator for the round function.

In LSQ, each layer of weights and each layer of activations have their unique step size represented as a 32-bit floating point value. These step sizes are initialized

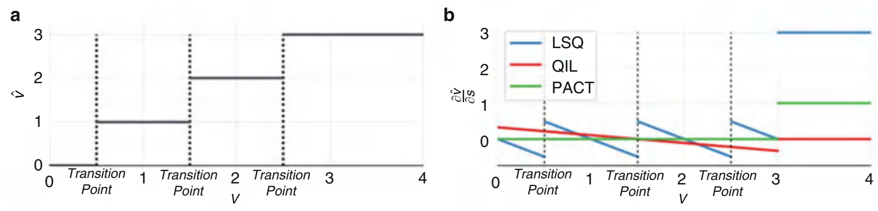


Fig. 4.2 Given $s = 1$, $Q_N = 0$, $Q_P = 3$, (**a**) quantizer output and (**b**) gradients of the quantizer output concerning step size, s, for LSQ, or a related parameter controlling the width of the quantized domain (equal to $s(QP + QN)$) for QIL [26] and PACT [8]. The gradient employed by LSQ is sensitive to the distance between v and each transition point, whereas the gradient employed by QIL [26] is sensitive only to the distance from quantizer clip points and the gradient employed by PACT [8] is zero everywhere below the clip point. Here, we demonstrate that networks trained with the LSQ gradient reach a higher accuracy than those trained with the QIL or PACT gradients in prior work

to $2|v|/\sqrt{Q_P}$ and calculated from the initial weight values or the first batch of activations, respectively.

4.5.1.3 Step Size Gradient Scale

It has been demonstrated that good convergence during training can be achieved when the ratio of average update magnitude to average parameter magnitude is consistent across all weight layers in a network. Setting the learning rate correctly helps prevent updates from being too large and causing repeated overshooting of local minima or too small, leading to a slow convergence time. Based on this reasoning, it is reasonable to assume that each step size should also have its update magnitude proportional to its parameter magnitude, similarly to the weights. Therefore, for a network trained on a loss function L, the ratio

$$R = \frac{\nabla_s L}{s} / \frac{\|\nabla_w L\|}{\|w\|}, \tag{4.11}$$

should be close to 1, where $\|x\|$ denotes the l2-norm of z. However, as precision increases, the step size parameter is expected to be smaller (due to finer quantization), and the step size updates are expected to be larger (due to the accumulation of updates from more quantized items when computing its gradient). A gradient scale g is multiplied by the step size loss to address this. For the weight step size, g is calculated as $1/\sqrt{N_W Q_P}$, and for the activation step size, g is calculated as $1/\sqrt{N_W Q_P}$, where N_W is the number of weights in a layer and N_f is the number of features in a layer.

4.5.1.4 Training

LSQ trains the model quantizers by making the step sizes learnable parameters, with the loss gradient computed using the quantizer gradient mentioned earlier. In contrast, other model parameters can be trained with conventional techniques. A common method of training quantized networks [10] is employed where full-precision weights are stored and updated, while quantized weights and activations are used for forward and backward passes. The gradient through the quantizer round function is calculated using the straight-through estimator [4] so that:

$$\frac{\partial \hat{v}}{\partial v} = \begin{cases} 1, \text{ if } -Q_N < v/s < Q_p, \\ 0, \text{ otherwise}, \end{cases} \tag{4.12}$$

and stochastic gradient descent is used to update parameters.

For ease of training, the input to the matrix multiplication layers is set to $\hat{v}$, mathematically equivalent to the inference operations described above. The input activations and weights are set to 2, 3, 4, or 8 bits for all matrix multiplication layers except the first and last, which are always set to 8 bits. This standard practice in quantized networks has been shown to improve performance significantly. All other parameters are represented using FP32. Quantized networks are initialized using weights from a trained full-precision model with a similar architecture before being fine-tuned in the quantized space.

4.5.2 TRQ: Ternary Neural Networks with Residual Quantization

4.5.2.1 Preliminary

The main operation in deep neural networks is expressed as

$$z = \mathbf{w}^\top \mathbf{a}, \tag{4.13}$$

where $\mathbf{w} \in \mathbb{R}^n$ indicates the weight vector and $\mathbf{a} \in \mathbb{R}^n$ indicates the input activation vector computed by the previous network layer.

A ternary neural network means representing the floating-point weights and/or activations with ternary values. Formally, the quantization can be expressed as:

$$Q_x(\mathbf{x}) = \beta_x \mathbf{T_x}, \tag{4.14}$$

where $\mathbf{x}$ indicates floating-point parameters including weights $\mathbf{w}$ and activations $\mathbf{a}$ and $\mathbf{T_x}$ denotes ternary values after the quantization on $\mathbf{x}$. β_x is a scalar used to scale the ternary values, which can be computed from the floating-point parameters or learned via back propagation. $\mathbf{T_x}$ is usually obtained by thresholding function:

$$\mathbf{T_x} = \begin{cases} +1 & if\ \mathbf{x} > \Delta \\ 0 & if\ |\mathbf{x}| \leqslant \Delta, \\ -1 & if\ \mathbf{x} < -\Delta \end{cases} \tag{4.15}$$

where Δ denotes a fixed threshold used for quantization. With the ternary weights and activations, the vector multiplications in the forward propagation can be reformulated as:

$$z = Q_w(\mathbf{w})^\top Q_a(\mathbf{a}) = \beta_w \beta_a (\mathbf{T_w} \odot \mathbf{T_a}), \tag{4.16}$$

where $\odot$ represents the inner product for vectors with bitwise operations.

In general, the derivative of quantization function $Q_x(\mathbf{x})$ is non-differentiable and thus unpractical to directly apply the back propagation to perform the training phase.

For this issue, we follow the now widely adopted "straight-through estimator (STE)" [23] to approximate the partial gradient calculation, which is formally expressed as:

$$\frac{\partial Q_x(\mathbf{x})}{\partial \mathbf{x}} \approx \beta \mathbf{1}_{|\mathbf{x}| \leqslant 1}. \tag{4.17}$$

TNNs with Residual Quantization (TRQ)

Existing TNNs are based on directly thresholding method for ternary implementation, inevitably causing performance degradation due to an inaccurate mapping of full-precision values to ternary counterparts. To deal with the issue, residual quantization (TRQ) [29] is introduced to learn TNNs. TRQ can extract binarized stem and residual, respectively, by performing recursive quantization on full-precision weights, which are combined to generate refined ternary representation, leading to the stem-residual framework for TNNs.

In our stem-residual ternarization framework, the stem is first extracted as a coarse fitting for full-precision weight $\mathbf{w}$, which is calculated by performing sign$(\cdot)$ on $\mathbf{w}$ as:

$$\mathbf{S_w} = \alpha \text{sign}(\mathbf{w}), \tag{4.18}$$

where α is a learnable coefficient, which avoids a very careful tuning to seek the optimal quantization scale compared with the previous methods. Then, we further calculate the quantization error as:

$$\mathbf{R} = \mathbf{w} - \mathbf{S_w} \tag{4.19}$$

Furthermore, we calculate the residual $\mathbf{R_w}$ from $\mathbf{R}$ by performing sign$(\cdot)$ on the quantization error $\mathbf{R}$:

$$\mathbf{R_w} = \alpha \text{sign}(\mathbf{R}). \tag{4.20}$$

Based on Eqs. 4.18 and 4.20, we finally obtain our ternary weight designed for more accurate approximation as:

$$\mathbf{T_w} = \mathbf{S_w} + \mathbf{R_w}. \tag{4.21}$$

Up to now, we achieve the ternary quantization in a stem framework, with the full-precision weights quantized to ternary values, i.e., $\{-2\alpha, 0, 2\alpha\}$. Obviously, seeking a better coefficient α is significantly important for the effectiveness of quantizer, which would be elaborated in the following section.

Backward Propagation of TRQ

In the backward propagation, what need to be learned and updated are the full-precision weight $\mathbf{w}$ and the learnable coefficient α. For the stem-residual framework, the two kinds of parameters are jointly learned. And in each layer, TRQ updates the $\mathbf{w}$ first and then the α.

Update w: For $\mathbf{w}$ updating, the gradient through the quantizer to weights are estimated by a STE that pass the gradient whose weight value is in the range of (-2α, 2α):

$$\frac{\partial \mathbf{T_w}}{\partial \mathbf{w}} = \mathbf{1}_{|\mathbf{x}| \leqslant 2\alpha}. \tag{4.22}$$

Then, we can obtain the updating process of $\mathbf{w}$:

$$\delta_{\mathbf{w}} = \frac{\partial L}{\partial \mathbf{T_w}} \frac{\partial \mathbf{T_w}}{\partial \mathbf{w}}, \tag{4.23}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \delta_{\mathbf{w}}, \tag{4.24}$$

where L is the loss function and η is the learning rate.

Update α: The coefficient α determines the scale of binarized stem and residual, which is directly related to the quality of the ternary weights. Moreover, we also empirically find the recognition performance is quite sensitive to the α. Thus, rather than a coarse gradient acquired like $\mathbf{w}$, we disassemble the quantizer to calculate a finer gradient of α:

$$\frac{\partial \mathbf{T_w}}{\partial \alpha} = \operatorname{sign}(\mathbf{w}) + \operatorname{sign}(\mathbf{R}) + \alpha \frac{\partial \operatorname{sign}(\mathbf{R})}{\partial \alpha} \tag{4.25}$$

where

$$\begin{aligned} \frac{\partial \operatorname{sign}(\mathbf{R})}{\partial \alpha} &= \frac{\partial \operatorname{sign}(\mathbf{R})}{\partial \mathbf{R}} \frac{\partial \mathbf{R}}{\partial \alpha} \\ &= \mathbf{1}_{|\mathbf{R}| \leqslant 1} \left(-\operatorname{sign}(\mathbf{w})\right). \end{aligned} \tag{4.26}$$

Then, we can obtain the updating process of α:

$$\delta_\alpha = \sum \frac{\partial L}{\partial \mathbf{T_w}} \frac{\partial \mathbf{T_w}}{\partial \alpha}, \tag{4.27}$$

$$\alpha \leftarrow \alpha - \eta \delta_\alpha. \tag{4.28}$$

4.5.2.2 Generalization to n-Bit Quantization

We focus on ternary quantization in this paper, while it does not mean that TRQ is limited to ternary applications. Actually, TRQ could also be generalized to multiple bits by recursively encoding residual. In this section, we propose a feasible scheme for TRQ expansion, which is not the only way and could be further explored in the future work.

We obtain the subtly quantized weights by recursively performing quantization on full-precision weights. In this process, residual at different quantization levels is generated for refining the quantized weights. Here for n-bit ($n = 2, 3, 4, \ldots$) quantization, we define the residual at level i ($i = 1, 2, \ldots, 2^n - 3$) as $\mathbf{R}_{\mathbf{w}}^{i}$, which could be computed as:

$$\mathbf{R}_{\mathbf{w}}^{i} = \alpha \mathrm{sign}(\mathbf{w} - \mathbf{T}_{\mathbf{w}}^{i-1}), \tag{4.29}$$

where $\mathbf{T}_{\mathbf{w}}^{i-1}$ denotes the quantized weights at ($i - 1$)th level, and we recursively acquire the quantized weights at level i as:

$$\mathbf{T}_{\mathbf{w}}^{i} = \mathbf{T}_{\mathbf{w}}^{i-1} + \mathbf{R}_{\mathbf{w}}^{i}. \tag{4.30}$$

Here we regard the ternary quantization as the initial state for recursive quantization as:

$$\mathbf{T}_{\mathbf{w}}^{0} = \mathbf{S}_{\mathbf{w}} + \alpha \mathrm{sign}(\mathbf{w} - \mathbf{S}_{\mathbf{w}}). \tag{4.31}$$

Based on such recursive quantization, we could easily obtain the residual at different levels, thus refining the residual and reducing the approximation error with the full-precision counterparts.

For the updating of α in backward propagation, due to the complexity of recursive process, we just roughly estimate the gradient α by regarding it as the coefficient of $\mathbf{S}_{\mathbf{w}}$ and $\mathbf{R}_{\mathbf{w}}^{i}$:

$$\frac{\partial \mathbf{T}_{\mathbf{w}}}{\partial \alpha} = \mathbf{S}_{\mathbf{w}} + \sum_{i=0}^{2^n-3} \mathbf{R}_{\mathbf{w}}^{i}. \tag{4.32}$$

4.5.2.3 Complexity Analysis

A comprehensive comparison on computational complexity is shown in Table 4.1. We assume that the input number of the neuron is N, i.e., N inputs and one neuron output. For computational complexity of TNNs, we follow the setting of GXNOR-Net [11] for a comparison. As described in GXNOR-Net, with the event-driven paradigm, the resting computation would occur when the weight or activation of TNNs is zero, and the exception cases are achieved by XNOR operations. As a result, the computational complexity of TNNs is similar to BNNs, half of the network with 1-bit weights and 2-bit activations. Noted that for TRQ, the stem-residual framework is only employed on weights; thus, it also enjoys the low complexity $O(N)$ as normal TNNs.

Table 4.1 Operation overhead comparisons with different computing bit width

	Operations				
Bit width(A/W)	Multiplication	Accumulation	XNOR	BitCount	Complexity
32/32	N	N	0	0	–
1/1	0	0	N	1	O(N)
2/1	0	0	2N	1	O(2N)
ter/ter	0	0	0∼N	0/1	O(N)

4.5.2.4 Differences of TRQ from Existing Residual Quantization Methods

Residual quantization has been first proposed in high-order residual quantization (HORQ) [32] to enhance the performance of BNNs, further being explored by [16, 18] to be encoded into low-bit width CNNs. All above works compute residual error and recursively approximate it by a series of binary maps. However, limited by the residual scales, they can be just applied to n-bit quantization, with no generalization ability to arbitrary value quantization even parameter changes, such as the TNNs emphasized in this paper. Instead, our TRQ enjoys the flexibility by the skillful combination of the binarized stem and residual, thus enabling ternary quantization and even arbitrary value quantization by recursively refining the residual. Moreover, a key feature of the prior residual schemes is the use of analytically calculated scaling coefficients, which can be suboptimal. In contrast, our TRQ employs learnable coefficient α to minimize the training loss, thus fully utilizing the strength of back propagation algorithm to seek for the suitable quantization scale automatically.

4.5.2.5 Implementation Details

Data Preprocessing

For CIFAR-10/100, all the images are padded with 4 pixels on each side, and then a random 32×32 crop is applied, followed by a random horizontal flip. During inference, the scaled images are used without any augmentation. For ImageNet, training images are randomly cropped into the resolution of 224×224. After that, the images are normalized using the mean and standard deviation. No additional augmentations are performed except the random horizontal flip. However, for validation images, we use center crop instead of random crop and no flip is applied (Fig. 4.3).

Training Procedure

We conduct experiments mainly on ResNet [21] backbones, including ResNet-18 and ResNet-34. VGG-Small [38] is also leveraged for the CIFAR-10 and CIFAR-100 experiments. Similar with previous works [17, 28, 34], we do not quantize the first and last layers. For experiments on CIFAR-10/100, we run the training algorithm for 200 epochs with a batch size of 256. Besides, a linear learning rate

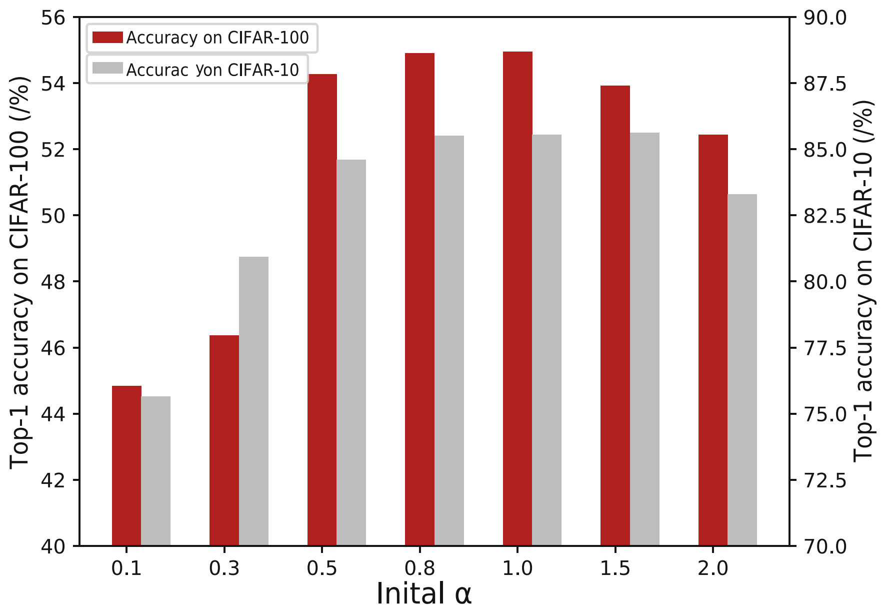


Fig. 4.3 The Top-1 accuracy (%) on CIFAR-10 and CIFAR-100 with different initial α

decay scheduler is used, and the initial learning rate is set to 0.01. For experiments on ImageNet, we train the models for up to 100 epochs with a batch size of 256. The learning rate starts from 0.001 and is decayed twice by multiplying 0.1 at 75th and 95th epoch. For all settings, Adam with momentum of 0.9 is adopted as the optimizer.

4.5.2.6 Ablation Study on CIFAR

In this section, we first perform hyperparameter sweeps to determine the value of initial α to use. Following this we analyze the necessity of α, then show TRQ's generalization to multiple bits, and finally evaluate the effectiveness of TRQ on CIFAR datasets.

Initial Value of α

The initiation of parameters is always important for network training. Thus, we set different initial values 0.10, 0.3, 0.5, 0.8, 1, 1.5, and 2 to α, to explore their influence on classification. The experiments are performed on the CIFAR-10/100 with ResNet-18 backbone. From the results on CIFAR-10 in Fig. 4.4, we can observe that the performance is similar when the initial values of α are set between 0.8 and 1.5, and the best performance can be obtained at 1.5. Meanwhile, from the results on CIFAR-100, the good performance plateau appears when initial α is at the

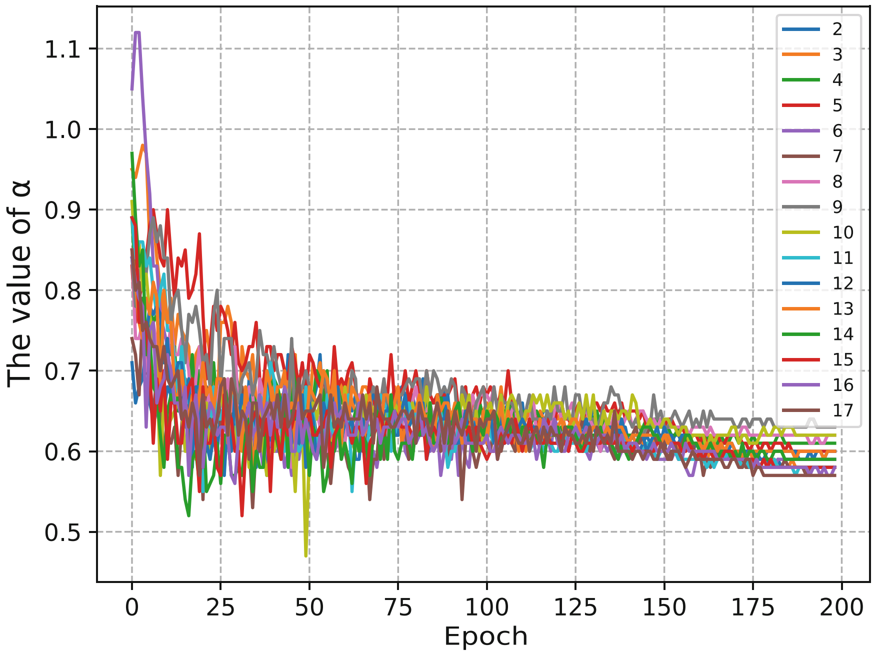


Fig. 4.4 Evolution of α values in different layers during training with ResNet-18 backbone on CIFAR-100

Table 4.2 The accuracy (%) of TRQ with and without α (TRQ and TRQ-wo) and with fixed $\alpha = 0.6$ (TRQ-0.6) on CIFAR-100

	Width	TRQ-wo	TRQ	TRQ-0.6
ResNet-18	16-16-32-64	52.1	54.9	53.9
ResNet-18	32-32-64-128	60.5	62.7	61.3
VGG-Small	–	62.6	65.4	60.5

range between 0.8 and 1, and the performance of initial value 1 performs slightly better than that of 0.8. For both CIFAR-10 and CIFAR-100, the performance of initial values outside the 0.5 to 1.5 is fairly worse, which shows the importance of setting the initial value of α carefully. Based on the above discoveries, we set the initial value of α as 1 in the following experiments, which shows a stably high classification performance on both two datasets.

Analysis of α

α is introduced in stem-residual framework to automatically seek for a reasonable quantization scale. To valid the necessity of α, we provide the experiments with and without α on CIFAR-100 with the backbone ResNet-18. As shown in Table 4.2, compared with the TRQ without α (TRQ-wo), TRQ achieves better performance by a large margin (more than 2%), thus indicating that α is quite important for training TRQ.

Simultaneously, as illustrated in Fig. 4.4, we explore how the value of α changes during training. It can be observed that α converges to around 0.6 with training. However, this doesn't mean that α should be fixed and not optimized. As shown in Table 4.2, we compare the results in two cases, i.e., α is fixed to 0.6 (TRQ-0.6) and α is optimized by back propagation. As we can see, when fixed α as 0.6, a greater performance decrease happens. When employed with VGG-Small backbone, the accuracy even drops nearly 5% compared with the learnable α, thus validating the superiority of the learnable α. We conjecture that is because with the learnable α in stem-residual framework, the quantizer could be automatically fine-tuned to find the best quantization mapping for each layer, thus yielding better performance than the fixed case.

Quantization Error

In order to better understand our TRQ, which achieves more accurate mapping between ternary weights and their full-precision counterparts, we adopt mean square error (MSE) [14] to calculate the quantization error between $\mathbf{w}$ and $\mathbf{T_w}$:

$$\mathbf{E} = \frac{1}{M}\sum\left(\frac{\mathbf{w}-\mathbf{T_w}}{\mathbf{w}}\right)^2, \tag{4.33}$$

where M denotes the total number of weights in each layer. In Fig. 4.5, we plot the quantization error for the 2th–17th layer of ResNet-18. The results show our methods (the red histogram) have lower quantization error compared with baseline (the gray histogram) which achieved with the method in Section 3.1 in most layers. In particular, the quantization error can be reduced by more than 25% (0.8 *vs* 0.6) in the 9th layer.

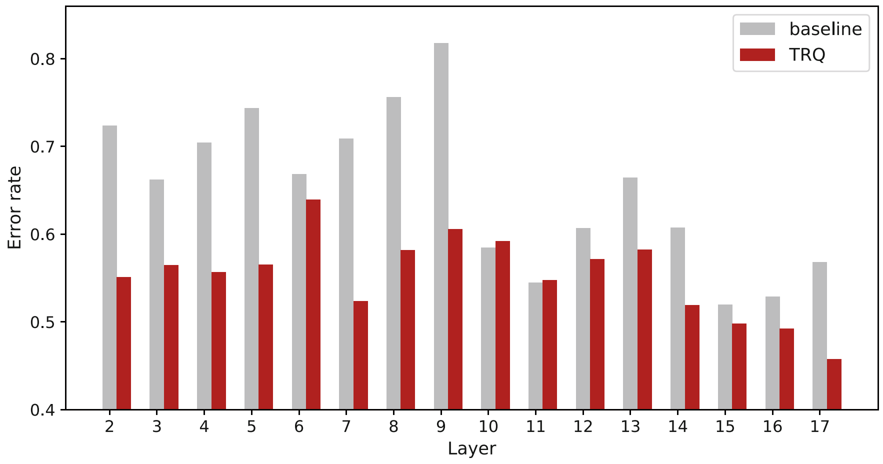


Fig. 4.5 Quantization error of TRQ and baseline based on ResNet-18 backbone

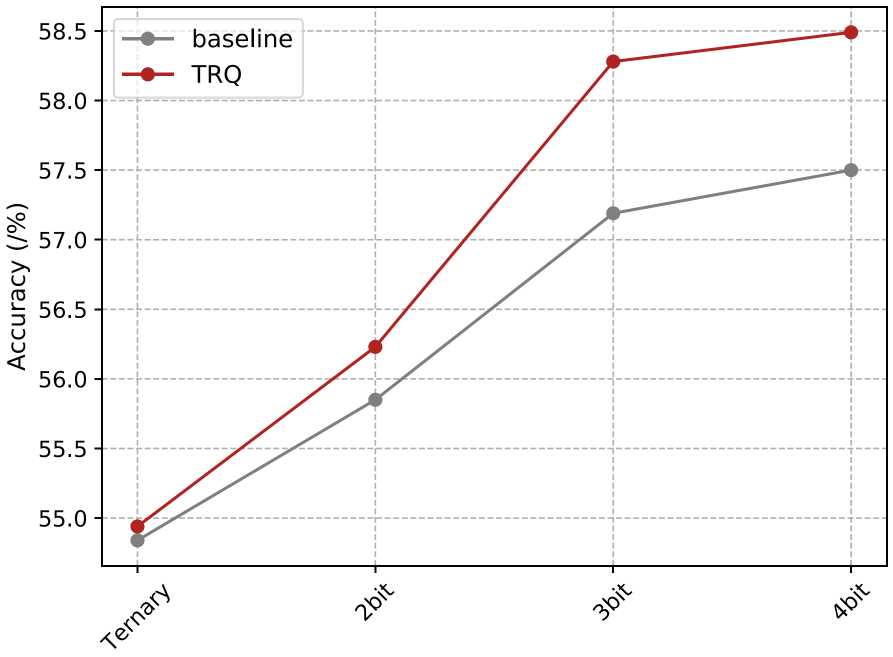


Fig. 4.6 The results of TRQ with multi-bits expansion on CIFAR-100

Generalization to n-Bit Quantization

We illustrate that our TRQ can not only improve the performance on ternary quantization but also could be generalized to multiple bits. Here we adopt the expansion method described in Sect. 3.4 and perform the experiments on CIFAR-100 with the backbone of ResNet-18. The baseline model is implemented in a similar way as DoReFa-Net [45]. As shown in Fig. 4.6, we can see that the accuracy of TRQ increases (56.2% $\rightarrow$ 58.3% $\rightarrow$ 58.5%) as the bit width increases from 2bit to 4bit, indicating that the compound residual at multi-levels could refine the quantized weights, thus improving the recognition accuracy. Moreover, our TRQ consistently surpasses the baseline on each bit width (0.4%, 1.1%, 1.0% on 2bit, 3bit, and 4bit, respectively), which demonstrates the superiority and potential of the residual quantization on multiple bits.

Evaluation on CIFAR

To validate the effectiveness of TRQ, here we perform ablation evaluation on CIFAR datasets. Three backbones are used in this experiment, including VGG-Small, ResNet-18 with the width of 16-16-32-64 and 32-32-64-128. We report the performance of baseline and TRQ on both CIFAR-10 and CIFAR-100 in Table 4.3. As shown in Table 4.3, for ResNet-18, TRQ achieves stable improvement on both CIFAR-10 and CIFAR-100 datasets compared with the corresponding baseline. Moreover, TRQ with the backbone ResNet-18 whose width is 32-32-64-128 even

Table 4.3 The experimental comparison of baseline and TRQ on CIFAR datasets

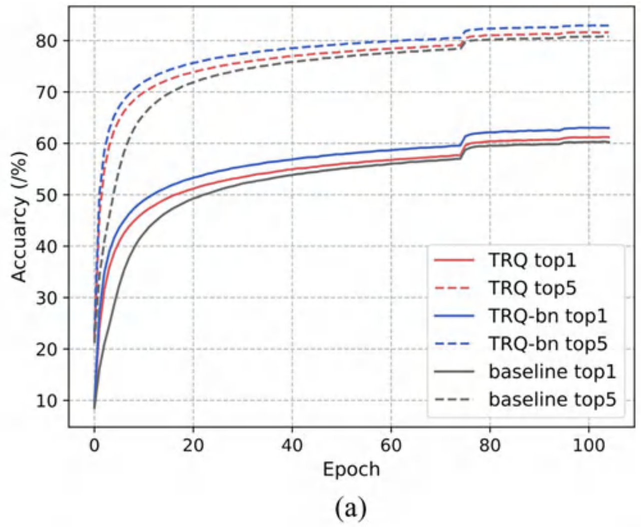
		CIFAR-10/%	CIFAR-100/%
ResNet-18	Full-precision	87.7	58.2
16-16-32-64	Baseline	85.2	54.8
	TRQ	85.5	54.9
ResNet-18	Full-precision	90.9	63.0
32-32-64-128	Baseline	87.5	60.6
	TRQ	89.3	62.7
VGG-Small	Full-precision	92.6	66.8
	Baseline	89.1	61.8
	TRQ	91.2	65.4

realizes nearly lossless ternarization on CIFAR-100 (only with a 0.3% performance drop). All these demonstrate the effectiveness of TRQ on ResNet. For VGG-Small, our TRQ consistently surpasses the baseline by a margin of 2.1% and 3.5% on CIFAR-10 and CIFAR-100, respectively, which further shows the general improvement brought by TRQ.

Comparison on ImageNet
We further analyze the effectiveness of TRQ on the large-scale dataset ImageNet. Since the dataset is challenged for network optimization, we use multi-batch normalization (multi-bn) strategy on ResNet architecture to alleviate optimization problems, which is termed as TRQ-bn in the experiment. For a basic block in TRQ-bn, three batch normalization layers are employed: the first is a pre-bn [43] before quantization, the second is a normal bn following the ternary convolutional layer, and the last is an additional bn following the shortcut. Such multi-bn can significantly improve the network performance by improving the distribution of feature maps with only small additional memory and computation.

We illustrate the training and validation accuracy curves of baseline, TRQ, and TRQ-bn in Fig. 4.7, which are based on a ResNet-18 backbone. From Fig. 4.7, we can observe that TRQ greatly improves the convergence speed of TNNs. Simultaneously, from the results in Table 4.4, TRQ improves baseline by 1.0% on both ResNet-18 and ResNet-34 Top-1 accuracy, which validates the effectiveness of our TRQ on large-scale dataset. Moreover, TRQ-bn could further obtain an improvement of about 2% on both the two networks, which finally achieves approximately 93% of the accuracy of their full-precision counterparts.

To evaluate the overall performance of TRQ, we further compare TRQ with four state-of-the-art quantization on ImageNet, i.e., XNOR-Net [36], BiReal-Net [34], LQ-Net [43], HWGQ [6], and RTN [30]. To perform fair comparison with RTN whose quantization procedure of weight and activation are both improved, we apply residual quantization to activation as well, leading to TRQ-a. The results are reported in Table 4.4. From Table 4.4, by comparing with the state-of-the-art BNNs including XNOR-Net and BiReal-Net, we can significantly boost the performance.



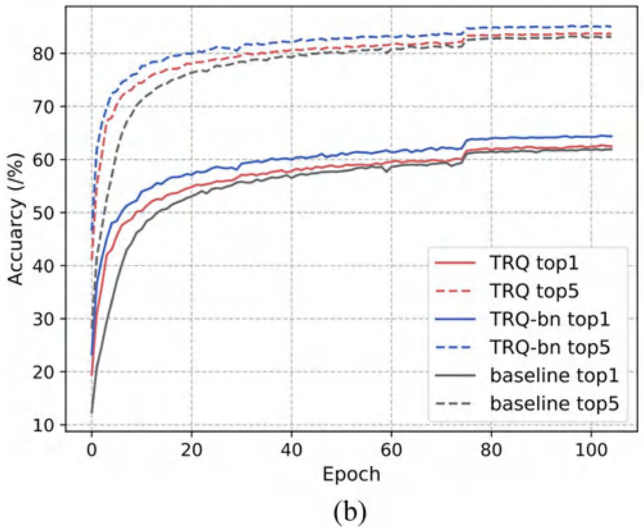


Fig. 4.7 Accuracy curves of baseline, TRQ, and TRQ-bn with ResNet-18 backbone on ImageNet. (**a**) Training accuracy curves on ImageNet. (**b**) Validation accuracy curves on ImageNet

For example, TRQ outperforms XNOR-Net and BiReal-Net by 11% and 6% on ResNet-18, respectively. It is because that ternary values $\{-1, 0, 1\}$ have stronger representational capability than binary values $\{-1, 1\}$, while the complexity of the two methods is the same because of the event-driven paradigm in TNNs. Moreover, our TRQ can even achieve better performance than the methods with $O(2N)$ complexity, including the "A/W = 2/1" cases in LQ-Net and HWGQ. Besides,

Table 4.4 Comparison of Top-1 and Top-5 accuracy on ImageNet

Network	Method	A/W	Top-1/%	Top-5/%	Complexity
ResNet-18	Full-precision	32/32	69.3	89.2	–
	Baseline	ter/ter	61.6	82.7	O(N)
	TRQ(ours)	ter/ter	62.6	83.7	O(N)
	TRQ-bn(ours)	ter/ter	64.4	85.1	O(N)
	TRQ-a(ours)	ter/ter	65.7	85.9	O(N)
	RTN	ter/ter	64.5	–	O(N)
	XNOR-Net	1/1	51.2	73.2	O(N)
	BiReal-Net	1/1	56.4	79.5	O(N)
	LQ-Net	2/1	62.6	84.3	O(2N)
ResNet-34	Full-precision	32/32	73.3	91.3	–
	Baseline	ter/ter	65.2	85.7	O(N)
	TRQ(ours)	ter/ter	66.2	86.3	O(N)
	TRQ-bn(ours)	ter/ter	68.2	87.7	O(N)
	BiReal-Net	1/1	62.2	83.9	O(N)
	LQ-Net	2/1	66.6	86.9	O(2N)
	HWGQ	2/1	64.3	85.7	O(2N)

our TRQ-a surpasses RTN 1.2% in accuracy, demonstrating the advantage of the effective residual quantization scheme.

4.5.3 OMPQ: Orthogonal Mixed Precision Quantization

Recently, we have seen a noticeable trend in deep learning, that models have a rapidly increasing complexity [21, 22, 37–39, 44]. Due to practical limitations such as latency, battery, and temperature, the host hardware where the models are deployed cannot keep up with this trend. It results in a large, ever-increasing gap between computational demands and resources. To address this issue, network quantization [3, 23, 27, 33, 36], which maps single-precision floating-point weights or activations to lower bit integers for compression and acceleration, has attracted considerable research attention. Network quantization can be naturally formulated as an optimization problem, and a straightforward approach is to relax the constraints to make it a tractable optimization problem at the cost of an approximated solution. e.g. straight-through estimation (STE) [4].

With the recent development of inference hardware, arithmetic operations with variable bit width have become possible, bringing further flexibility to network quantization. To take full advantage of hardware capabilities, mixed precision quantization [12, 31, 41, 42] aims to quantize different network layers to different bit widths to achieve a better trade-off between compression ratio and accuracy. While benefiting from the extra flexibility, mixed precision quantization also needs a more complicated and challenging optimization problem with a non-differentiable and extremely nonconvex objective function. Therefore, existing approaches [12, 31, 41, 42] often require numerous data and computing resources to search for the optimal bit configuration.

For example, FracBits [42] approximates bit width by performing a first-order Taylor expansion on the adjacent integer, making the bit variable differentiable. This allows it to integrate the search process into training to obtain the optimal bit configuration. However, the search and training process still requires many computation resources to derive a decent solution. To resolve the significant demand for training data, Dong et al. [12] use the average eigenvalue of the Hessian matrix of each layer as the metric for bit allocation. However, the matrix-free Hutchinson algorithm to implicitly calculate the average of the eigenvalues of the Hessian matrix still needs 50 iterations for each network layer. Another direction is black-box optimization. For example, Wang et al. [41] use reinforcement learning to allocate the bits of each layer. Li et al. [31] use an evolutionary search algorithm [19] to derive the optimal bit configuration, together with a block reconstruction strategy to optimize the quantized model efficiently. But the population evolution process requires input data $1,024$ and iterations 100, which are time-consuming.

Different from the existing approaches of black-box optimization or constraint relaxation, we propose constructing a proxy metric, which could have a substantially different form, but be highly correlated with the objective function of the original linear programming. In general, we propose to obtain the optimal bit configuration by using the orthogonality of the neural network. Specifically, we deconstruct the neural network model into a set of functions and define the orthogonality of the model by extending its definition from a function $f : \mathbb{R} \to \mathbb{R}$ to the entire network $f : \mathbb{R}^m \to \mathbb{R}^n$. Orthogonality measurement could be performed efficiently with Monte Carlo sampling and the Cauchy-Schwarz inequality, based on which we propose an efficient metric named **OR**thogonality **M**etric (ORM) as the proxy metric. As illustrated in Fig. 4.8, we only need a single-pass search process on a small amount of data with ORM. In addition, we derive an equivalent form of ORM to accelerate the computation.

On the other hand, model orthogonality and quantization accuracy are positively correlated on different networks. Therefore, maximizing model orthogonality is taken as our objective function. Meanwhile, our experiments show that layer orthogonality and bit width are positively correlated. We assign a more significant bit width to the layer with larger orthogonality while combining specific constraints to construct a linear programming problem. The optimal bit configuration can be obtained simply by solving the linear programming problem (Fig. 4.9).

4.5.3.1 Network Orthogonality

A neural network can be naturally decomposed into a set of layers or functions. Formally, for the given input $x \in \mathbb{R}^{1\times(C\times H\times W)}$, we decompose a neural network into $\mathcal{F} = \{f_1, f_2, \cdots, f_L\}$, where f_i represents the transformation from the input x

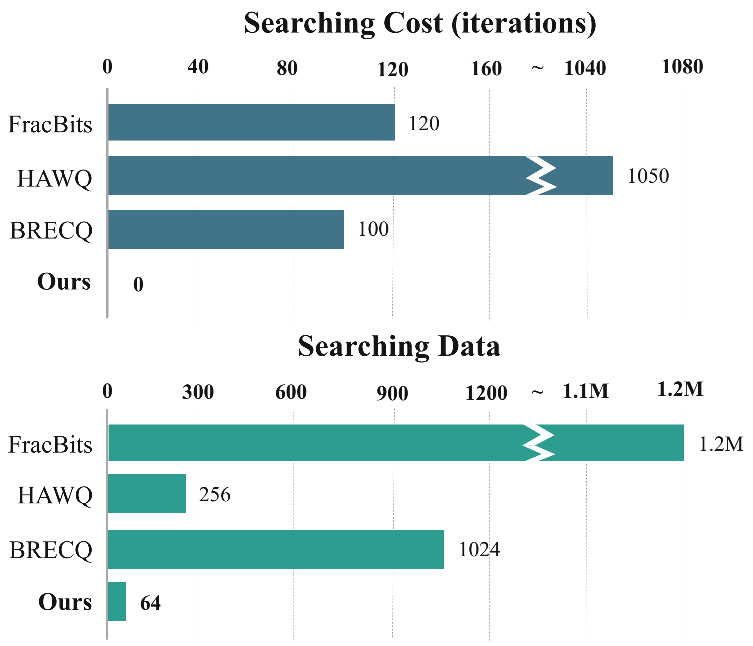


Fig. 4.8 Comparison of the resources used to obtain the optimal bit configuration between our algorithm and other mixed precision algorithms (FracBits [42], HAWQ [12], BRECQ [31]) on ResNet-18. "Search Data" is the number of input images

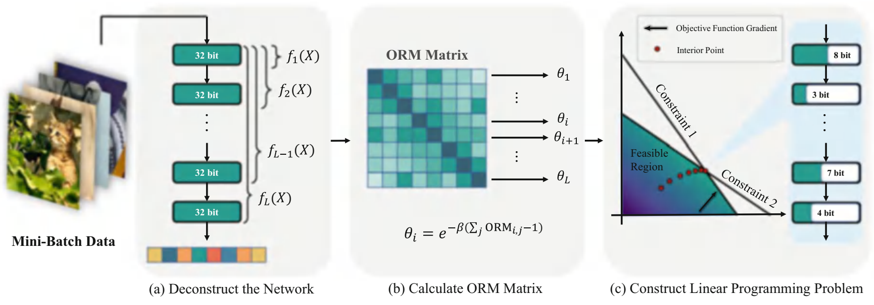


Fig. 4.9 Overview. Left: Deconstruct the model into a set of functions $\mathcal{F}$. Middle: ORM symmetric matrix calculated from $\mathcal{F}$. Right: Linear programming problem constructed by the importance factor θ to derive optimal bit configuration

to the result of the i-th layer. In other words, if g_i represents the function *of* of the i-th layer, then $f_i(x) = g_i\big(f_{i-1}(x)\big) = g_i\Big(g_{i-1}\big(\cdots g_1(x)\big)\Big)$. Here, we introduce the inner product [1] between the functions f_i and f_j, which is formally defined as:

$$\langle f_i, f_j \rangle_{P(x)} = \int_{\mathcal{D}} f_i(x) P(x) f_j(x)^T dx, \tag{4.34}$$

where $f_i(x) \in \mathbb{R}^{1\times(C_i\times H_i\times W_i)}$, $f_j(x) \in \mathbb{R}^{1\times(C_j\times H_j\times W_j)}$ are known functions when the model is given and $\mathcal{D}$ is the domain of x. If we set $f_i^{(m)}(x)$ to be the m-th element of $f_i(x)$, then $P(x) \in \mathbb{R}^{(C_i\times H_i\times W_i)\times(C_j\times H_j\times W_j)}$ is the probability density matrix between $f_i(x)$ and $f_j(x)$, where $P_{m,n}(x)$ is the probability density function of the random variable $f_i^{(m)}(x)\cdot f_j^{(n)}(x)$. According to the definition in [1], $\langle f_i, f_j\rangle_{P(x)} = 0$ means that f_i and f_j are weighted orthogonal. In other words, $\langle f_i, f_j\rangle_{P(x)}$ is negatively correlated with the orthogonality between f_i and f_j. When we have a known set of functions to quantify $\mathcal{F} = \{f_i\}_{i=1}^{L}$, to approximate an arbitrary function h^*, the quantization error can be expressed as the mean square error: $\xi \int_{\mathcal{D}} |h^*(x) - \sum_i \psi_i f_i(x)|^2 dx$, where ξ and ψ_i are the combination coefficient. According to the Parseval equality [40], if $\mathcal{F}$ is an orthogonal basis function set, the mean square error could reach 0. Furthermore, the orthogonality between the basis functions is more substantial; the mean square error is smaller, i.e., and the model corresponding to the linear combination of basis functions has a more robust representation capability. Here, we further introduce this insight to network quantization. The larger the bit, the greater the representational capability of the corresponding model [34]. Specifically, we propose to assign a larger bit width to the layer with stronger orthogonality against all other layers to maximize the representation capability of the model. However, Eq. 4.34 has the integral of a continuous function, which is untractable in practice. Therefore, we derive a novel metric to efficiently approximate the orthogonality of each layer in Sect. 4.5.3.2.

4.5.3.2 Efficient Orthogonality Metric

To avoid the intractable integral, we propose using Monte Carlo sampling to approximate the orthogonality of the layers. Specifically, from the Monte Carlo integration perspective in [5], Eq. 4.34 can be rewritten as:

$$\begin{aligned}\langle f_i, f_j\rangle_{P(x)} &= \int_{\mathcal{D}} f_i(x)P(x)f_j(x)^T dx \\ &= \left\| E_{P(x)}[f_j(x)^T f_i(x)] \right\|_F.\end{aligned} \tag{4.35}$$

We randomly obtain N samples $x_1, x_2, \ldots, x_N$ from a training dataset with the probability density matrix $P(x)$, which allows the expectation $E_{P(x)}[f_j(x)^T f_i(x)]$ to be further approximated as:

$$\begin{aligned}\left\| E_{P(x)}[f_j(x)^T f_i(x)] \right\|_F &\approx \frac{1}{N}\left\| \sum_{n=1}^{N} f_j(x_n)^T f_i(x_n) \right\|_F \\ &= \frac{1}{N}\left\| f_j(X)^T f_i(X) \right\|_F,\end{aligned} \tag{4.36}$$

where $f_i(X) \in \mathbb{R}^{N\times(C_i\times H_i\times W_i)}$ represents the output of the i-th layer, $f_j(X) \in \mathbb{R}^{N\times(C_j\times H_j\times W_j)}$ represents the output of the j-th layer, and $||\cdot||_F$ is the Frobenius norm. From Eqs. 4.35–4.36, we have:

$$N\int_{\mathcal{D}} f_i(x)P(x)f_j(x)^T dx \approx \left\| f_j(X)^T f_i(X) \right\|_F. \tag{4.37}$$

However, the comparison of orthogonality between different layers is difficult due to differences in dimensionality. To this end, we use the Cauchy-Schwarz inequality to normalize it to [0, 1] for the different layers. Applying the Cauchy-Schwarz inequality to the left side of Eq. 4.37, we have:

$$\begin{aligned} 0 &\leq \left(N\int_{\mathcal{D}} f_i(x)P(x)f_j(x)^T dx\right)^2 \\ &\leq \int_{\mathcal{D}} Nf_i(x)P_i(x)f_i(x)^T dx \int_{\mathcal{D}} Nf_j(x)P_j(x)f_j(x)^T dx. \end{aligned} \tag{4.38}$$

We substitute Eq. 4.37 into Eq. 4.38 and perform some simplifications to derive our **OR**thogonality **M**etric (ORM):[1]

$$\text{ORM}(X, f_i, f_j) = \frac{||f_j(X)^T f_i(X)||_F^2}{||f_i(X)^T f_i(X)||_F ||f_j(X)^T f_j(X)||_F}, \tag{4.39}$$

where ORM $\in [0, 1]$. f_i and f_j are orthogonal when ORM $= 0$. On the contrary, f_i and f_j depend on ORM $= 1$. Therefore, ORM is negatively correlated with orthogonality.

Calculation Acceleration Given a specific model, calculating Eq. 4.39 involves huge matrices. Suppose that $f_i(X) \in \mathbb{R}^{N\times(C_i\times H_i\times W_i)}$, $f_j(X) \in \mathbb{R}^{N\times(C_j\times H_j\times W_j)}$, and that the dimension of the features in the j-th layer is larger than that of the i-th layer. Furthermore, the time complexity of computing ORM(X, f_i, f_j) is $\boldsymbol{O}(NC_j^2H_j^2W_j^2)$. The huge matrix occupies a lot of memory resources and increases the entire algorithm's time complexity by several orders of magnitude. Therefore, we derive an equivalent form to accelerate the calculation. If we take $Y = f_i(X)$, $Z = f_j(X)$ as an example, then $YY^T, ZZ^T \in \mathbb{R}^{N\times N}$. We have the following:

$$||Z^TY||_F^2 = \left\langle \mathbf{vec}(YY^T), \mathbf{vec}(ZZ^T)\right\rangle, \tag{4.40}$$

[1] ORM is formally consistent with CKA. However, we pioneered discovering its relationship with quantized model accuracy. We confirmed its validity in mixed precision quantization from the perspective of function orthogonality, and CKA explores the relationship between hidden layers from the perspective of similarity. In other words, CKA implicitly verifies the validity of ORM further.

where **vec**$(\cdot)$ represents the operation of flattening matrix into vector. From Eq. 4.40, the time complexity of calculating ORM(X, f_i, f_j) becomes $\boldsymbol{O}(N^2 C_j H_j W_j)$ through the inner product of vectors. When the number of samples N is larger than the dimension of features $C \times H \times W$, the norm form is faster to calculate due to the lower time complexity and vice versa.

4.5.3.3 Mixed Precision Quantization

Effectiveness of ORM on Mixed Precision Quantization ORM directly indicates the importance of the layer in the network, which can eventually be used to decide the bit width configuration. We conducted extensive experiments to provide sufficient and reliable evidence for this claim. Specifically, we first sample different quantization configurations for ResNet-18 and MobileNetV2. We were then fine-tuning to obtain the performance. Meanwhile, the overall orthogonality of the sampled models is calculated separately. Interestingly, we find that the orthogonality and performance of the model are positively correlated with the sum of ORM in Fig. 4.10. Naturally, inspired by this finding, maximizing orthogonality is taken as our objective function, which is employed to integrate the model size constraints and construct a linear programming problem to obtain the final bit configuration.

For a specific neural network, we can calculate an orthogonality matrix K, where $k_{ij} = \text{ORM}(X, f_i, f_j)$. K is a symmetric matrix, and the diagonal elements are 1. We add the non-diagonal elements of each row of the matrix:

$$\gamma_i = \sum_{j=1}^{L} k_{ij} - 1. \tag{4.41}$$

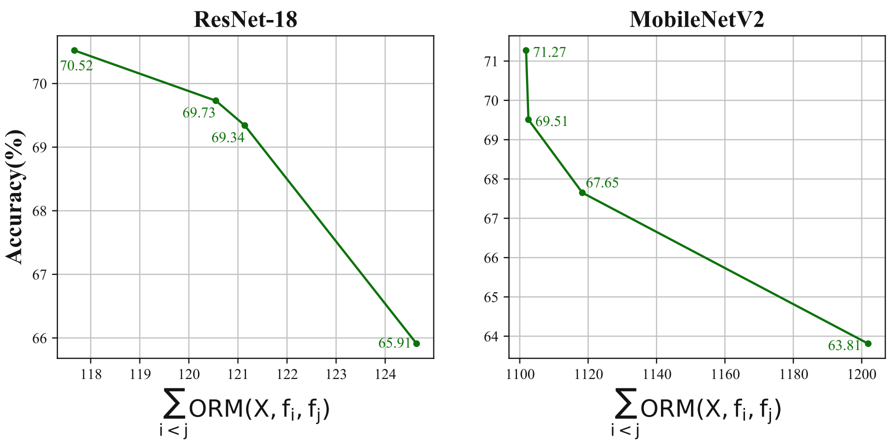


Fig. 4.10 Relationship between orthogonality and accuracy for different quantization configurations on ResNet-18 and MobileNetV2

The smaller γ_i means stronger orthogonality between f_i and other functions in the set of functions $\mathcal{F}$, and it also means that the former i layers of the neural network are more independent. Thus, we use the monotonically decreasing function e^{-x} to model this relationship:

$$\theta_i = e^{-\beta \gamma_i}, \tag{4.42}$$

where β is a hyperparameter to control the bit width difference between different layers; we also investigate the other monotonically decreasing functions (for details, refer to Sect. 4.5.3.5). θ_i is used as the important factor for the former i layers of the network, and then we define a linear programming problem as follows:

$$\begin{aligned} \text{Objective: } & \max_{\mathbf{b}} \sum_{i=1}^{L} \left(\frac{b_i}{L-i+1} \sum_{j=i}^{L} \theta_j \right), \\ \text{Constraints: } & \sum_{i}^{L} M^{(b_i)} \le \mathcal{T}. \end{aligned} \tag{4.43}$$

$M^{(b_i)}$ is the model size of the i-th layer under b_i bit quantization and $\mathcal{T}$ represents the target model size. $\mathbf{b}$ is the optimal bit configuration. Maximizing the objective function means assigning the larger bit width to a more independent layer, which implicitly maximizes the model's representation capability.

Solving the linear programming problem in Eq. 4.43 is highly efficient which only takes a few seconds on a single CPU. In other words, our method is highly efficient (9s on MobileNetV2) compared to previous methods [12, 31, 42], which require a lot of data or iterations to search. In addition, our algorithm can be combined as a plug-and-play module with quantization-aware training or post-training quantization schemes due to the high efficiency and low data requirements.

4.5.3.4 Experiment

The ImageNet dataset includes 1.2M training data and 50,000 validation data. We randomly obtain 64 training data samples for ResNet-18/50 and 32 training data samples for MobileNetV2 following similar data preprocessing [21] to derive the set of functions $\mathcal{F}$. For the models with many parameters, we directly adopt the round function to convert the bit width into an integer after linear programming. Meanwhile, we adopt a depth-first search (DFS) to find the bit configuration that strictly meets the different constraints for a small model, e.g. ResNet-18. The processes above are highly efficient and only take a few seconds on these devices. Additionally, OMPQ [35] is flexible and can leverage different search spaces with QAT and PTQ under different requirements. The fine-tuning implementation details are listed below.

For the experiments on the QAT quantization scheme, we use two NVIDIA Tesla V100 GPUs. Our quantization framework does not contain integer division or floating-point numbers in the network. In the training process, the initial learning rate is set to $1e-4$, and the batch size is set to 128. We use the cosine learning rate scheduler and the SGD optimizer with weight decay $1e-4$ during 90 epochs without distillation. Following the previous work, we fix the weight and activation values of the first and last layers at 8 bits, where the search space is 4–8 bits.

4.5.3.5 Ablation Study

Monotonically Decreasing Function We then investigate the monotonically decreasing function in Eq. 4.42. The second-order derivatives of monotonically decreasing functions in Eq. 4.42 influence the changing rate of orthogonality differences. In other words, the variance of the orthogonality between different layers becomes larger as the rate increases. We test the accuracy of five different monotonically decreasing functions on quantization-aware training of ResNet-18 (6.7Mb) and post-training quantization of MobileNetV2 (0.9Mb). We fixed the activation to 8 bit.

It can be seen from Table 4.5 that accuracy gradually decreases while the change rate increases. We also observe that a more significant change rate for the corresponding bit configuration means a more aggressive bit allocation strategy. In other words, OMPQ tends to assign more different bits between layers at a high rate of change, leading to worse performance in network quantization. Another interesting observation is the accuracy of ResNet-18 and MobileNetV2. Specifically, quantization-aware training on ResNet-18 requires numerous data, making the accuracy change insignificant. In contrast, post-training quantization on MobileNetV2 cannot assign bit configuration that meets the model constraints when the functions are set to $-x^3$ or $-e^x$. To this end, we select e^{-x} as our monotonically decreasing function in the following experiments.

Deconstruction Granularity We study the impact of different granularities of deconstruction on the model's accuracy. Specifically, we tested four different granularities, including layer-wise, block-wise, stage-wise, and net-wise, in the quantized-aware training of ResNet-18 and the post-training quantization of MobileNetV2.

Table 4.5 The Top-1 accuracy (%) with different monotonically decreasing functions on ResNet-18 and MobileNetV2

Decreasing Function	ResNet-18 (%)	MobileNetV2 (%)	Changing Rate
e^{-x}	72.30	**63.51**	e^{-x}
$-logx$	72.26	63.20	x^{-2}
$-x$	**72.36**	63.0	0
$-x^3$	71.71	–	$6x$
$-e^x$	–	–	e^x

Table 4.6 Top-1 accuracy (%) of different deconstruction granularity. The activation bit widths of MobileNetV2 and ResNet-18 are both 8. $*$ means a mixed bit

Model	W bit	Layer	Block	Stage	Net
ResNet-18	5$*$	72.51	**72.52**	72.47	72.31
MobileNetV2	3$*$	**69.37**	69.10	68.86	63.99

As reported in Table 4.6, the accuracy of the two models increases with finer granularities. This difference is more significant in MobileNetV2 due to the different sensitiveness between point-wise and depth-wise convolutions. Thus, we employ layer-wise granularity in the following experiments.

References

1. George B Arfken and Hans J Weber. Mathematical methods for physicists, 1999.
2. Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems*, 31, 2018.
3. Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Neural Information Processing Systems(NeurIPS)*, 32, 2019.
4. Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.
5. Russel E Caflisch. Monte carlo and quasi-monte carlo methods. *Acta numerica*, 7:1–49, 1998.
6. Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5918–5926, 2017.
7. Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. Neural gradients are near-lognormal: improved quantized and sparse training. *arXiv preprint arXiv:2006.08173*, 2020.
8. Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.
9. Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.
10. Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.
11. Lei Deng, Peng Jiao, Jing Pei, Zhenzhi Wu, and Guoqi Li. Gxnor-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework. *Neural Networks*, 100:49–58, 2018.
12. Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. In *Neural Information Processing Systems(NeurIPS)*, pages 18518–18529, 2020.
13. Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. Learned step size quantization. *ArXiv*, abs/1902.08153, 2019.
14. Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.

15. Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M Roy, and Ali Ramezani-Kebrya. Adaptive gradient quantization for data-parallel sgd. *Advances in neural information processing systems*, 33:3174–3185, 2020.
16. Joshua Fromm, Shwetak Patel, and Matthai Philipose. Heterogeneous bitwidth binarization in convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 4006–4015, 2018.
17. Jiaxin Gu, Ce Li, Baochang Zhang, Jungong Han, Xianbin Cao, Jianzhuang Liu, and David Doermann. Projection convolutional neural networks for 1-bit cnns via discrete back propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8344–8351, 2019.
18. Yiwen Guo, Anbang Yao, Hao Zhao, and Yurong Chen. Network sketching: Exploiting binary structure in deep cnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5955–5963, 2017.
19. Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pages 544–560. Springer, 2020.
20. Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.
21. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
22. Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
23. Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.
24. Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.
25. Tianchu Ji, Shraddhan Jain, Michael Ferdman, Peter Milder, H Andrew Schwartz, and Niranjan Balasubramanian. On the distribution, sparsity, and inference-time quantization of attention values in transformers. *arXiv preprint arXiv:2106.01335*, 2021.
26. Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Youngjun Kwak, Jae-Joon Han, and Changkyu Choi. Joint training of low-precision neural network with quantization interval parameters. *arXiv preprint arXiv:1808.05779*, 2, 2018.
27. Hyungjun Kim, Kyungsu Kim, Jinseok Kim, and Jae-Joon Kim. Binaryduo: Reducing gradient mismatch in binary activation network by coupling binary activations. In *International Conference on Learning Representations*.
28. Hyungjun Kim, Kyungsu Kim, Jinseok Kim, and Jae-Joon Kim. Binaryduo: Reducing gradient mismatch in binary activation network by coupling binary activations. *arXiv preprint arXiv:2002.06517*, 2020.
29. Yue Li, Wenrui Ding, Chunlei Liu, Baochang Zhang, and Guodong Guo. Trq: Ternary neural networks with residual quantization. In *AAAI Conference on Artificial Intelligence*, 2021.
30. Yuhang Li, Xin Dong, Sai Qian Zhang, Haoli Bai, Yuanpeng Chen, and Wei Wang. Rtn: Reparameterized ternary network. In *AAAI*, pages 4780–4787, 2020.
31. Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. {BRECQ}: Pushing the limit of post-training quantization by block reconstruction. In *International Conference on Learning Representations (ICLR)*, 2021.
32. Zefan Li, Bingbing Ni, Wenjun Zhang, Xiaokang Yang, and Wen Gao. Performance guaranteed network acceleration via high-order residual quantization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2584–2592, 2017.

33. Chunlei Liu, Wenrui Ding, Xin Xia, Baochang Zhang, Jiaxin Gu, Jianzhuang Liu, Rongrong Ji, and David Doermann. Circulant binary convolutional networks: Enhancing the performance of 1-bit dcnns with circulant back propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2691–2699, 2019.
34. Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018.
35. Yuexiao Ma, Taisong Jin, Xiawu Zheng, Yan Wang, Huixia Li, Guannan Jiang, Wei Zhang, and Rongrong Ji. Ompq: Orthogonal mixed precision quantization. *ArXiv*, abs/2109.07865, 2021.
36. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.
37. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, pages 4510–4520, 2018.
38. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
39. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
40. James Tanton. *Encyclopedia of Mathematics*. Facts on file, 2005.
41. Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Computer Vision and Pattern Recognition (CVPR)*, pages 8612–8620, 2019.
42. Linjie Yang and Qing Jin. Fracbits: Mixed precision quantization via fractional bit-widths. *AAAI Conference on Artificial Intelligence (AAAI)*, 35:10612–10620, 2021.
43. Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.
44. Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *arXiv preprint arXiv:1707.01083*, 2017.
45. Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Chapter 5
Network Pruning

5.1 Introduction

Network pruning is a technique used in deep learning to reduce the size and complexity of neural networks by eliminating unnecessary connections or parameters. Pruning aims to create more efficient and streamlined models that maintain or improve performance while reducing computational requirements and memory footprint.

Pruning involves identifying and removing unimportant connections or parameters from the neural network. Importance criteria are used to determine the significance of each connection or parameter. Some standard criteria include:

Magnitude-based criteria Connections or parameters with small magnitudes are considered less important and are pruned. This can be done by setting a threshold below which connections are removed or keeping the top-k connections with the highest magnitudes.

Sensitivity-based criteria Connections or parameters that have the most negligible impact on the model's performance are pruned. This is typically determined by calculating the gradients or sensitivities of the output concerning each connection or parameter.

Once the important criteria are defined, pruning methods remove the connections or parameters identified as unimportant. Pruning can be categorized into different techniques:

Weight pruning Weight pruning eliminates individual connections or parameters based on their importance. This can result in a sparse model where some connections have zero values.

Structured pruning Structured pruning removes entire filters, channels, or layers instead of individual connections. This approach can lead to more efficient and

B. Zhang et al., *Neural Networks with Model Compression*,
Computational Intelligence Methods and Applications,
https://doi.org/10.1007/978-981-99-5068-3_5

regular network structures and may require specialized hardware support for practical inference.

Unit pruning Unit pruning involves removing entire neurons or units from the network. This can be based on their importance or by analyzing their impact on the network's performance.

Network pruning is often performed iteratively to achieve higher levels of sparsity (the proportion of pruned connections in the network). The process involves multiple pruning iterations, where the least essential connections are successively removed. Pruning can be applied layer-wise or globally across the entire network.

Pruning can be integrated into the training process itself. This approach, known as "structured sparsity training" or "learning with sparsity," involves introducing regularization techniques that naturally encourage the network to sparsify during training. This leads to a more efficient model, eliminating the need for post-training pruning.

One of the primary advantages of network pruning is the reduction in model size. Pruning creates a more compact model by eliminating unnecessary connections and parameters from the neural network. This size reduction has practical implications, especially in limited storage or memory capacity. Smaller models are easier to store, transmit, and deploy, making them more feasible for real-world applications.

Pruning also leads to improved computational efficiency. By removing unimportant connections, the computational workload during inference is reduced. This results in faster inference times, which is crucial for real-time applications or situations requiring low-latency responses. The streamlined model can also leverage efficient sparse matrix operations and specialized hardware accelerators, further enhancing computational efficiency.

Another benefit of network pruning is the reduction in memory footprint. The memory requirements for storing and processing the model are reduced by pruning sparse connections. This is particularly advantageous in resource-constrained environments like mobile devices or embedded systems. It allows for the efficient execution of deep learning models in memory-limited scenarios.

Network pruning facilitates the compression and deployment of models. The reduced size and complexity of pruned models make them easier to compress and deploy. They require less storage space and bandwidth, making them suitable for scenarios with limited resources or low-bandwidth networks. Pruned models can be efficiently deployed on edge devices, IoT devices, or cloud platforms, enabling the scalable and resource-efficient deployment of deep learning models.

It's important to note that the degree of pruning should be carefully balanced to achieve the desired benefits without significantly sacrificing model performance. Aggressive pruning may lead to accuracy degradation, so a trade-off between model size reduction and performance should be considered during the pruning process.

5.2 Structured Pruning

Structured pruning reduces model size and complexity by pruning entire structured components, such as channels, layers, or blocks, rather than individual weights or filters. Structured pruning methods aim to maintain the inherent structure of the network while achieving model compression. By removing entire structured components, rather than randomly selecting individual parameters, structured pruning preserves the architectural characteristics of the network. This allows for more efficient hardware acceleration, reduced memory footprint, and simplified model deployment.

There are different types of structured pruning techniques commonly used:

Channel Pruning Channel pruning involves removing entire channels or feature maps from convolutional layers. Channels represent specific patterns or feature detectors learned by the network. Channel pruning reduces the model's complexity and computational requirements by eliminating redundant or less important channels.

Layer Pruning Layer pruning focuses on pruning entire layers from the network architecture. Less critical or contributing layers are identified and removed, resulting in a shallower model. This reduces the number of parameters and simplifies the overall structure of the network.

Block Pruning Block pruning targets the removal of entire blocks or modules within the network. Blocks often represent repeated structures or groups of layers. The network's complexity is reduced by pruning these blocks while maintaining the essential architectural characteristics. This can be particularly effective in deep networks with complex architectures.

Structured Sparsity Structured sparsity techniques enforce structured sparsity patterns within the network. Instead of pruning individual weights or filters, specific structured patterns or structures within the network must be sparse. This can involve enforcing sparsity in specific rows, columns, or other structured patterns of weight matrices.

The benefits of structured pruning are numerous. Structured pruning enables efficient hardware acceleration by leveraging specialized hardware accelerators to exploit structured sparsity patterns. This results in faster inference times and improved energy efficiency. Secondly, it significantly reduces the memory footprint of neural networks, making them more suitable for deployment on memory-constrained devices or in scenarios with limited storage capacity. Structured pruning simplifies model deployment by reducing complexity and facilitating the model transfer, compression, and integration into production systems.

5.3 Unstructured Pruning

Unstructured pruning is a technique used in deep learning to reduce the size and complexity of neural networks by selectively removing individual weights or filters without considering their structural relationships. Unlike structured pruning, which prunes entire structured components, unstructured pruning focuses on the fine-grained elimination of individual parameters based on their importance or redundancy. This approach offers flexibility in achieving model compression but may result in irregular and unstructured sparsity patterns.

Unstructured pruning offers flexibility in achieving model compression since individual parameters can be selectively pruned. This allows for fine-grained control over the sparsity level and significantly reduces model size. Targeting and removing redundant or less important weights can also achieve high compression rates. The model's size and memory requirements can be significantly reduced by eliminating these parameters.

Unstructured pruning does not disrupt the structural integrity of the network since individual weights or filters are pruned independently. This means the model's architecture remains unchanged, allowing for easier integration and transfer of pruned models.

However, the irregular and unstructured sparsity patterns introduced by this technique may need to be more efficiently utilized by hardware accelerators designed for structured sparsity. Additionally, unstructured pruning may require careful fine-tuning to recover performance, as removing individual parameters can lead to a more significant performance drop than structured pruning.

5.4 Network Pruning

5.4.1 Efficient Structured Pruning Based on Deep Feature Stabilization

Conventional filter pruning methods generally rely on the important criteria such as the ℓ_1-norm value [40] and ℓ_2-norm value [21] of filters. Two central problems may lie in the existing methods. Firstly, the important criterion can only be employed on filter selection, i.e., a block cannot be evaluated by criteria such as norm criterion. Secondly, the importance of each filter seems too simple and inefficient due to the existence of batch normalization (BN) [30] and nonlinear activation functions, e.g., rectifier linear unit (ReLU) [13]. To overcome these two shortcomings, the reconstruction-based method is introduced. He et al. [23] propose a channel pruning method based on the local reconstruction error of every block and optimized the reconstruction loss via most minor absolute shrinkage and selection operator (LASSO) [59] regression. Likewise, accelerated proximal gradient (APG) [73] and Taylor expansion [42] are employed to optimize the

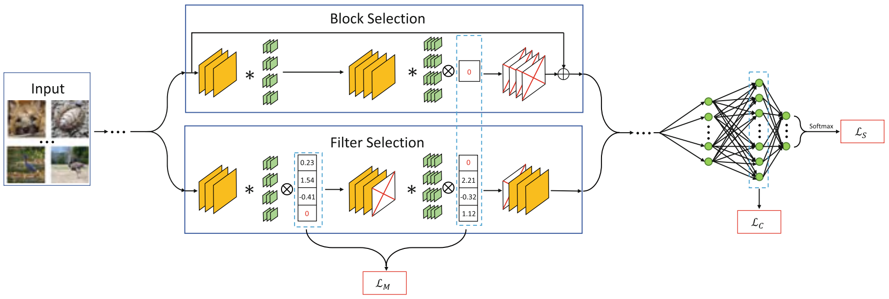


Fig. 5.1 An illustration of EPFS. From left to right lies the training process. The yellow and green square sets denote feature maps and filters, respectively. As represented in the middle, our EPFS can be effectively implemented in block or filter selection by setting the specific full-precision soft masks after specific layers. The soft mask and other parameters will be updated via FISTA and SGD, respectively. When the mask element is zero, the corresponding filter or block is equivalent to being pruned. $\mathcal{L}_M$, $\mathcal{L}_C$ and $\mathcal{L}_S$ are employed to supervise mask sparsity, deep feature stabilization, and network output, respectively

reconstruction loss. However, the small reconstruction error might be magnified and propagated in the deep networks, leading to large reconstruction errors in the global outputs.

We propose an end-to-end efficient pruning method based on feature stability (EPFS) [68]. The framework of EPFS is shown in Fig. 5.1. For block pruning, we introduce a mask on the output of the layers and use the sparsity supervision, i.e., ℓ_1-norm, to supervise the updating of the mask. We introduce a novel ℓ_2-regularization term for filter pruning to supervise mask updating. The sparsity supervision and cross-entropy make up a couple of adversaries between sparsity and accuracy. The center loss [64] is employed to further stabilize the deep feature during learning. However, using conventional stochastic gradient descent (SGD) to optimize the mask tends to obtain lower performance. Thus, we introduce a fast iterative shrinkage-thresholding algorithm (FISTA) [3, 14] to optimize the learning process, achieving a faster and more reliable pruning process of the mask.

5.4.1.1 Preliminaries

Consider a CNN model consisting of L layers (convolutional and fully connected layers) interlaced with rectifier linear units (ReLU) and pooling. We can formulate the convolutional layer's output size as $K^l \times W^l \times C^l$. Hence, we define a convolution-batch normalization (Conv-BN) operation transforming the input tensor

$\boldsymbol{x}^{l-1} \in \mathbb{R}^{K^{l-1} \times W^{l-1} \times C^{l-1}}$ to the output tensor $\boldsymbol{x}^{l} \in \mathbb{R}^{K^{l} \times W^{l} \times C^{l}}$ as:

$$\begin{aligned} x_j^l &= f^l(\boldsymbol{x}^{l-1}, \boldsymbol{F}^l, \boldsymbol{\pi}^l, \boldsymbol{\beta}^l, \boldsymbol{\tau}^l, \boldsymbol{\gamma}^l) \\ &= \gamma_j^l \frac{\sum_{i=1}^{C^{l-1}} x_i^{l-1} * F_{i,j}^l - \pi_j^l}{\tau_j^l} + \beta_j^l, \end{aligned} \tag{5.1}$$

where C^l represents the number of channels in the l-th layer. x_j^l and x_i^{l-1} are the j-th output feature map and the i-th input feature map at the l-th layer. $*$ denotes the convolutional operation. $(\pi_j^l, \tau_j^l, \beta_j^l, \gamma_j^l)$ are the corresponding BN parameters of the j-th channel. $F_{i,j}^l$ is the i-th kernel of the j-th filter of the l-th layer.

5.4.1.2 Sparse Supervision for Block Pruning

As shown in Fig. 5.1, different soft masks should be deployed for different pruning tasks. We will state this subsection in three parts, i.e., mask setups for block and filter pruning, respectively, and loss formulation.

We first modify the denotation in Eq. 5.1 block-wise for block selection. As plotted in Fig. 5.1, we introduce a scalar m^k for selecting the k-th block. Considering the k-th block containing k_N layers, we formulate it as:

$$\begin{aligned} \boldsymbol{x}^k &= m^k \cdot \mathcal{F}^k(\boldsymbol{x}^{k-1}, \boldsymbol{F}^k, \boldsymbol{\pi}^k, \boldsymbol{\beta}^k, \boldsymbol{\tau}^k, \boldsymbol{\gamma}^k) + \mathcal{S}(\boldsymbol{x}^{k-1}) \\ &= m^k \cdot f^{k_1} \circ f^{k_2} \circ f^{k_3} \circ \cdots f^{k_N} + \mathcal{S}(\boldsymbol{x}^{k-1}) \end{aligned} \tag{5.2}$$

where $f^1 \circ f^2 = f^2(f^1)$. f^{k_i} denotes the k_n-th layer in the k-th block in sequence, i.e., $f^{k_n}(x^{k_n-1}, \boldsymbol{F}^{k_n}, \boldsymbol{\pi}^{k_n}, \boldsymbol{\beta}^{k_n}, \boldsymbol{\tau}^{k_n}, \boldsymbol{\gamma}^{k_n})$. $\mathcal{S}()$ denotes the shortcut transformation. We introduce a learnable mask $\boldsymbol{m} = [m^1, \cdots, m^L]$ to scale the output. To guarantee the input of the following blocks is not 0, we only implement the mask to the blocks having residual connections, rather than blocks in generalized meaning such as [26, 54]. For the convergence of mask, we use ℓ_1-regularization to punish $\boldsymbol{m}$ to optimize the elements to 0. Therefore, $\boldsymbol{m}$ is learned by:

$$\mathcal{L}_M = \mu \sum_{k=1}^{K} |m^k|, \tag{5.3}$$

where m^k represents the mask scalar for every block.

5.4.1.3 Constrained Sparse Supervision for Filter Pruning

For filter pruning, we introduce a vector $\boldsymbol{m^l}$ sized $1 \times 1 \times C^l$ to scale the output of l-th Conv-BN layer. We formulate the filter selection process as:

$$\begin{aligned} x_j^l &= f^l(\boldsymbol{m}^{l-1} \otimes \boldsymbol{x}^{l-1}, F_j^l, \pi_j^l, \beta_j^l, \tau_j^l, \gamma_j^l) \\ &= \gamma_j^l \frac{\sum_{i=1}^{C^{l-1}} (m_i^{l-1} \cdot x_i^{l-1}) * F_{i,j}^l - \pi_j^l}{\tau_j^l} + \beta_j^l, \end{aligned} \tag{5.4}$$

where $\otimes$ denotes element-wise multiplication. Equation 5.4 denotes scaling the output of $(l-1)$-th Conv-BN layer by soft mask $\boldsymbol{m^{l-1}}$.

Unlike the prior work [23, 43], we introduce a novel constraint for sparse supervision deployed on filter pruning as:

$$\mathcal{L}_M = \mu \sum_{l=1}^{L} \|\boldsymbol{m}^l\|_1 + \alpha \sum_{l=1}^{L} \sum_{j=1}^{C^l} \|m_j^l - \|F_j^l\|_2\|_2^2, \tag{5.5}$$

where we introduce a strong constraint to guarantee the safe convergence of $\boldsymbol{m}$. As we observe that the ℓ_1-regularization may cause damage to the architecture, we introduce the ℓ_2-regularization to consider the magnitude of filters. We only zero m_j^l when the filter magnitude $\|F_j^l\|_2$ is close to zero. In Eq. 5.5, $\mathcal{L}_M$ represents the loss function to update the mask. K and L denote the network's total blocks and layers. μ and α are hyperparameters to control the proportion of sparsity and constraint. Note that the constraint $\|m_j^l - \|F_j^l\|_2\|_2^2$ is only employed in updating $\boldsymbol{m}$.

5.4.1.4 Loss Function

Cross-entropy is still employed in the learning process to improve image classification accuracy. It is formulated as:

$$\mathcal{L}_S = -\sum_{q=1}^{Q} \log \frac{e^{W_{y_q}^L \boldsymbol{x}_{(q)}^{L-1} + b_{y_q}^L}}{\sum_{p=1}^{P} e^{W_p^L \boldsymbol{x}_{(q)}^{L-1} + b_p^L}}, \tag{5.6}$$

where $\boldsymbol{x}_{(q)}^{L-1}$ denotes the deep feature of the q-th input in mini-batch. Q denotes the mini-batch size. For $q \in Q$, y_q denotes the network's forecast of the y_q-th class in P.

Furthermore, we introduce center loss [64] to maintain the feature stabilization of the pruned networks. The center loss function is defined as:

$$\mathcal{L}_C = \frac{\lambda}{2} \sum_{q=1}^{Q} \|\boldsymbol{x}_{(q)}^{L-1} - c_{y_q}\|_2^2. \tag{5.7}$$

In Eq. 5.7, $x_{(q)}^{L-1}$ is the deep feature of q-th input image in batch. $g(x) \in \mathbb{R}^{C^{L-1} \times C^L}$. C^{L-1} and C^L are the channels of input and output, respectively. c_{y_q} is the feature center of the ground truth label of q-th input, which denotes the feature center of every class. And λ is the hyperparameter for balancing the proportion of center loss and two others. SGD updates c. We will give a more detailed description in the next subsection.

We use joint supervision through cross-entropy and center loss [64] to achieve discriminative feature learning. The formulation is given as:

$$\mathcal{L} = \mathcal{L}_S + \mathcal{L}_C + \mathcal{L}_M \tag{5.8}$$

$$= -\sum_{q=1}^{Q} \log \frac{e^{W_{y_q}^L \boldsymbol{x}^{\boldsymbol{L-1}} + b_{y_q}^L}}{\sum_{p=1}^{P} e^{W_p^L \boldsymbol{x}^{\boldsymbol{L-1}} + b_p^L}} + \frac{\lambda}{2} \sum_{q=1}^{Q} \|\boldsymbol{x}_{(q)}^{L-1} - c_{y_q}\|_2^2 + \mathcal{L}_M. \tag{5.9}$$

The joint loss well supervises the output and deep feature.

5.4.1.5 Optimization

SGD can be directly introduced to update the feature center c_p and model parameters W to solve the optimization problem in Eq. 5.9. We update W and c_p as

$$c_p^{t+1} = c^t - \beta \delta_{c_p} \tag{5.10}$$

$$W^{t+1} = c^t - \eta \delta_W \tag{5.11}$$

$$\delta_{c_p} = \frac{\sum_{p=1}^{P} \mathcal{I}(y_q = p) \cdot (c_p - \boldsymbol{x}^{L-1})}{1 + \sum_{p=1}^{P} \mathcal{I}(y_q = p)} \tag{5.12}$$

$$\begin{aligned}\delta_W &= \frac{\partial \mathcal{L}_S}{\partial W} + \lambda \frac{\partial \mathcal{L}_C}{\partial W} \\ &= \frac{\partial \mathcal{L}_S}{\partial W} + \lambda \frac{\partial \mathcal{L}_C}{\partial \boldsymbol{x}^{L-1}} \cdot \frac{\partial \boldsymbol{x}^{L-1}}{\partial W} \\ &= \frac{\partial \mathcal{L}_S}{\partial W} + \lambda \frac{\partial \boldsymbol{x}^{L-1}}{\partial W} \cdot (\boldsymbol{x}^{L-1} - c_{y_q})\end{aligned} \tag{5.13}$$

In Eq. 5.12, $\mathcal{I}(y_q = p)$ is the indicative function, defined as:

$$\mathcal{I}(y_q = p) = \begin{cases} 0, & y_q \neq p \\ 1, & y_q = p \end{cases} \tag{5.14}$$

Then every simple weight in convolution or BN layer can be updated as Eq. 5.13. However, it is unreliable to implement SGD to solve the optimization problem in Eq. 5.3. Because the SGD may bring the vibration of $\mathcal{L}_M$, affecting the convergence of the mask learning. The misleading mask may remove the necessary structures and decrease the accuracy. Thus, a threshold is required to limit the vibration. Under this guiding ideology, we introduce FISTA [14] to optimize $\mathcal{L}_M$.

We first use SGD to optimize the $\mathcal{W}_p$ and c. The whole procedure relies on the original forward-backward pass. Then, we introduce FISTA to solve $\boldsymbol{m}$ as:

$$\arg\min_{\boldsymbol{m}} \mathcal{L}_M + \mathcal{R}(\boldsymbol{m}) \tag{5.15}$$

The mask $\boldsymbol{m}$ can be updated by FISTA with the initial $\tau^1 = 1$:

$$\begin{aligned}\tau^1 &= 1 \\ \tau^{t+1} &= \frac{1 + \sqrt{1 + 4[\tau^{t+1}]^2}}{2} && (5.16) \\ \boldsymbol{y}^{t+1} &= \boldsymbol{m}^t + \frac{\tau^t - 1}{\tau^{t+1}}[\boldsymbol{m}^t - \boldsymbol{m}^{t-1}] && (5.17) \\ \boldsymbol{w}^{t+1} &= \boldsymbol{y}^{t+1} - \eta^{t+1} \cdot \frac{\partial \mathcal{G}[\boldsymbol{m}^{t+1}]}{\partial \boldsymbol{m}^{t+1}} && (5.18) \\ \boldsymbol{m}^{t+1} &= \text{sign}(\boldsymbol{w}^{t+1}) \circ \max\left\{|\boldsymbol{w}^{t+1}| - \eta^{t+1} \cdot \mu\right\}. && (5.19)\end{aligned}$$

η is an iterative learning rate. The whole optimization procedure is shown in Algorithm 8.

Algorithm 8: The updating algorithm of EPFS

```
   Input: Training data $\boldsymbol{x}^0 = \{x^0_{(1)}, \dots, x^0_{(Q)}\}$ with a mini-batch size $Q$, mask factor $\mu$, number
          of steps $t$, maximum iterations $T$, mask step iteration number $S$.
   Output: The weights $W$ and mask $\boldsymbol{m}$
1  Initialize $W, \boldsymbol{m} \sim \mathcal{N}(0, 1)$ and $k = 1$
2  repeat
3      For $t$ steps do
4          Forward pass masked network in a sample mini-batch of $Q$ examples get
   $\left\{\boldsymbol{x}^{L-1}_{(1)}, \dots, \boldsymbol{x}^{L-1}_{(Q)}\right\}$ and $\left\{\boldsymbol{x}^{L}_{(1)}, \dots, \boldsymbol{x}^{L}_{(Q)}\right\}$.
5          Update the $\mathcal{W}_p$ by Eqs. 5.10 and 5.12
6          Update the each $c_j$ by Eqs. 5.11 and 5.13
7          If $t == S$, then it is the mask step:
8              Update the $\boldsymbol{m}$ with FISTA from Eqs. 5.15 to 5.19.
9      end for
10 until convergence or $t$ reaches the maximum iterations $T$
```

5.4.1.6 Pruning on ResNet

Unlike VGGNets [55] or AlexNets [35], in ResNets, each residual block contains two or three convolutional layers (followed by both BN and ReLU) and shortcut connections. For the consistency of two parts for the sum operation, the number of output feature maps in the last convolutional layer must be consistent with that of the projection shortcut layer. In particular, when the dimensions of input/output channels are mismatched in a residual block, the shortcut connections perform a linear projection.

This work implements our EPFS method on ResNet-18 (two convolutional layers in each block) for filter pruning. We focus on pruning the first layer in each residual block, as illustrated in Fig. 5.2c. And no pruning operation is conducted in the last convolutional layer of each residual block. About $\frac{128-76}{128} = 40.2\%$ parameters are

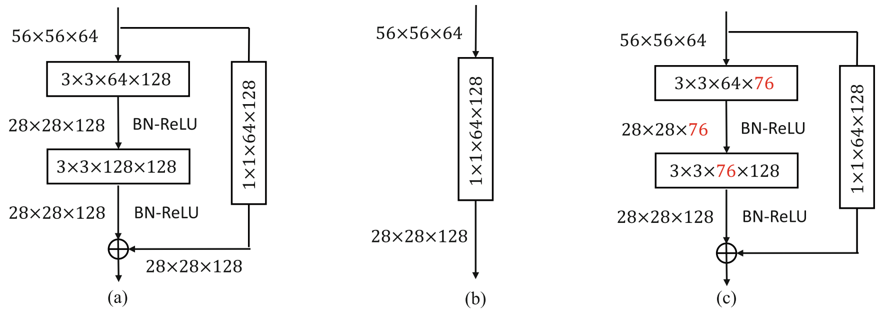


Fig. 5.2 Illustration of pruning ResNet-18. The red value is the number of remaining filters/channels. (**a**) Original block of ResNet-18. (**b**) Block pruning for ResNet-18. (**c**) Filter pruning for ResNet-18

pruned as illustrated in Fig. 5.2c, which make up a large proportion in the residual block.

For block pruning, the parameters of shortcut connection are far less than those of residual mapping. And the shortcut provides the chance to scale the mapping out to 0 without breaking the shape of the output. So, the strategy to set masks is clear. We add the mask before the comprehensive point of mapping and shortcut. The pruned ResNet-18 can be viewed as Fig. 5.2b.

5.4.1.7 Experiments

We implement extensive experiments to validate the effectiveness of our EPFS on filter pruning and block pruning. For filter pruning, we use six convolutional networks on two datasets, i.e., ResNet-20, ResNet-56 [21], VGGNet [56], and MobileNetV3-Large/Small [26] on CIFAR-10 [34] and ResNet-18 [19] on ImageNet ILSVRC2012 [35]. For block pruning, we also use six convolutional networks on two datasets, i.e., ResNet-20, ResNet-56, MobileNetV3-Large/Small on CIFAR-10, ResNet-18, and MobileNetV3-Small on ImageNet ILSVRC2012. Furthermore, we implement the comprehensive pruning (pruning block and filter sequentially), using three models on two datasets, i.e., ResNet-20, ResNet-56 on CIFAR-10, and ResNet-18 on ImageNet ILSVRC2012.

We use PyTorch [51] to implement our EPFS method. We use four NVIDIA GTX 2080 Ti GPUs with 128 GB of RAM to compute the above learning process. The hyperparameter μ is selected in the [0.001, 1] range after cross-validation. And λ is set to 0.0003. The weight decay is set to 0.0001, and the momentum is 0.9. The initial learning rate of $\mathcal{L}_S$, $\mathcal{L}_M$, i.e., η is set as 0.1. And it will be scaled by a factor of 0.1 every 20 epochs. The learning rate of feature center, i.e., β, is set as 0.5. Total epochs are 60.

We evaluate the performance of EPFS on CIFAR-10 for five networks, ResNet-20/ResNet-56, MobileNetV3-Large/MobileNetV3-Small, and VGGNet. For block pruning, ResNet-20 and ResNet-56 have 9 and 27 blocks for pruning, respectively. In MobileNetV3, there are no shortcuts in the downsampling blocks. Therefore, MobileNetV3-Large has 15 blocks, in which only 10 have residual connections that can be pruned. Likewise, MobileNetV3-Small has 7 of 11 blocks that can be pruned. For filter pruning, we implement it on ResNet-20/ResNet-56 and VGGNet. Moreover, we implement the comprehensive pruning on ResNet-20/ResNet-56 and compare its performance with the ones using block/filter pruning alone. *P.S.*, we use EPFS-B/F-μ denoting implementing block/filter pruning via EPFS with special hyperparameter μ. EPFS-C-μ_1-μ_2 denotes the comprehensive pruning, i.e., block and filter pruning sequentially. μ_1 and μ_2 denote the hyperparameters used for block and filter pruning, respectively. α is set to 1×10^{-4}.

ResNet-20 To evaluate the effectiveness of our method, we prune ResNet-20, where the mask's effectiveness can be examined subtly. The pruning results are shown in Table 5.1. For ResNet-20, when μ is set to 0.6, *three* out of *nine*

Table 5.1 Filter/block/comprehensive pruning for ResNet-20 on CIFAR-10

Model	Top-1/+FT%	Params(PR%)	FLOPs(PR%)	Speedup
ResNet-20 [19]	92.17	0.27M	40.55M	–
MIL [9]	–/91.43	–	32.31M(20.3%)	1.26×
SFP [21]	90.83/–	–	23.44M(42.2%)	1.73×
EPFS-B-0.6	91.51/91.91	0.20M(24.6%)	30.83M(24.0%)	1.32×
EPFS-B-0.8	91.31/91.50	0.17M(36.9%)	22.72M(44.0%)	1.79×
EPFS-F-0.05	90.20/90.83	0.14M(51.1%)	20.84M(48.6%)	1.94×
EPFS-C-0.6-0.05	90.01/90.98	0.12M(56.0%)	18.98M(53.2%)	2.14×

Table 5.2 Filter/block/comprehensive pruning for ResNet-56 on CIFAR-10

Model	Top-1/+FT%	Params(PR%)	FLOPs(PR%)	Speedup
ResNet-56 [19]	93.26	0.85M	125.49M	–
PFEC [40]	–/93.06	–	90.9M(27.6%)	1.21×
CP [23]	90.80/91.80	–	62M(50.6%)	2.02×
NISP [71]	–/93.01	–	81M(35.5%)	1.55×
EPFS-B-0.6	91.16/92.89	0.61M(27.7%)	75.91M(39.5%)	1.65×
EPFS-B-0.8	90.91/92.34	0.35M(58.6%)	65.32M(47.9%)	1.92×
EPFS-F-0.01	92.10/92.96	0.68M(20.0%)	89.60M(28.6%)	1.40×
EPFS-F-0.05	90.92/92.09	0.34M(40.1%)	64.50M(44.7%)	1.81×
EPFS-C-0.6-0.05	91.71/92.53	0.28M(67.1%)	56.47M(55.0%)	2.22×

residual blocks are pruned with 24.0% FLOPs pruned rate, and we only have 0.26% accuracy decrease. This indicates that there are redundant blocks for ResNet-20. Compared with SFP [21], our method achieves a much better performance. As for filter pruning, when μ is set to 0.05, 48.6% FLOPs are removed with 1.34% absolute accuracy drop. Moreover, the comprehensive pruning can remove 53.2% FLOPs with only a 1.19% performance decrease, demonstrating that we have achieved a new state-of-the-art result.

ResNet-56 For ResNet-56, the pruning results are shown in Table 5.2. For block pruning, when μ is set to 0.6, 11 out of 27 residual blocks are pruned, thus realizing 39.5% FLOPs pruned rate with a decrease of 0.37% accuracy. Compared with CP [23], PFEC [40], and NISP [71], our method achieves a much better trade-off between accuracy and compression rate. As for filter pruning, when μ is set to 0.01, 28.6% FLOPs are removed with 0.30% absolute accuracy drop. Moreover, comprehensive pruning can lead to a 55.0% pruning rate with only a 0.73% performance decrease. It gains a higher accuracy and pruning rate than EPFS-B-0.8 and EPFS-F-0.05, demonstrating the structured redundancy accounting for a large proportion in both width and depth aspects.

MobileNetV3-Large/MobileNetV3-Small MobileNetV3 is the state-of-the-art model. It was obtained by neural architecture search (NAS) [77]. We implement the block pruning for MobileNetV3 to validate our method's effectiveness and

Table 5.3 Block pruning for MobileNetV3-Large/MobileNetV3-Small on CIFAR-10

Model	Top-1/+FT%	Params(PR%)	FLOPs(PR%)	Speedup
MobileNetV3-Large [26]	94.05	4.18M	227.74M	–
EPFS-B-0.2	93.71/93.81	3.41M(18.4%)	179.50M(21.2%)	1.26×
EPFS-B-0.6	93.83/94.07	2.95M(29.4%)	140.03M(38.5%)	1.62×
EPFS-B-0.8	93.22/93.32	3.01M(27.9%)	146.97M(35.5%)	1.55×
MobileNetV3-Small [26]	93.14	2.93M	66.17M	–
EPFS-B-0.2	92.81/92.90	2.79M(4.64%)	45.00M(32.0%)	1.47×
EPFS-B-0.6	92.79/92.82	2.55M(12.9%)	38.18M(42.3%)	1.73×
EPFS-B-0.8	90.99/91.13	2.55M(27.9%)	37.51M(43.3%)	1.77×

Table 5.4 Filter pruning for VGGNet on CIFAR-10

Model	Top-1/+FT%	Params(PR%)	FLOPs(PR%)	Speedup
VGGNet [55]	93.50	14.98M	313.73M	–
PFEC [40]	93.40	5.40(64.0%)	206.00M(34.3%)	1.21×
EPFS-F-0.001	91.37/93.61	6.49M(56.7%)	200.2M(36.2%)	1.57×
EPFS-F-0.005	92.57/94.67	4.41M(69.1%)	156.87M(47.5%)	1.90×

generalization ability. Results are shown in Table 5.3. For MobileNetV3-Large, when μ is set to 0.6, 4 out of 11 blocks are pruned with 38.5% FLOPs pruned rate, and we achieve a 0.02% accuracy increase. For MobileNetV3-Small, when μ is set to 0.6, two out of $seven$ blocks are pruned with 42.3% FLOPs pruned rate, and we achieve only 0.28% accuracy decrease. Finally, the pruned model has about 93% accuracy and only 36.91M FLOPs on CIFAR-10, resulting in new state-of-the-art results. The experiments on MobileNetV3 demonstrate that the searched architectures by NAS still have redundancy, while our EPFS can efficiently reduce it.

VGGNet For VGGNet, there are no blocks. Hence, we deploy filter pruning via EPFS on VGGNet, whose results are shown in Table 5.4. We set the hyperparameter μ in [0.001, 0.01], for a small μ that is needed to balance the $\mathcal{L}_M$ produced by 4736 channels in VGGNet. When μ is set to 0.005, 2022 out of 4736 channels are pruned, thus removing 47.5% FLOPs. Compared with PFEC [40], our method achieves a much better accuracy and compression rate performance (Fig. 5.3).

We further evaluate the performance of EPFS on large-scale ImageNet and ILSVRC2012 in two networks, ResNet-18 and MobileNetV3-Small. ResNet-18 and MobileNetV3-Small have $eight$ and six blocks to prune for the block pruning, respectively. We then implement filter pruning on ResNet-18. Moreover, we implement the comprehensive pruning on ResNet-18 and compare its performance with the ones using block/filter pruning alone. And α is set to 1×10^{-4}.

ResNet-18 We also evaluated our method on ImageNet using ResNet-18. We train the pruned network with a mini-batch size of 128 for 60 epochs. As shown in Table 5.5, our method can obtain 1.41× and 1.86× speedup by setting μ to 0.2

Fig. 5.3 ImageNet ILSVRC2012 dataset

Table 5.5 Filter/block/comprehensive pruning for ResNet-18 on ImageNet ILSVRC2012

Model	Top-1/+FT%	Top-5/+FT%	Params (PR%)	FLOPs (PR%)	Speed Up
ResNet-18 [19]	69.75	89.24	10.67M	1.81B	–
MIL [9]	–/66.33	–/86.94	–	1.18B(34.6%)	1.54×
SFP [21]	–/67.10	–/87.78	–	1.05B(41.8%)	1.72×
FPGM [22]	67.78/68.34	88.01/88.53	–	1.05B(41.8%)	1.72×
EPFS-B-0.2	67.91/68.21	87.80/88.20	8.13M(23.8%)	1.28B(29.3%)	1.41×
EPFS-B-0.6	66.79/67.53	86.91/87.83	7.12M(33.3%)	0.98B(46.0%)	1.86×
EPFS-F-0.05	67.21/67.81	87.12/88.37	6.98M(34.6%)	1.05B(42.1%)	1.72×
EPFS-C-0.6-0.05	67.41/68.12	87.30/88.29	5.65M(47.0%)	0.81B(55.2%)	2.23×

and 0.5 for block pruning, with the decrease of 1.56%/1.04% and 2.23%/1.41% in Top-1/Top-5 accuracy, respectively. For filter pruning, our EPFS obtained the 67.81% Top-1 accuracy with 34.6% FLOPs removed. Furthermore, we implement comprehensive pruning for ResNet-18 on ImageNet ILSVRC2012. We set the hyperparameter μ as 0.6 and 0.05 for block and filter pruning, respectively. Our EPFS can obtain 68.12% Top-1 and 88.29% Top-5 accuracy with 2.23× acceleration, largely outperforming the state of the art.

MobileNetV3-Small We implement the block pruning in MobileNetV3-Small on ImageNet ILSVRC2012 to validate the effectiveness of our method. Results are shown in Table 5.6. When μ is set to 0.2, $three$ out of $seven$ blocks are pruned with 31.8% FLOPs pruned rate, and we achieve only 1.30% accuracy drop. And when μ

Table 5.6 Block pruning for MobileNetV3-Small on ImageNet ILSVRC2012

Model	Top-1/+FT%	Top-5/+FT%	Params (PR%)	FLOPs (PR%)	Speed Up
MobileNetV3-Small[26]	67.4	87.1	2.93M	66.17M	–
EPFS-B-0.2	64.82/66.10	85.45/86.15	1.92M(34.5%)	45.12M(31.8%)	1.41×
EPFS-B-0.6	64.51/65.81	84.78/85.91	1.71M(41.6%)	37.10M(43.9%)	1.86×

Table 5.7 Controlled block pruning experiments for different optimizers on CIFAR-10. μ is set as 0.6 and 0.2 for MobileNetV3-Small and ResNet-20, respectively

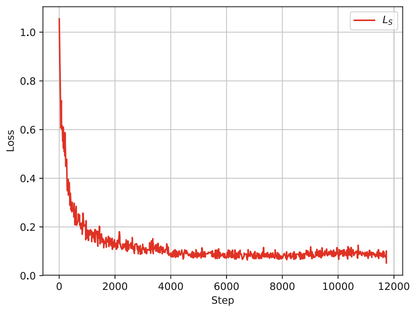
Model	Top-1/+FT%	Params(PR%)	FLOPs(PR%)
ResNet-20 [19]	92.17	0.27M	40.55M
EPFS-FISTA	91.51/91.91	0.20M(24.6%)	30.83M(24.0%)
EPFS-SGD	91.51/91.55	0.21M(22.2%)	32.34M(20.2%)
EPFS-LASSO	90.13/90.99	0.19M(29.6%)	28.31M(30.2%)
MobileNetV3-Small	93.14	2.93M	66.17M
EPFS-FISTA	92.81/92.90	2.79M(4.64%)	45.00M(32.0%)
EPFS-SGD	91.82/91.88	2.62M(10.6%)	43.11M(34.8%)
EPFS-LASSO	90.12/90.45	2.37M(19.1%)	38.12M(42.4%)

is set to 0.6, $four$ out of $seven$ blocks are pruned with 41.6% FLOPs pruned rate, and we lead to 1.59% accuracy decrease and new state-of-the-art results.

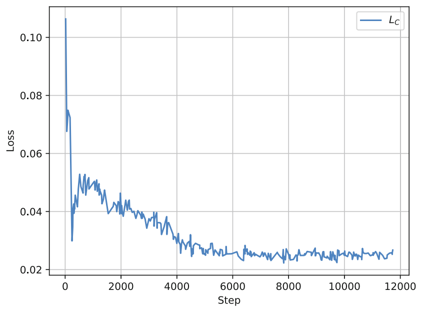
5.4.1.8 Ablation Study

The effectiveness of our method comes from FISTA and center loss. To examine how they affect the final performance, we select ResNet-20 and MobileNetV3-Large/MobileNetV3-Small for an ablation study.

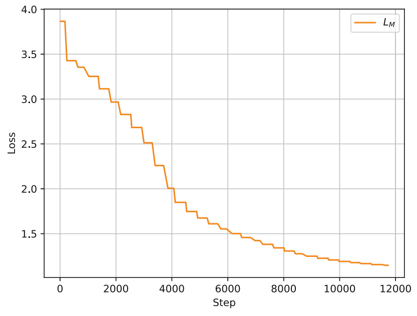
FISTA In our ablation, we use ResNet-20 and MobileNetV3-Small to prune networks based on FISTA, SGD, and LASSO. The results are presented in Table 5.7. Compared to SGD, FISTA achieves a higher accuracy and better pruning rate in the same experimental settings. Compared to LASSO, FISTA achieves a higher accuracy with a lower pruning rate. However, these results also demonstrate that the pruned network via FISTA achieves a better trade-off than LASSO. The SGD optimizer can provide better initial parameters to prune a network, but its fine-tuning performance is worse than FISTA. Generally, EPFS with FISTA can fast and steadily prune the network and achieve better accuracy than SGD and LASSO. As plotted in Fig. 5.4, $\mathcal{L}_M$, $\mathcal{L}_S$, and $\mathcal{L}_C$ can converge pretty well. $\mathcal{L}_M$ is updated on specific iterations via FISTA so that the loss curve descends discretely.



(a) L_S



(b) L_C



(c) L_M

Fig. 5.4 Loss curve of $\mathcal{L}_M$, $\mathcal{L}_S$, and $\mathcal{L}_C$

Table 5.8 Controlled block pruning experiments for center loss on CIFAR-10. μ is set as 0.6. EPFS* denotes the control group without center loss

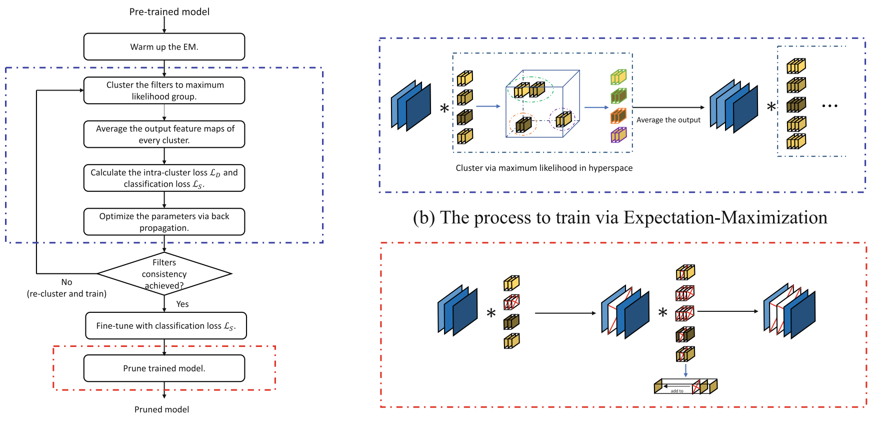
Model	Top-1/+FT%	Params(PR%)	FLOPs(PR%)
MobileNetV3-Large [26]	94.05	4.18M	227.74M
EPFS	93.83/94.07	2.95M(29.4%)	140.03M(38.5%)
EPFS*	93.54/93.65	2.72M(22.2%)	128.76M(43.5%)
MobileNetV3-Small [26]	93.14	2.93M	66.17M
EPFS	92.79/92.82	2.55M(12.9%)	38.18M(42.3%)
EPFS*	91.60/91.86	2.51M(14.3%)	29.48M(55.4%)

Center Loss To find out whether the center loss works, we used MobileNetV3-Large/MobileNetV3-Small to prune in two situations distinguished by the existence of center loss. The results are presented in Table 5.8. For MobileNetV3-Large, the EPFS without center loss got the decrease of 0.38% accuracy. Also, the EPFS without center loss achieves a decrease of 0.96% accuracy for MobileNetV3-Small. We conclude that center loss is vital in optimizing masks and stabilizing deep features.

5.4.2 *Toward Compact and Sparse CNNs via Expectation-Maximization*

Among structured pruning, filter pruning has attracted the most attention for its ability to slim the network, making it a thinner architecture without specific hardware support for accelerating. Similarly, block pruning can reduce the FLOPs of networks by shortening the network architecture. We produce a thinner and shorter architecture after pruning using the two methods. Most previous filter pruning methods [21, 22, 40] were based on the information of filters, such as the value of filter norm. They use the norm information to evaluate the filter and then hard prune or zero the filters which fail the criterion. Other methods are based on various techniques to zero out the filters, including generative adversarial learning [43], greedy search [46], Taylor expansion [42], etc.. These methods have achieved a high pruning rate with an acceptable performance drop. However, prior methods have three main areas for improvement. The first one is that filter pruning criterion based only on filter information remains insufficient, which resulted from the existence of nonlinear activation functions (e.g., rectifier linear unit (ReLU)), and other complex operations (e.g., batch normalization (BN)). For example, computing the convolution of vector $a = (0, 1)^T$ by vector $b = (5, 1)^T$ and $c = (1, 4)^T$, we have $\text{ReLU}(a * b) < \text{ReLU}(a * c)$, while $\|c\|_1 < \|b\|_1$ and $\|c\|_2 < \|b\|_2$. The second one is that prior work devoted to zeroing the output of filters may cause permanent structured damage in training. For instance, a full-precision mask is employed to sparse the output feature maps under the supervision of ℓ_1 regularization while lacking an efficient backtracking mechanism. Once the damage is caused to structured, i.e., an unsatisfactory element is updated to zero, it will never be repaired for the sparsity supervision of ℓ_1-norm. Moreover, fine-tuning always demands zeroing out the pruning pattern for the damage caused in training. However, the fine-tuning process may cause additional redundancy for pruned architecture, which causes less sparsity of remaining filters and more or less performance drop for prior pruning methods.

We focus on training a network with less redundancy and higher sparsity. The intention is to employ the expectation-maximization (EM) algorithm in the training process, as illustrated in Fig. 5.5. First, we analyze the distribution of filters in hyperspace, i.e., employ the Gaussian mixture models (GMM) to analyze the filters and EM algorithm to solve the GMM. The expectation step (E-step) is deployed to cluster the filters into the maximum likelihood distribution group. The maximization step (M-step) is employed in calculating the maximum likelihood distribution parameters and formulating a well-defined loss function to monitor the filters with similar distribution to converge to be consistent. Dynamic clustering method is implemented to reanalyze and re-cluster the filters to improve the distribution diversity. After a certain number of epochs, the distribution loss can converge to zero, which means the current network is identical to the pruned one. We fine-tune the clustered networks to optimal inter-cluster sparsity and then we can prune the network with the optimal weights.


Pre-trained model
Warm up the EM.
Cluster the filters to maximum likelihood group.
Average the output feature maps of every cluster.
Calculate the intra-cluster loss $\mathcal{L}_D$ and classification loss $\mathcal{L}_S$.
Optimize the parameters via back propagation.
Filters consistency achieved?
No (re-cluster and train)
Yes
Fine-tune with classification loss $\mathcal{L}_S$.
Prune trained model.
Pruned model
Average the output
Cluster via maximum likelihood in hyperspace
add to

(a) The complete process to prune CNNs

(b) The process to train via Expectation-Maximization

(c) The layer-wise pruning process

Fig. 5.5 Illustration of SPEM scheme. (**a**) The process includes warming up the EM algorithm, clustering the filters via the expectation step, averaging the output feature maps of every cluster, calculating the loss, optimizing parameters, and pruning the well-trained model. (**b**) Detailed illustration of training via EM, including clustering the filters with different distributions in hyperspace, averaging the output of the same cluster. For instance, the first layer's 1-st and 2-nd filters are clustered together. (**c**) Pruning two layers with filters of the same cluster sharing the consistent distribution. For instance, trim the filters sharing the same distribution to one such as the 2-nd filter in the first layer and 2-nd and 3-rd filters in the second layer. In particular, the 2-nd kernels of every filter in the second layer should be pruned by adding the second kernels to their corresponding 1-st ones

5.4.2.1 Preliminaries

As shown in Fig. 5.5, we analyze the l-th layer, and we should categorize the filters into distributions. In modern CNNs, the BN is always followed after convolutions. For the consistency of pruned networks, we regard the possible subsequent BN and scaling layers as part of the convolutional layer. First, denote the output of the j-th filter in l-th layer as:

$$x_j^l = \gamma_j^l \frac{\sum_{i=1}^{C^{l-1}} x_i^{l-1} * F_{i,j}^l - \pi_j^l}{\tau_j^l} + \beta_j^l, \tag{5.20}$$

where C^l represents the number of channels in l-th layer. x_j^l and x_i^{l-1} are the j output feature map and i-th input feature map at the l-th layer. $F_{i,j}^l$ denotes the i-th kernel of j-th filter at l-th layer. $*$ denotes the convolutional operation. $(\pi_j^l, \tau_j^l, \beta_j^l, \gamma_j^l)$ are the corresponding BN parameters of j-th channel. Hence, the parameter set of j-th channel at i-th layer is formulated as:

$$\mathcal{W}_j^l = \left\{F_j^l,\ \gamma_j^l,\ \beta_j^l,\ \pi_j^l,\ \tau_j^l\right\}. \tag{5.21}$$

To analyze the distribution, we propose the hypothesis that every filter in a pre-trained network approximately satisfies multidimensional Gaussian distribution, i.e., $F_j^l \sim \mathcal{N}(\mu_j^l, \Sigma_j^l)$. And the filters satisfy the individual hypothesis for the linearity of convolution. The clustering process can be simplified by categorizing the filters with similar distributions into the same cluster. To simplify the computation, we reshape the matrix $F_j^l \in \mathbb{R}^{\mathrm{k}^l \times \mathrm{w}^l \times C^{l-1}}$ to a vector $F_j^l \in \mathbb{R}^{(\mathrm{k}^l \cdot \mathrm{w}^l \cdot C^{l-1}) \times 1}$ during cluster process, which also satisfies the individual hypothesis.

5.4.2.2 Distribution-Aware Forward and Loss Function

Based on the hypothesis above, the l-th layer satisfies the GMM. The pruning ratio can define the GMM's dimension K^l. Then we have:

$$\begin{aligned} K^l &= < C^l \times (1 - \mathcal{E}) > \\ \mathcal{P}(\boldsymbol{F}^l | \boldsymbol{\Theta}^l) &= \alpha_k^l \sum_{k=1}^{K^l} \Phi(\boldsymbol{F}^l | \Theta_k^l), \end{aligned} \tag{5.22}$$

where $\mathcal{E}$ is a pre-defined pruning ratio to supervise the clustering process. $< a >$ represents rounding the float a to its integer approximation. α_k^l represents the ratio of k-th distribution accounting. C^l is the dimension of l-th layer. Hence, K^l is the number of clusters and the dimension of GMM. Θ_k^l denote the parameter of k-th distribution, i.e., (μ_k^l, Σ_k^l). $\boldsymbol{\Theta}^l$ denotes the assembly of Θ_k^l. $\boldsymbol{F}^l$ denotes the assembly of filters at l-th layer as well as the observed data. Then we introduce the hidden variable ξ_{jk}^l to formulate the maximum likelihood estimation (MLE) of GMM as:

$$\xi_{jk}^l = \begin{cases} 1, & F_j^l \in \mathcal{D}_k^l \\ 0, & \text{else} \end{cases} \tag{5.23}$$

In Eq. 5.23, ξ_{jk}^l is the hidden variable describing the affiliation between j-th filter and k-th cluster at l-th layer. $\mathcal{D}_k^l$ denotes the k-th distribution at l-th layer.

Based on the multi-dim Gaussian distribution hypothesis and the preliminaries above, we formulate MLE as:

$$\mathcal{P}(\boldsymbol{F}^l | \boldsymbol{\Theta}^l) = \prod_{k=1}^{K^l} \alpha^{|\mathcal{D}_k^l|} \prod_{j=1}^{C^l} \left\{ \frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma_j^l|^{\frac{1}{2}}} \times \exp\left[-\frac{1}{2} (F_j^l - \mu_k^l)^T \Sigma_j^{l^{-1}} (F_j^l - \mu_k^l) \right] \right\}^{\xi_{jk}^l}, \tag{5.24}$$

where $|\mathcal{D}_k^l| = \sum_{j=1}^{C^l} \xi_{jk}$ and $C^l = \sum_{k=1}^{K^l} |\mathcal{D}_k^l|$. d represents the dimension of filter, i.e., $\mathrm{k}^l \cdot \mathrm{w}^l \cdot C^{l-1}$. Next, we introduce the EM algorithm.

Expectation step First, we formulate the Q function as:

$$\begin{aligned}
Q(\boldsymbol{\Theta}^l, \boldsymbol{\Theta}^{l,e}) =& \mathbb{E}\left[\log \mathcal{P}(\boldsymbol{F}^l, \boldsymbol{\xi}^l|\boldsymbol{\Theta}^l)|\boldsymbol{F}^l, \boldsymbol{\Theta}^{l,e}\right] \\
=& \sum_{k=1}^{K^l}\left\{\sum_{j=1}^{C^l}(\mathbb{E}\xi_{jk}^l)\log\alpha_k^l\right. \\
&\left.+\sum_{j=1}^{C^l}(\mathbb{E}\xi_{jk}^l)\left[-\frac{1}{2}\log\,(2\pi)^d|\Sigma_j^l| + \frac{1}{2}(F_j^l-\mu_k^l)^T\Sigma_j^{l^{-1}}(F_j^l-\mu_k^l)\right]\right\},
\end{aligned}$$

where e denotes the current epoch number since we only cluster the filter once at the beginning of every epoch. Then compute the MLE of ξ_{jk}^l, i.e., $\mathbb{E}(\xi_{jk}^l|\boldsymbol{F}^l, \boldsymbol{\Theta}^l)$ via:

$$\begin{aligned}
\mathbb{E}(\xi_{jk}^l|\boldsymbol{F}^l, \boldsymbol{\Theta}^l) =& \mathcal{P}(\xi_{jk}^l = 1|\boldsymbol{F}^l, \boldsymbol{\Theta}^l) \\
=& \frac{\alpha_k^l\Phi(F_j^l|\Theta_k^l)}{\sum_{k=1}^{K^l}\alpha_k^l\Phi(F_j^l|\Theta_k^l)}.
\end{aligned} \tag{5.25}$$

We denote $\mathbb{E}(\xi_{jk}^l|\boldsymbol{F}^l, \boldsymbol{\Theta}^l)$ as $\hat{\xi}_{jk}^l$ for simplification. $\hat{\xi}_{jk}^l$ represents the relativity between k-th and j-th filter. Then we can modify the Q function by substituting $\mathbb{E}\xi_{jk}^l$ by the estimation $\hat{\xi}_{jk}^l$, so we have:

$$\begin{aligned}
Q(\boldsymbol{\Theta}^l, \boldsymbol{\Theta}^{l,t}) =& \sum_{k=1}^{K^l}\left\{\mathcal{D}_k^l\log\,\alpha_k^l + \right. \\
&\left.\sum_{j=1}^{C^l}\hat{\xi}_{jk}^l\left[-\frac{1}{2}\log\,(2\pi)^d|\Sigma_j^l| + \frac{1}{2}(F_j^l-\mu_k^l)^T\Sigma_j^{l^{-1}}(F_j^l-\mu_k^l)\right]\right\}.
\end{aligned} \tag{5.26}$$

In this way, we finish the E-step.

Then we find the coordinations of the max value of every row in $\hat{\boldsymbol{\xi}}^l$. For instance, if $\hat{\xi}_{jk}^l$ is the maximum one in j-th row, the j-th filter will be categorized into the k-th distribution cluster, as well as the corresponding BN parameter. Thus, we finish the clustering process by the E-step. Now the filters with similar distributions have been sorted out; we can define the forward and update the rule of filters. Take the k-th cluster, including the j-th filter at l-th layer, for example. We define the forward function as:

$$x_k^{l*} = \frac{1}{|\mathcal{D}_k^l|}\sum_{F_j^l\in\mathcal{D}_k^l}\frac{\sum_{i=1}^{C^{l-1}}x_i^{l-1^*} * F_{i,j}^l - \pi_j^l}{\tau_j^l} + \beta_j^l. \tag{5.27}$$

Equation 5.27 represents the procedure in that we average the output of parameters in the same cluster. $x_k^{l^*}$ represents the averaged output feature map of k-th cluster. Due to the distribution of filters in the same cluster being similar and the linearity of convolution, the outputs should also be similar. Consequently, the average conduction will not dramatically influence the network accuracy. In this way, we have formulated the forward propagation of the network successfully.

Maximization step We formulate the optimization problem as:

$$\boldsymbol{\Theta}^{l,e+1} = \arg\max_{\boldsymbol{\Theta}^l} Q(\boldsymbol{\Theta}^l, \boldsymbol{\Theta}^{l,e}) \tag{5.28}$$

As mentioned above, Θ_k^l represents (μ_k^l, Σ_k^l), and we can compute μ_k^l, Σ_k^l and α_k^l by enforcing their corresponding partial derivatives toward Q to 0. Then we have:

$$\hat{\mu}_k^{l,e+1} = \frac{\sum_{j=1}^{C^l} \hat{\xi}_{jk}^{l,e} F_j^{l,e}}{\sum_{j=1}^{C^l} \hat{\xi}_{jk}^{l,e}} \tag{5.29}$$

$$\hat{\Sigma}_k^{l,e+1} = \frac{\sum_{j=1}^{C^l} \hat{\xi}_{jk}^{l,e} (F_j^{l,e} - \mu_k^{l,e})^T (F_j^{l,e} - \mu_k^{l,e})}{\sum_{j=1}^{C^l} \hat{\xi}_{jk}^{l,e}} \tag{5.30}$$

$$\hat{\alpha}_k^{l,e} = \frac{\sum_{j=1}^{C^l} \hat{\xi}_{jk}^{l,e}}{C^l} \tag{5.31}$$

To supervise filters in the same cluster to converge to the same distribution, the loss function of i-th layer during e-th epoch can be formulated as:

$$\mathcal{L}_{D_\mu}^{l,e} = \sum_{k=1}^{K^l} \frac{\hat{\alpha}_k^{l,e}}{|\mathcal{D}_k^l|} \sum_{F_j^l \in \mathcal{D}_k^l} \|F_j^l - \hat{\mu}_k^{l,e}\|_2^2, \tag{5.32}$$

$$\mathcal{L}_{D_\Sigma}^{l,e} = \sum_{k=1}^{K^l} \frac{\hat{\alpha}_k^{l,e}}{|\mathcal{D}_k^l|} \sum_{F_j^l \in \mathcal{D}_k^l} \|(F_j^l - \hat{\mu}_k^{l,e})^T (F_j^l - \hat{\mu}_k^{l,e}) - \hat{\Sigma}_k^{l,e}\|_2^2. \tag{5.33}$$

In Eqs. 5.32 and 5.33, we formulate the supervision to constrain the constraint of filters. We denote $\mathcal{L}_D^{l,e}$ as $\mathcal{L}_D^l$ for simplification. Likewise, the $\mathcal{L}_D$ of BN parameters can be solved according to single-dimensional GMM. Hence, we can formulate $\mathcal{L}_D$ as:

$$\mathcal{L}_D = \sum_{l=1}^{L} \mathcal{L}_{D_\mu}^l + \mathcal{L}_{D_\Sigma}^l, \tag{5.34}$$

where L denotes the number of total layers of the network. The total loss supervising the parameters is formulated as:

$$\mathcal{L} = \mathcal{L}_S + \lambda \mathcal{L}_D. \tag{5.35}$$

In Eq. 5.35, the loss $\mathcal{L}$ consists of $\mathcal{L}_S$ and $\mathcal{L}_D$, i.e., cross-entropy and the distribution loss defined above.

5.4.2.3 Optimization and Analysis

We use the stochastic gradient descent (SGD) to optimize the F_j^l. Since we average some parameter output, the gradient derived from $\mathcal{L}_S$ should satisfy the chain rule. In contrast, the $\mathcal{L}_D$ comprises the parameters themselves, and the gradient can be computed straightforwardly. Hence, we formulate the optimization of the filter in k-th cluster as:

$$F_j^{l,t+1} \leftarrow F_j^{l,t} - \eta \Delta F_j^{l,t}, \tag{5.36}$$

where η is the learning rate. Then the gradient $\Delta F_j^{l,t}$ can be solved as:

$$\begin{aligned} \Delta F_j^{l,t} = \frac{\partial \mathcal{L}}{\partial F_j^l} &= \frac{\partial \mathcal{L}_S}{\partial F_j^l} + \lambda \frac{\partial \mathcal{L}_D}{\partial F_j^l} \\ &= \sum_{k=1}^{K^l} \frac{\hat{\xi}_{jk}^{l}}{|\mathcal{D}_k^l|} \cdot \frac{\partial \mathcal{L}_S}{\partial x_k^{l*}} \cdot \frac{\partial x_j^l}{\partial F_j^l} + \lambda \left(\frac{\partial \mathcal{L}_{D_\mu}^l}{\partial F_j^l} + \delta \cdot \lambda \frac{\partial \mathcal{L}_{D_\sigma}^l}{\partial F_j^l} \right). \end{aligned} \tag{5.37}$$

Based on Eq. 5.37, all the gradients become solvable. And the optimization of F_j^l becomes easy. Likewise, the parameter of BN can be solved similarly. Then the general train algorithm is shown in Algorithm 1.

5.4.2.4 Filter Modification

After training, the well-trained $\mathcal{W}$ is outputted. The filters of k-th cluster follow the same distribution, so we can leave the first one and trim others. Then we use F_k^l to denote the left one. So the l-th layer parameter set after filter pruning is:

$$\boldsymbol{F}^l = \left\{ F_k^l \in \mathcal{D}_k^l | 1 \le k \le K^l \right\} \tag{5.38}$$

Algorithm 9: Algorithm of FEM-BP

```
Input: Training data $\boldsymbol{x}^0 = \{x_1^0, \cdots, x_m^0\}$ and the corresponding ground truth labels,
hyperparameter $\lambda$, learning rate $\eta$, and filter pruning ratio $\mathcal{E}$.
Output: The trained parameter $\mathcal{W}$.
 Initialize $\mathcal{W}$=$\mathcal{W}_{baseline}$ and $e = 1$. Warm up the EM algorithm for rounds and get a set
 of $\{\boldsymbol{\mu}^0, \boldsymbol{\Sigma}^0, \boldsymbol{\alpha}^0\}$
 for $e \leq$ MaxEpoch do
   if $\mathcal{L}_D^{e-1} \neq 0$ then
     Compute and select the maximum expectation clusters from Eqs. 5.22 to 5.26.
     Compute the GMM distribution center via Eqs. 5.27 to 5.31.
   end if
   for $t \leq$ MaxIter do
     Forward the training data $\boldsymbol{x}^0 = \{x_1^0, \cdots, x_m^0\}$ via through the network.
     Calculate the network loss from Eqs. 5.32 to 5.35.
     Update $\mathcal{W}$ from Eqs. 5.36 to 5.37.
   end for
 end forObtain the sparse and compact $\mathcal{W}^*$ from $\mathcal{W}$
 return: $\mathcal{W}^*$
```

As the output dimension of the l-th layer drops to $K^l \; =< C^l \times (1-\mathcal{E}) >$ after pruning, the input dimension $(l+1)$-th layer changes correspondingly. In the same way, the input dimension of l-th layer changes to K^{l-1}. Hence, we sum the kernels corresponding to the same input to make the dimension identical, as illustrated in Fig. 5.5c. Then k-th filter in l-th layer changes to $F_k^{l*} \in \mathbb{R}^{k^l \times w^l \times K^{l-1}}$, and BN parameters obey the existence of their filters.

5.4.2.5 Experiments

Models and datasets We evaluated our SPEM method by conducting comprehensive experiments using eight convolutional neural networks on two datasets, i.e., ResNet-20/ResNet-32/ResNet-56 and VGGNet on CIFAR-10 [34] and ResNet-18/ResNet-34/ResNet-50 [19] VGGNet on ImageNet ILSVRC2012 [35]. CIFAR-10 is a dataset consisting of 50,000 training images and 10,000 test images with a size of 32×32 from 10 classes. And ImageNet ILSVRC2012 is the large-scale dataset with 1.28M training images and 50,000 validation images with a size of 224×224 from 1000 classes.

Implementations We implement the ResNets for experiments conducted on CIFAR-10 and according to [19] for experiments conducted on ImageNet ILSVRC2012. We implemented our training process on 1 NVIDIA 2080TI GPU with 11 GB and 128G RAM for experiments conducted on CIFAR-10. And for experiments conducted on ImageNet ILSVRC2012, we implemented our training process on 3 NVIDIA TITANV GPUs with 12 GB and 96 GB RAM. The weight decay is set as 1×10^{-4}, and momentum is 0.9. The hyperparameter λ is set as 1×10^{-4} and 5×10^{-6} for experiments on CIFAR-10 and ImageNet ILSVRC2012, respectively. $P.S.$, we use SPEM-$\mathcal{E}$ to present the setting of the pruning rate in experiments.

We evaluate SPEM's performance on CIFAR-10 using ResNet-20/ResNet-32/ResNet-56 and VGGNet. The initial learning rate η is set to 0.1, and the learning strategy is to scale the η by a factor of 0.1 at the 100-th and 150-th epoch. The total number of epochs is 200. And batch size is 128. Table 5.9 shows the accuracy of the base and pruned model and their absolute disparity. FLOPs of the pruned model and the pruning rate of FLOPs are also shown. We only display the results with filter pruning rate set as 30% and 40% in Table 5.9, which can achieve about $1 - (1 - 30\%)^2 = 51\%$ and $1 - (1 - 40\%)^2 = 64\%$ FLOPs pruning rate. Prior works mostly prune the FLOPs at a ratio from 30% to 65%. So we show these two kinds of results for fair comparison. More results with a filter pruning rate set from 10% to 90% will be displayed in efficiency analysis.

ResNets As shown in Table 5.9, we present the experimental results with hyperparameter $\mathcal{E}$ set as 30% and 40%. Hence, the corresponding architecture after pruning is 11-22-45 and 10-19-38, respectively. Our SPEM achieved the state-of-the-art trade-off between acceleration and accuracy. For example, FPEM-40% achieves an 11.8% higher pruning rate on ResNet-20 compared to FPGM with fine-tuning, with only 0.08% Top-1 accuracy lower. Likewise, FPEM-30% achieves only a 1.5% lower pruning rate on ResNet-20 compared to FPGM while having 0.36% Top-1 accuracy higher. To conclude, SPEM achieves a better trade-off between accuracy and acceleration than FPGM. Similarly, SPEM far outperforms MIL and SFP.

On ResNet-32, situations are similar to the ones on ResNet-20. Compared to FPGM, SPEM-30% can achieve a comparable pruning rate with 0.19% accuracy higher. And SPEM-40% achieves 1.05% and 1.71% accuracy higher than MIL/SFP with muck higher pruning rate. In conclusion, SPEM advances the state of the art on pruning ResNet-32.

On ResNet-56, we observe three phenomena: (1) FPEM-30% outperforms all the listed work with the lower pruning rate on Top-1 accuracy. (2) FPEM-40% outperforms C-SGD with a higher accuracy by 0.13% and higher pruning rate by 13.3%. (3) FPEM achieves a higher Top-1 accuracy than the baseline as well as accelerating the base ResNet-56 by a factor of 2.11× and 2.83×, which indicates that SPEM can achieve an architecture with fewer redundancy as well as higher capability to extracting features.

VGGNet We further validate our SPEM on the single-branch network such as VGGNet. Results are listed at the bottom of Table 5.9. As our work can prune without fine-tuning for the particular pruning method based on consistency, we selected the fine-tuned results of prior works for a fair comparison. Compared to PFEC, GAL, and FPGM, SPEM-30% can achieve a higher pruning rate with the Top-1 accuracy 1.01%, 0.99%, and 0.41% higher. Moreover, SPEM-40% can achieve 63.9% pruning rate, i.e., 2.77× acceleration, with only 0.05% accuracy drop, which is the best pruning result.

Table 5.9 Results on CIFAR-10. The "Acc. ↓" is the accuracy drop between the pruned model and the baseline model; the smaller, the better. And the "FLOPs ↓" is the rate describing how much the pruned FLOPs count in the baseline model; the higher, the better

Model	Method	Base Top-1%	Pruned Top-1%	Top-1 Acc. ↓ %	FLOPs	FLOPs ↓ %
ResNet-20	MIL [9]	–	91.68	–	2.60E7	20.3
ResNet-20	SFP [21]	**92.20**	90.83	1.37	2.43E7	42.2
ResNet-20	FPGM [22]	**92.20**	91.99	0.21	1.87E7	54.0
ResNet-20	SPEM-30%	92.17	**92.35**	**−0.18**	**1.92E7**	**52.5**
ResNet-20	SPEM-40%	92.17	**91.91**	**0.26**	**1.38E7**	**65.8**
ResNet-32	MIL [9]	92.33	90.74	1.59	4.70E7	31.2
ResNet-32	SFP [21]	**92.63**	92.08	0.55	4.03E7	41.5
ResNet-32	FPGM [22]	**92.63**	92.82	−0.19	3.23E7	53.2
ResNet-32	SPEM-30%	**92.63**	**93.03**	**−0.43**	**3.26E7**	**52.6**
ResNet-32	SPEM-40%	**92.63**	**92.79**	**−0.13**	**2.35E7**	**65.9**
ResNet-56	PFEC [40]	93.04	93.06	−0.02	9.09E7	27.6
ResNet-56	CP [23]	92.80	91.80	1.00	–	50.0
ResNet-56	SFP [21]	**93.59**	92.26	1.33	5.94E7	52.6
ResNet-56	FPGM [22]	**93.59**	93.49	0.10	5.94E7	52.6
ResNet-56	C-SGD [8]	93.39	93.44	−0.05	4.91E7	60.9
ResNet-56	SPEM-30%	93.26	**93.76**	**−0.50**	**5.94E7**	**52.7**
ResNet-56	SPEM-40%	93.26	**93.57**	**−0.31**	**4.28E7**	**65.9**
VGGNet	PFEC [40]	93.58	93.40	0.56	2.16E8	34.3
VGGNet	GAL [43]	**93.96**	93.42	0.54	1.80E8	45.2
VGGNet	FPGM [22]	93.58	94.00	−0.42	–	34.2
VGGNet	SPEM-30%	**93.96**	**94.41**	**−0.45**	**1.54E8**	**50.9**
VGGNet	SPEM-40%	**93.96**	**93.91**	**0.05**	**1.13E8**	**63.9**

SPEM is further evaluated on ImageNet ILSVRC2012 with ResNet-18/ResNet-34/ResNet-50 and VGGNet. We train the pruning network for 100 epochs with an initial learning rate η set as 0.1 and scaled by a factor of 0.1 at the 40-th, 60-th, and 80-th epoch. We implement the process on three GPUs with a mini-batch size of 128 for ResNet-18/ResNet-34 and VGGNet and 64 for ResNet-50, respectively. We plan to prune about 40%–60% FLOPs for models implemented on ImageNet ILSVRC2012. Hence, we will set $\mathcal{E}$ as 30% and 40$ for comparison.

ResNets As shown in Table 5.10, SPEM outperforms the prior works on ILSVRC-2012 dataset again. On ResNet-18, SPEM achieves the higher inference speedup, but its Top-1 accuracy exceeds by 1.99% and 1.22%, respectively. Compared to FPGM, SPEM-40% achieves higher accuracy by 21.1% with only an accuracy lower by 0.09%.

Table 5.10 Result on ImageNet ILSVRC2012

Model	Method	Base Top-1%	Pruned Top-1%	Top-1 Acc. ↓ %	Base Top-5%	Pruned Top-5%	Top-5 Acc. ↓ %	FLOPs ↓ %
ResNet-18	MIL [9]	69.98	66.33	3.65	89.24	86.94	2.30	34.6
ResNet-18	SFP [21]	**70.28**	67.10	3.18	**89.63**	87.78	1.85	41.8
ResNet-18	FPGM [22]	**70.28**	68.41	1.87	**89.63**	88.48	1.15	41.8
ResNet-18	SPEM-40%	69.70	**68.32**	**1.58**	89.39	**88.16**	**1.22**	**62.9**
ResNet-34	SFP[21]	**73.92**	71.83	2.09	**91.62**	90.33	1.29	41.1
ResNet-34	FPGM[22]	**73.92**	72.63	1.29	**91.62**	91.08	0.54	41.1
ResNet-34	SPEM-40%	**73.92**			**91.62**			
ResNet-50	ThiNet[46]	72.88	72.04	0.84	91.14	90.67	0.47	36.8
ResNet-50	GDP[42]	75.13	71.89	1.39	92.30	90.71	1.59	51.3
ResNet-50	SFP[21]	76.15	74.61	1.54	92.87	92.06	0.81	41.8
ResNet-50	FPGM[22]	76.15	74.83	1.32	92.87	92.32	0.55	53.5
ResNet-50	C-SGD[8]	75.33	74.93	0.40	92.56	90.27	0.29	46.2
ResNet-50	MetaPruning[8]	**76.60**	75.40	1.20	-	-	-	51.2
ResNet-50	SPEM-30%	75.22	**74.96**	**0.26**	**92.41**	**91.98**	**0.43**	**50.2**
VGGNet	SFP[21]	**73.92**	71.83	2.09	**91.62**	90.33	1.29	41.1
VGGNet	FPGM[22]	**73.92**	72.63	1.29	**91.62**	91.08	0.54	41.1
VGGNet	SPEM-40%	**73.92**			**91.62**			

5.4.2.6 Efficiency Analysis

Model Sparsity with Pruning Rate Inspired by the linearity of convolution, the amount of feature information extracted by convolutions relies much on the sparsity of filters. Hence, we define the sparsity of the filter F_j^l as $\|F_j^l - F_{GM}^l\|_2$, where F_{GM}^l is the geometric median of filters at l-th layer as defined in [22]. We plot the layer-wise sparsity of ResNet-20, ResNet-32, ResNet-56, and VGGNet in Fig. 5.6. For example, we can observe that the sparsity of pruned ResNet-32 with a pruning rate set from 10% to 40% can achieve comparable or higher layer-wise sparsity than baseline ResNet-32, which can subtly clarify the reason why the pruned ResNet-32 via SPEM can achieve even higher accuracy as well as satisfying acceleration rate compared to the baseline. In addition, we analyze the source of sparsity. It is derived from the strong constraint set in Eq. 5.27. For the outputs of filters in one cluster to be averaged, one cluster can only extract the same amount of features as just one filter. Then the gradient derived from cross-entropy forces the cluster to optimize for a sparser distribution. Hence, our main motivation can be proved theoretically and experimentally.

Another phenomenon that should be paid attention to is that the deeper network gains lower sparsity. Hence, the deeper network can accept the higher pruning rate while maintaining accuracy. For instance, pruning 40% filters on ResNet-20 and ResNet-32 will achieve 0.26% and -0.13% accuracy drop. This subtly demonstrates our viewpoint. Moreover, based on this viewpoint, we can further consider pruning the blocks and filtering for deep redundant networks such as ResNets and MoblieNets.

Joint Pruning Cooperated with Block Pruning Inspired by the GAL [43] and the model sparsity analysis, we conduct some additional experiments on pruning cooperated with block pruning set as [43] to analyze the efficiency on ResNet-56. As illustrated in Table 5.11, SPEM can work jointly with the block pruning methods. In particular, training via SPEM can effectively substitute the fine-tuning process. SPEM+GAL can achieve 3.75× and 4.69× theoretical acceleration rates.

5.4.3 Pruning Multi-view Stereo Net for Efficient 3D Reconstruction

Recent improvements in compatibility enable a fast 3D reconstruction with better accuracy and completeness, which have a wide range of applications, ranging from mapping, photogrammetry, autonomous driving, and robot planning to augmented reality and virtual reality, among many other scenarios [31, 48, 69]. To model a 3D space, depth information has to be inferred when reconstructing 3D scenarios. The most common approaches for depth inference are based on cameras with depth sensors such as Kinect, which restricts the accessibility for the outdoor environment.

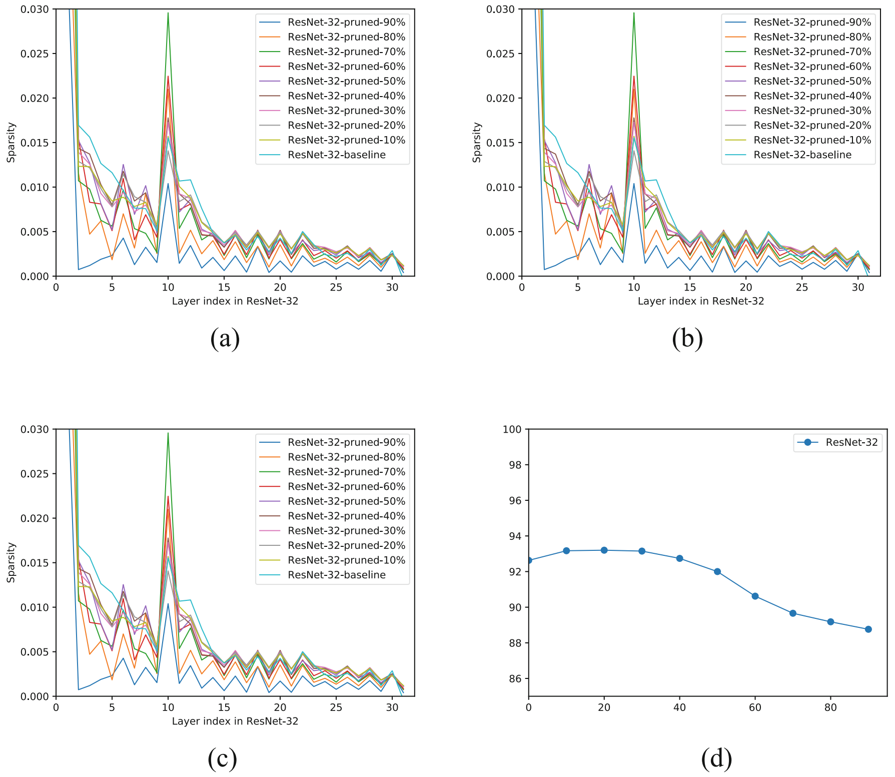


Fig. 5.6 Layer-wise sparsity analysis on various models and accuracy analysis. (**a**) Sparsity analysis on ResNet-20. (**b**) Sparsity analysis on ResNet-32. (**c**) Sparsity analysis on ResNet-56. (**d**) Accuracy of pruned model

Table 5.11 Comparison on GAL, SPEM, and SPEM+GAL. Accele. Denotes the theoretical acceleration rate

Method	Base Top-1%	Pruned Top-1%	Top-1 Acc. ↓ %	FLOPs	FLOPs ↓ %	Accele.
SPEM-30%	93.26	93.76	−0.50	5.94E7	52.7	2.09×
GAL-0.6 [43]	93.26	93.38	−0.12	8.21E7	37.6	1.60×
SPEM-30%+GAL-0.6 [43]	93.26	93.26	0.00	3.35E7	73.3	3.75×
SPEM-40%	93.26	93.57	−0.31	4.28E7	65.9	2.93×
SPEM-40%+GAL-0.6 [43]	93.26	93.07	0.19	2.80E7	78.7	4.69×

Also, such methods usually exert other forms of influence on the surface, which may cause other problems. In such a scenario, reconstructing from visible images is a practical choice. The stereo reconstruction problem was initially solved based on two visible images, while real-time multi-view processing could be more practical due to high computational cost [31, 48].

We are particularly interested in multi-view stereo (MVS). It is a more popular option in applications since more than two views improve accuracy and completeness, especially in occlusion cases. In terms of application scenarios, if multiple views are from a single moving camera and the camera captures an object from different views, a slightly different scene, or a moving object over time, then the sequence of multi-view stereo settings can be used to solve the problems of 3D reconstruction [12], structure from motion [61, 65, 66], visual SLAM [38] and visual odometry [25], and so on.

5.4.3.1 Channel Pruning for 2D CNNs

To speed up the reconstruction, we introduce PruMVS [67], which adds a soft mask to the 2D CNNs for feature extraction to prune the redundant channels and train a dense encoder-decoder structure to help prune the 3D CNNs. We first focus on filter pruning (*a.k.a.*, channel pruning) and aim to zero out some unnecessary filters by learning a soft mask. More specifically, we warp masks on each channel on the 2D part to mark the importance of the according to filter (Fig. 5.7b). The approach has several advantages. Firstly, such a technique could significantly reduce the model's size by exploiting the redundancy among filters since it has been observed that some filters are unimportant in a common CNN structure [43, 63]. Secondly, without customizing other structures in the network, pruning filters can handle many other similar MVS learning models with similar architectures. Lastly, we can produce desired redundancy by adding regularization terms on loss according to the mask and training the network. The value of the related mask could indicate the less important information. By eliminating the channel of the 2D network, the data fed into the 3D part would be significantly reduced.

5.4.3.2 Optimization Based on a Mixed Back Propagation

The soft mask removes the corresponding channels and filters for 2D CNNs. We define the weights in 2D CNNs as $\mathcal{W}$, the soft mask as $\mathbf{m}$, and λ as the parameter controlling L1 regularization term and denote the loss function as $\mathcal{L}(\mathcal{W}, \mathbf{m})$, which will be detailed in the next section. The model parameters $\mathcal{W}$, $\mathbf{m}$ are learned by solving:

$$\arg\min_{\mathcal{W},\mathbf{m}} \mathcal{L}(\mathcal{W}, \mathbf{m}) + \lambda||\mathbf{m}||_1. \tag{5.39}$$

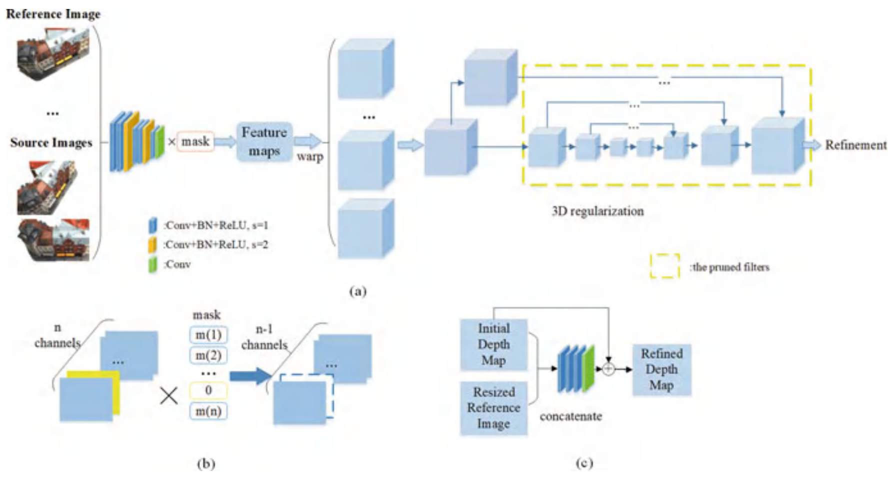


Fig. 5.7 PruMVS overview. (**a**) is a general description of our network, (**b**) corresponds to the mask, and (**c**) corresponds to refinement in (**a**). Detailed description: (**a**) The architecture of PruMVS. Reference and source images go through an eight-layer 2D CNN with a mask attached to the last layer to generate feature maps and sort out the redundant filters. Differentiable homography is used to warp the 2D images to 3D volumes and operate a variance-based algorithm to aggregate all the volumes into a single cost volume. The output depth map is generated from a 3D CNN similar to U-Net. (**b**) The illustration of channel pruning. The yellow channel in the figure denotes the redundant filter, of which the corresponding mask would be trained to zero. (**c**) Depth map refinement

Our pruning approach is simple yet principled [1, 2], and we just use **m** as a soft mask added on each filter as:

$$F_h^{l+1} = \mathbf{m}_g f(\sum_g F_g^l * \mathcal{W}_{h,g}^l), \tag{5.40}$$

where F_g^l and F_h^{l+1} are the h-th input feature map and the g-th output feature map at the l-th layer. $*$ and $f(\cdot)$ refer to the convolutional operator and activation, respectively. The mask **m** can be learned end-to-end in the mixed backpropagation process, which will be detailed later. In particular, the fast iterative shrinkage-thresholding algorithm (FISTA) [3, 43] is used to optimize **m**, which leads to a sparse solution of the soft mask and is built based on the L1-norm minimization.

Stochastic gradient descent (SGD) or RMSprop can be directly introduced to solve the optimization problem in Eq. 5.39. However, they are less efficient in convergence, and by using RMSprop, we have observed non-convergence scaling factors in the soft mask **m**. Also, most factors are of the same order of magnitude, i.e., which does not create enough sparsity in the soft mask layer. Therefore, we need a threshold to remove the corresponding structures whose scaling factors are lower than the threshold. By doing so, the accuracy of the pruned network is significantly lower than the baseline. To solve this problem, we use the proximal operator, *a.k.a.*, the proximal gradient optimization, where the SGD and the constraint are updated separately [1, 2]. We introduce FISTA into the model to effectively solve the optimization via two alternating steps updating $\mathcal{W}$ and $\mathbf{m}$.

1. Fixing $\mathbf{m}$, we use RMSprop to update $\mathcal{W}$ by descending its gradient.
2. Fixing $\mathcal{W}$, then $\mathbf{m}$ is updated by FISTA with the initialization of $\alpha_{(1)} = 1$:

$$\mathbf{n}_{(k+1)} = \mathbf{m}_{(k)} + \frac{\alpha_{(k)} - 1}{\alpha_{(k+1)}}\big(\mathbf{m}_{(k)} - \mathbf{m}_{(k-1)}\big), \tag{5.41}$$

$$\alpha_{(k+1)} = \frac{1}{2}\left(1 + \sqrt{1 + 4\alpha_{(k)}^2}\right), \tag{5.42}$$

$$\mathbf{m}_{(k+1)} = \mathbf{prox}_{\gamma_{(k+1)}\lambda||\cdot||_1}\left(\mathbf{n}_{(k+1)} - \gamma_{(k+1)}\frac{\partial\mathcal{L}(\cdot, \mathbf{n}_{(k+1)})}{\partial\mathbf{n}_{(k+1)}}\right), \tag{5.43}$$

where $\gamma_{(k+1)}$ is the learning rate at the iteration $k + 1$ and $\mathbf{prox}_{\gamma_{(k+1)}\lambda||\cdot||_1}(\mathbf{x}_i) = sign(\mathbf{x}_i) \circ (|\mathbf{x}_i| - \gamma_{(k+1)}\lambda)_+$. We can learn desirable zero and sparsity in soft mask $\mathbf{m}$ with an appropriate learning rate (Fig. 5.8).

5.4.3.3 3D CNN Pruning

As mentioned in our motivation, 3D CNNs are vital to an MVS system. For a UNet-like architecture, simply pruning channels based on existing pruning techniques are impractical for 3D CNN architecture, considering skip connections, and deconvolutions prevent us from using the most regular pruning strategies.

We have made several attempts to prune the 3D CNNs by adding soft masks, a frequently used method, yet the result could have been better. The many soft masks trained are of the same magnitude and don't show a specific pattern, i.e., zeroing out a specific channel. When a threshold filters out channels, the resulting reconstruction quality slumps significantly. This is likely caused by the fact that (1) the 3D channels in the 3D CNNs relate to each other closely and (2) the 3D CNNs are of the UNet-like architecture, which contains many skip connections and deconvolutions. The feature maps from the decoder are combined with those from the encoder sub-network via skip connections. Therefore, removing a specific feature map may lead to the information being lost in both the decoder and encoder.

To address it, *we propose a novel pruning technique by training a hierarchical architecture* and simply adopting its more minor part in our proposed PruMVS model. As shown in the network architecture (see Fig. 5.9), we add nested and dense skip connections, which is similar to U-Net++ [76], a deeply supervised encoder-decoder network where the encoder and decoder sub-networks are connected through a series of nested, dense skip pathways. When trained in deep supervision, the convolutional layers we add to the skip pathway can be pruned; thus, only level 0 is left. By adding convolutional blocks in the skipping connections and training

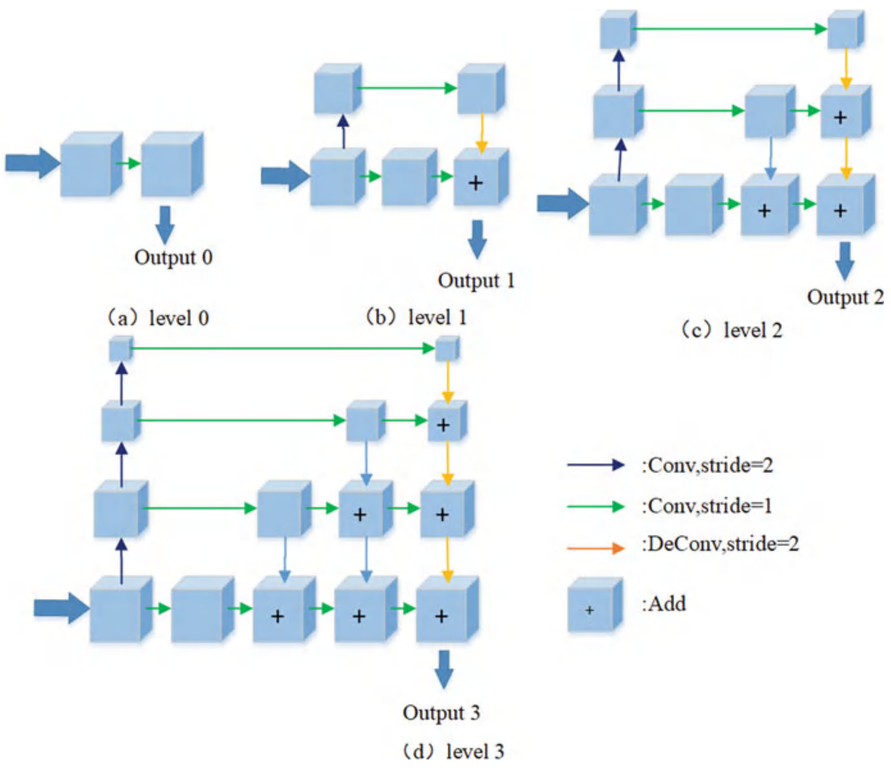


Fig. 5.8 Illustration of different levels in the 3D nested U-Net. The first level consists of two convolutional operations, while the other levels are shown in the picture above. Every level has an output, and we pick everyone to evaluate the final result, which helps pruning

the extensive network altogether, the model integrates features of different levels, and at last, we prune all the upper levels out, which helps prevent the interference of the connections between levels.

Every level could be regarded as a smaller version of the higher level and could generate its result. The re-designed skip pathways aim at reducing the semantic gap between the feature maps of the encoder and decoder sub-networks. The underlying hypothesis behind our architecture is that the model can more effectively capture fine-grained details of the foreground objects when high-resolution feature maps from the encoder network are gradually enriched before fusion with the corresponding semantically rich feature maps from the decoder network. Moreover, the network would have a more accessible learning task when the feature maps from the decoder and encoder networks are semantically similar. Higher-level results could help lower-level weights to train better through back propagation, i.e., to improve gradient flow through the skip connections. To help training converge more efficiently and prevent the deep model from gradients vanishing or exploding, we

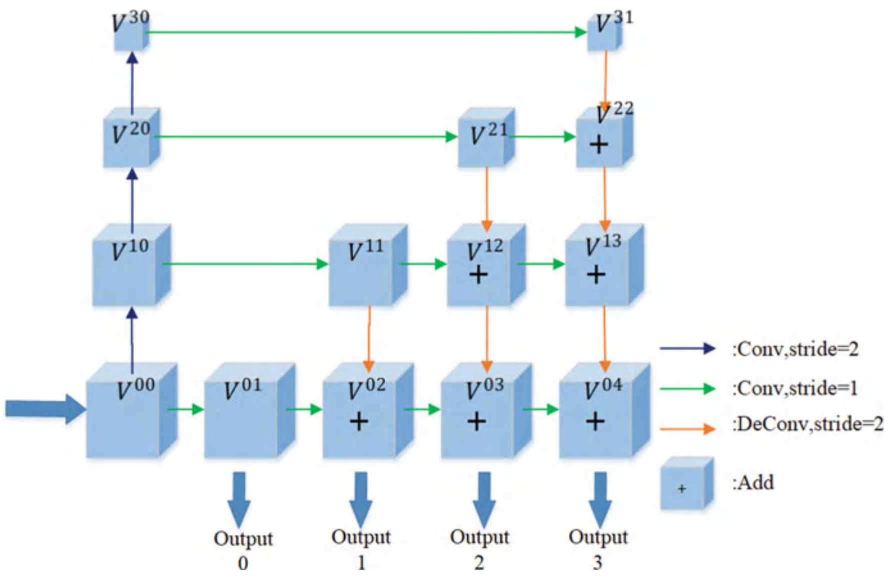


Fig. 5.9 A hierarchal architecture for 3D regularization. The 3D network shapes like a nested U-Net and has four levels. It consists of an encoder and decoder whose sub-network is connected by skip pathways represented by arrows

also adopt the deep supervision idea [37] by adding loss from different levels' results.

Now, we formulate the volume as follows:

$$V^{ij} = \begin{cases} \mathbf{C_2}(V^{i-1,j}), & j = 0 \\ \mathbf{C_1}(V^{i,0}), & j = 1 \\ \sum_0^{j-1} \mathbf{C_1}(V^{i,k}) + \mathbf{D}(V^{i+1,j-1}), & j > 1 \end{cases} \tag{5.44}$$

V^{ij} denotes the volume at the location (i, j) where i indexes the de-convolutional layer along the encoder; j denotes the sequential volume at the same ith level. $\mathbf{C_2}(\cdot)$ is a two-stride convolutional operation with a batch normalization to downsample the input, and $\mathbf{C_1}(\cdot)$ is a one-stride convolutional layer. $\mathbf{D}(\cdot)$ denotes an up-sampling layer through a two-stride de-convolutional operation. Figure 5.9 shows the overview of the proposed architecture. To be more specific, volumes at level $j = 0$ only receive an input from the previous downsampling convolutional layer, and volumes at level $j = 1$ receive an input from a common convolutional layer. Volumes at level $j > 1$ receive two inputs from the neighboring two volumes from the $j - 1$ level, generating one from the convolutional layer and the other from the up-sampling de-convolutional operation. We obtain the output of $j - 1$ level volume by adding these two inputs.

Our new investigation shows that training a hierarchical architecture improves the performance of a low-level network compared to simply training a low-level network. As a result, a lower-level network can obtain performance close to the whole net with a smaller model size. The completeness of the proposed network is better than other networks, mainly because the redundant parameters are pruned out, and only the nonredundant parameters remain. Due to the reduction of parameters, the model is not only more efficient but also more generalizable. The result is shown in the *Experiments* section.

5.4.3.4 Loss Function

Following MVSNet [69], we let p_{valid} denote the set that contains valid ground truth pixels, $d(p)$ denote the ground truth depth value of pixel p, $d_i(p)$ denote the initial depth estimation, and $d_r(p)$ denote the refined depth estimation, and then we define the loss function as:

$$Loss = \sum_{p \in \mathbf{p}_{valid}} \underbrace{\|d(p) - \hat{d}_i(p)\|}_{Loss0} + \sigma \cdot \underbrace{\|d(p) - \hat{d}_r(p)\|}_{Loss1} \tag{5.45}$$

Here *Loss0* came from the distance between the initial depth estimation and the ground truth depth, and *Loss1* came from the distance between refined depth estimation and the ground truth depth. σ leverages the two losses in the loss function. The total loss function will be modified in the following due to pruning.

Taking advantage of the nested skip connections, all the semantic level creates feature maps of the same size. As we enabled deep supervision in the new architecture, results from different levels will all be considered in the loss function. Thus, we obtain a new loss function as follows:

$$\mathcal{L} = \sum_{l=0}^{3} \sum_{p \in \mathbf{p}_{valid}} \underbrace{\|d(p) - \hat{d}_i^j(p)\|_l}_{Loss0} + \sigma \cdot \underbrace{\|d(p) - \hat{d}_r^j(p)\|_l}_{Loss1} \tag{5.46}$$

The parameter σ is set to 1.0 in the experiment. l denotes the level of the network. The model can infer two modes through deep supervision: improved performance by taking the average of different level outputs and a pruned mode by adopting the single network level.

5.4.3.5 Implementation of 2D/3D MVS Net

Feature extraction The input $\mathbf{I}_i$ includes selected source images and a reference image. An eight-layer 2D CNN is applied, where the strides of layers 3 and 6 are set to 2 to divide the feature pyramids into three scales. Two convolutional layers are applied to extract the higher-level image representation within each scale. Each convolutional layer is followed by a rectified linear unit (ReLU) except for the last layer. Also, similar to common matching tasks, parameters are shared among all feature pyramids for efficiency. The outputs of the 2D CNNs are N 32-channel feature maps downsized by four in each dimension compared with input images. Compared with simply performing dense matching on original images, the extracted feature maps significantly boost the reconstruction quality.

Finally, we will obtain N feature maps according to N different views, each of which is the size of $\frac{H}{4} \times \frac{W}{4} \times C$, where H and W is the height and width of the input image and C indicates the number of channels. It is noteworthy that though the image frame is downsized after feature extraction, the original neighboring information of each remaining pixel has already been encoded into the 32-channel pixel descriptor, which prevents dense matching from losing helpful context information.

2D to 3D Next, a 3D volume is built from the extracted feature maps and input cameras. While previous works [32] divide the space using regular grids, for our task of depth map inference, we construct the cost volume upon the reference viewing frustum. All feature maps are warped into different frontal-parallel planes of the reference camera to form N feature volumes $\{\mathbf{V}_i\}_{i=1}^{N}$.

In the 3D vision, a homography matrix $\mathbf{H}$ is used to relate a plane from one camera view into another and is subject to the rotation and translation of both views. As captured by a perspective transformation, 3D points are mapped onto image planes using the transformation matrix as $\mathbf{x(i)} = \mathbf{K}[\mathbf{R}|\mathbf{T}]\mathbf{X}$, where $\mathbf{K}$ represents the camera intrinsic, $\mathbf{R}$ is the rotation matrix, and $\mathbf{T}$ denotes the translation. Formally, let $\{\mathbf{K}_i, \mathbf{R}_i, \mathbf{T}_i\}$ be the camera parameters of image ith and $\mathbf{n}_i$ be the principal axis of the reference camera, and the homography for the ith feature map and the reference feature map at depth d could be expressed as a 3×3 matrix $\mathbf{H}_i(d)$ [69]:

$$\mathbf{H}_i(d) = \mathbf{K}_i \cdot \mathbf{R}_i \cdot (\mathbf{I} - \frac{(\mathbf{t}_1 - \mathbf{t}_i) \cdot \mathbf{n}_1^T}{d}) \cdot \mathbf{R}_1^T \cdot \mathbf{K}_1^T \tag{5.47}$$

Without loss of generality, the homography for reference feature map $\mathbf{F_1}$ itself is a 3×3 identity matrix. The warping process is similar to that of the classical plane sweeping stereo [7], except that the differentiable bilinear interpolation is used to sample pixels from feature maps$\{\mathbf{F}_i\}_{i=1}^{N}$ rather than images $\{\mathbf{I}_i\}_{i=1}^{N}$. As the core step to bridge the 2D feature extraction and the 3D regularized networks, the warping operation is implemented differently, enabling end-to-end training.

Cost volume As we obtain the feature volumes $\mathbf{V_i(d)}$ of multiple angles, including reference image and source images numbered i, the next step is to fuse the multiple features into one cost volume in the 3D space, representing the extent of which each point in the 3D space matches among different angles. Note that the size of each feature volume is $\frac{W}{4} \times \frac{H}{4} \times D \times C$, where H, W, D, C are the input image height, width, depth sample number, and the channel number of the feature map. The cost volume should be of the same size to represent every point in the 3D space. We then adapt variance among all feature volumes and define the cost volume as such variance, i.e.:

$$\mathbf{C} = \mathbf{Var}(\mathbf{V}) = \frac{\sum_{i=1}^{N}(\mathbf{V_i} - \overline{\mathbf{V}})^2}{N} \tag{5.48}$$

where N means the number of feature volumes, the same as the number of input images. As the squared deviation of a random variable from its mean, the variance measures the extent to which the value deviated from their mean, i.e., a point holding a relatively low variance shows that its value matches from various angles, explaining that the point in the 3D space is more probable to exist as a frontier in the real world since it also shows that various viewing rays should intersect at that point. On the other hand, a point having a high variance indicates that it is less probable to represent a point in the real-world 3D space.

Regularization As shown in some other research, particularly researches involving learning approaches [27, 32, 33], the raw cost volume generated by directly matching between different angles could be noisy and thus need refinement. Such a step is also known as cost volume regularization. It should consider smoothness constraints and depth information to refine the cost-volume point representation.

As described above, the cost volume **C** represents the matching extent of each 3D point. Therefore, the regularization step takes in cost volume **C**, which contains complete information of the agreement on each point from various angles, refines it, and outputs the probability volume **P**, which indicates the probability distribution for the depth inference, i.e., the probability of each point as the boundary of the 3D object in the real world.

Notably, such a process is usually accomplished by a 3D CNN regularization network, which we replaced with the multi-scale 3D CNNs in our proposed architecture, as shown in Fig. 5.7a.

Refinement While the depth map retrieved from the probability volume is a qualified output, the reconstruction boundaries may suffer from over-smoothing due to the large receptive field involved in the regularization, similar to the problems in semantic segmentation and image matting. Since the reference image contains boundary cues, the initial depth map is refined using the reference image as a supplement whose boundary information is complete, as shown in Fig. 5.7c.

The initial depth map and the resized reference image are concatenated as a 4-channel input. This is then passed through three 32-channel 2D convolutional layers and one 1-channel convolutional layer to learn the depth residual. The initial depth map is then added back to generate the refined depth map. Also, to prevent bias at a particular depth scale, we pre-scale the initial depth magnitude to the range [0, 1] and convert it back after the refinement. The last layer does not contain the BN layer and the ReLU unit to learn the negative residual.

Filtering and depth map fusion The original depth maps are inaccurate, and those outliers must be filtered out. Photometric and geometric consistency are the criteria we propose to filter out the outliers. Photometric consistency measures the matching quality, while geometric consistency measures the consistency of the maps among various views, and the pixels should be visible in at least three viewpoints. As is shown in the network, cost volume will go through a softmax layer to get a probability volume, and we filter out the pixel whose probability is lower than a threshold of 0.8.

Referring to many other MVS approaches, we choose to operate a fusion step to aggregate every depth map, which use different reference images in various views to decrease the error of the reconstructed model. We use the fusion algorithm of Gipuma [11] in our model, which differs from [69], and obtain different results in our comparison.

The mismatches mainly occur in untextured and shaded areas outside the camera's viewing frustum. Many of these cases can be found because depth maps estimated from different viewpoints differ. To detect those mismatches, we again declare that each image, in turn, the reference view, converts its depth map into a dense set of 3D points and re-projects them onto each of the $N - 1$ other views, producing a 2D coordinate p_i and a parallax $\hat{d}_i$ for each view. If $\hat{d}_i$ is equal to the relating d_i, which is kept in the depth map, the match is considered to be correct with a threshold f_ϵ which is based on the scale of the reconstructed figure. The depth is accepted if consistent in more than f_ϵ. The parameters that filter out some pixels are unreliable and must be set under a trade-off between accuracy and completeness. Different applications require different settings. In our work, we apply the same setting on every model to estimate their performance impartially.

5.4.3.6 Performance Comparison

To show how our proposed PruMVS balance the reconstruction performance and the model size pretty well, we examine our approach on the evaluation set consisting of 22 scenes and compare it with the other state-of-the-art methods, including not only the baseline MVSNet [69] but also Gipuma [11], Camp [6], Tola [60], SurfaceNet [32], and Furu [10] which is called PMVS in [69].

To ensure the methods are all evaluated similarly, we utilized the same depth fusion algorithm with the same settings on our network. For other models that reconstruct from grids and point clouds with only steps similar to depth fusion and filtering, we experiment on the best settings they proposed with their techniques, which is fair to them. Notably, due to the variety of 3D reconstruction approaches (point cloud, depth map, grids, voxels), such post-processing and refinement steps (fusion and filtering) may vary significantly. But we have controlled varying factors as much as possible in the evaluation.

Table 5.12 compares the quantitative evaluations: PruMVS maintains good accuracy and completeness, which is comparable to the best ones. Even the smallest model outperforms most other approaches, and the 32-channel level 0 model obtained the best completeness. It is noticeable that the objective of our model is to accelerate the reconstruction, which is very useful for practical applications.

Table 5.12 Comparison with other methods

Methods	Error	Comp.(err)	Overall(err)
Furu (PMVS)	0.613	0.941	0.777
Gipuma	0.283	0.873	0.578
Camp	0.834	0.554	0.694
Tola	0.342	1.190	0.766
Colmap	0.400	0.664	0.532
SurfaceNet	0.454	1.354	0.904
MVSNet	0.569	0.609	0.589
PruMVS level 0	0.495	**0.433**	**0.464**
PruMVS level 0 16C	0.510	0.451	0.481

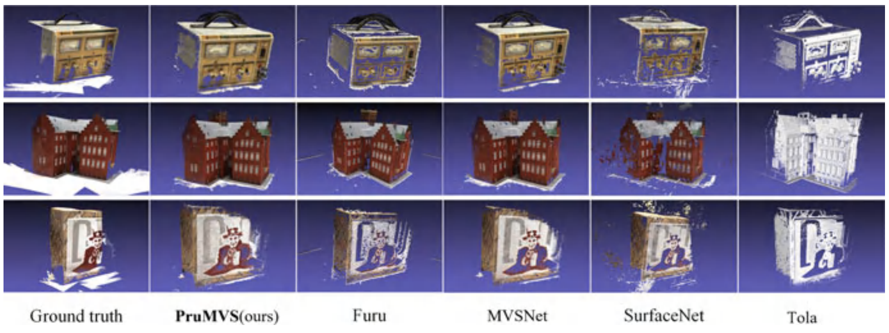


Fig. 5.10 Qualitative comparison to ground truth in DTU dataset and the reconstruction model generated by other networks

Figure 5.10 summarizes the qualitative comparison against the ground truth and the baseline MVSNet and Furu, SurfaceNet, and Tola.

5.4.4 Cogradient Descent for Dependable Learning

5.4.4.1 Gradient Descent

A basic bilinear optimization problem attempts to optimize the following objective function as:

$$\arg\min_{\mathbf{A},\mathbf{x}} G(\mathbf{A}, \mathbf{x}) = \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda\|\mathbf{x}\|_1 + R(A), \tag{5.49}$$

where $\mathbf{b} \in \mathbb{R}^{M\times 1}$ is an observation that can be characterized by $\mathbf{A} \in \mathbb{R}^{M\times N}$ and $\mathbf{x} \in \mathbb{R}^{N\times 1}$. $R(\cdot)$ represents the regularization, typically the ℓ_1 or ℓ_2 norm. $\|\mathbf{b}-\mathbf{A}\mathbf{x}\|_2^2$ can be replaced by any function with the form $\mathbf{A}\mathbf{x}$. Bilinear models generally have one variable with a sparsity constraint such as ℓ_1 regularization with the aim of avoiding overfitting.

Assuming **A** and **x** are independent, the conventional gradient descent method can be used to solve the bilinear optimization problem as:

$$\mathbf{A}^{t+1} = \mathbf{A}^t + \eta_1 \frac{\partial G}{\partial \mathbf{A}}, \tag{5.50}$$

where

$$(\frac{\partial G}{\partial \mathbf{A}})^T = \mathbf{x}^t(\mathbf{A}\mathbf{x}^t - \mathbf{b})^T = \mathbf{x}^t \hat{G}(\mathbf{A}, \mathbf{x}). \tag{5.51}$$

The function $\hat{G}$ is defined by considering the bilinear optimization problem as in Eq. 5.49, and we have:

$$\hat{G}(A, x) = (\mathbf{A}\mathbf{x}^t - \mathbf{b})^T. \tag{5.52}$$

Equation 5.51 shows that the gradient for **A** tends to vanish, when **x** approaches zero due to the sparsity regularization term $||x||_1$. Although it has a chance to be corrected in some tasks, more likely, the update will cause an asynchronous convergence. Note that for simplicity, the regularization term on A is not considered. Similarly, for **x**, we have:

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \eta_2 \frac{\partial G}{\partial \mathbf{x}}. \tag{5.53}$$

η_1 and η_2 are the learning rates. The conventional gradient descent algorithm for bilinear models iteratively optimizes one variable while keeping the other fixed. This unfortunately ignores the relationship of the two hidden variables in optimization.

5.4.4.2 Cogradient Descent for Dependable Learning

We consider the problem from a new perspective such that **A** and **x** are coupled. Firstly, based on the chain rule [52] and its notations, we have:

$$\hat{x}_j^{t+1} = x_j^t + \eta_2 \left(\frac{\partial G}{\partial x_j} + Tr\left(\left(\frac{\partial G}{\partial \mathbf{A}} \right)^T \frac{\partial \mathbf{A}}{\partial x_j} \right) \right), \tag{5.54}$$

where $(\frac{\partial G}{\partial \mathbf{A}})^T = \mathbf{x}^t \hat{G}(\mathbf{A}, \mathbf{x})$ as shown in Eq. 5.51. $Tr(\cdot)$ represents the trace of the matrix, which means that each element in the matrix $\frac{\partial G}{\partial x_j}$ adds the trace of the corresponding matrix related to x_j. Considering:

$$\frac{\partial G}{\partial \mathbf{A}} = \mathbf{A}\mathbf{x}^t\mathbf{x}^{T,t} - b\mathbf{x}^{T,t}, \tag{5.55}$$

we have:

$$\begin{aligned}\frac{\partial G(\mathbf{A})}{\partial x_j} &= Tr[(\mathbf{A}\mathbf{x}^t\mathbf{x}^{T,t} - b\mathbf{x}^{T,t})^T \frac{\partial \mathbf{A}}{\partial x_j}] \\ &= Tr[((\mathbf{A}\mathbf{x}^t - b)\mathbf{x}^{T,t})^T]\frac{\partial \mathbf{A}}{\partial x_j} \\ &= Tr[\mathbf{x}^t \hat{G}\frac{\partial \mathbf{A}}{\partial x_j}],\end{aligned} \tag{5.56}$$

where $\hat{G} = (\mathbf{A}\mathbf{x}^t - b)^T = [\hat{g}_1, \ldots, \hat{g}_M]$. Supposing that $\mathbf{A}_i$ and x_j are independent when $i \neq j$, we have:

$$\frac{\partial \mathbf{A}}{\partial x_j} = \begin{bmatrix} 0 \ldots \frac{\partial \mathbf{A}_{1j}}{\partial x_j} \ldots 0 \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ 0 \ldots \frac{\partial \mathbf{A}_{Mj}}{\partial x_j} \ldots 0 \end{bmatrix}, \tag{5.57}$$

and:

$$\mathbf{x}\hat{G} = \begin{bmatrix} x_1\hat{g}_1 & \ldots & x_1\hat{g}_j & \ldots & x_1\hat{g}_M \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot \\ x_N\hat{g}_1 & \ldots & x_N\hat{g}_j & \ldots & x_N\hat{g}_M \end{bmatrix}. \tag{5.58}$$

Combining Eqs. 5.57 and 5.58, we have:

$$\mathbf{x}\hat{G}\frac{\partial \mathbf{A}}{\partial x_j} = \begin{bmatrix} 0 \ldots x_1 \sum_i^M \hat{g}_i \frac{\partial \mathbf{A}_{ij}}{\partial x_j} \ldots 0 \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ \cdot \quad\quad \cdot \quad\quad \cdot \\ 0 \ldots x_N \sum_i^M \hat{g}_i \frac{\partial \mathbf{A}_{ij}}{\partial x_j} \ldots 0 \end{bmatrix}. \tag{5.59}$$

The trace of Eq. 5.59 is then calculated by:

$$Tr[\mathbf{x}^t \hat{G}\frac{\partial \mathbf{A}}{\partial x_j}] = x_j \sum_i^M \hat{g}_i \frac{\partial \mathbf{A}_{ij}}{\partial x_j}. \tag{5.60}$$

Remembering that $\mathbf{x}^{t+1} = \mathbf{x}^t + \eta_2 \frac{\partial G}{\partial \mathbf{x}}$, CoGD is established by combining Eqs. 5.54 and 5.60:

$$
\begin{aligned}
\hat{\mathbf{x}}^{t+1} &= \mathbf{x}^{t+1} + \eta_2 \begin{bmatrix} \sum_i^M \hat{g}_i \frac{\partial \mathbf{A}_{i1}}{\partial x_1} \\ . \\ . \\ . \\ \sum_i^M \hat{g}_i \frac{\partial \mathbf{A}_{iN}}{\partial x_N} \end{bmatrix} \odot \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_N \end{bmatrix} \\
&= \mathbf{x}^{t+1} + \eta_2 \begin{bmatrix} < \hat{G}, \frac{\partial \mathbf{A}_1}{\partial x_1} > \\ . \\ . \\ . \\ < \hat{G}, \frac{\partial \mathbf{A}_N}{\partial x_N} > \end{bmatrix} \odot \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_N \end{bmatrix} \\
&= \mathbf{x}^{t+1} + \eta_2 \mathbf{c} \odot \mathbf{x}^t.
\end{aligned}
\tag{5.61}
$$

We further define the kernelized version of $\mathbf{c}$ and have:

$$
\mathbf{c} = \begin{bmatrix} \hat{K}(\hat{G}, \frac{\partial \mathbf{A}_1}{\partial x_1}) \\ . \\ . \\ . \\ \hat{K}(\hat{G}, \frac{\partial \mathbf{A}_N}{\partial x_N}) \end{bmatrix}, \tag{5.62}
$$

where $\hat{K}(.,.)$ is a kernel function.[1] Remembering that Eq. 5.53, $\mathbf{x}^{t+1} = \mathbf{x}^t + \eta_2 \frac{\partial G}{\partial \mathbf{x}}$, Eq. 5.54 then becomes:

$$
\hat{\mathbf{x}}^{t+1} = \mathbf{x}^{t+1} + \eta_2 \mathbf{c}^t \odot \mathbf{x}^t, \tag{5.63}
$$

where $\odot$ represents the Hadamard product. It is then reformulated as a projection function as:

$$
\hat{\mathbf{x}}^{t+1} = P(\mathbf{x}^{t+1}, \mathbf{x}^t) = \mathbf{x}^{t+1} + \beta \odot \mathbf{x}^t, \tag{5.64}
$$

which shows the rationality of our method, *i.e.*, it is based on a projection function to solve the asynchronous problem of the bilinear optimization by controlling β.

We first judge when an asynchronous convergence happens in the optimization based on a form of logical operation as:

$$
(\neg s(\mathbf{x})) \wedge (s(\mathbf{A})) = 1, \tag{5.65}
$$

[1] $\hat{K}(x1, x2) = (x1 \cdot x2)^k$.

and

$$s(*) = \begin{cases} 1 & if\ R(*)\ \geq \alpha, \\ 0 & otherwise, \end{cases} \tag{5.66}$$

where α represents the threshold which changes for different applications. Equation 5.65 describes an assumption that an asynchronous convergence happens for $\mathbf{A}$ and $\mathbf{x}$ when their norms become significantly different. Accordingly, the update rule of the proposed CoGD [62] is defined as:

$$\hat{\mathbf{x}}^{t+1} = \begin{cases} P(\mathbf{x}^{t+1}, \mathbf{x}^{t}) & if\ (\neg s(\mathbf{x})) \wedge (s(\mathbf{A})) = 1, \\ \mathbf{x}^{t+1} & otherwise, \end{cases} \tag{5.67}$$

which leads to a synchronous convergence and generalizes the conventional gradient descent method. CoGD is then established.

Note that c in Eq. 5.61 is calculated based on $\hat{G}$, which differs for applications. $\frac{\partial \mathbf{A}_j}{\partial x_j} \approx \frac{\Delta \mathbf{A}_j}{\Delta x_j}$, where Δ denotes the difference of the variable over the epoch related to the convergence speed. $\frac{\partial \mathbf{A}_j}{\partial x_j} = \mathbf{1}$, if Δx_j or x_j approaches to zero. With above derivation, we define CoGD within the gradient descent framework, providing a solid foundation for the convergence analysis of CoGD. Based on CoGD, the variables are sufficiently trained and decoupled, which can enhance the causality of the learning system [74].

5.4.4.3 Applications

We apply the proposed algorithm on convolutional sparse coding (CSC) and deep learning to validate its general applicability to bilinear problems including image inpainting, image reconstruction, network pruning, and CNN model training.

Convolutional Sparse Coding

CSC operates on the whole image, decomposing a global dictionary and set of features. The CSC problem is theoretically more challenging than the patch-based sparse coding [47] and requires more sophisticated optimization model. The reconstruction process is usually based on a bilinear optimization model formulated as:

$$\begin{aligned} &\arg\min_{\mathbf{A},\mathbf{x}} \frac{1}{2} \left\| \mathbf{b} - \mathbf{A}\mathbf{x} \right\|_F^2 + \lambda \left\| \mathbf{x} \right\|_1 \\ &s.t.\ \left\| \mathbf{A}_k \right\|_2^2 \leq 1 \quad \forall k \in \{1, \ldots, K\}, \end{aligned} \tag{5.68}$$

where $\mathbf{b}$ denotes input images.

$\mathbf{x} = [\mathbf{x}_1^T, \ldots, \mathbf{x}_K^T]^T$ denotes coefficients under sparsity regularization. λ is the sparsity regularization factor. $\mathbf{A} = [\mathbf{A}_1, \ldots, \mathbf{A}_K]$ is a concatenation of Toeplitz matrices representing the convolution with the kernel filters $\mathbf{A}_k$, where K is the number of the kernels.

In Eq. 5.68, the optimized objectives or models are influenced by two or more hidden factors that interact to produce the observations. Existing solution tends to decompose the bilinear optimization problem into manageable subproblems [24, 50]. Without considering the relationship between two hidden factors, however, existing methods suffer from suboptimal solutions caused by an asynchronous convergence speed of the hidden variables. We attempt to purse an optimized solution based on the proposed CoGD.

Specifically, we introduce a diagonal or block-diagonal matrix $\mathbf{m}$ to the sparse coding framework defined in [17] and reformulate Eq. 5.68 as:

$$\arg\min_{\mathbf{A},\mathbf{x}} f_1(\mathbf{Ax}) + \sum_{k=1}^{K} (f_2(\mathbf{x}_k) + f_3(\mathbf{A}_k)), \tag{5.69}$$

where

$$\begin{aligned} f_1(\mathbf{v}) &= \frac{1}{2} \|\mathbf{b} - \mathbf{mv}\|_F^2, \\ f_2(\mathbf{v}) &= \lambda \|\mathbf{v}\|_1, \\ f_3(\mathbf{v}) &= ind_c(\mathbf{v}). \end{aligned} \tag{5.70}$$

In Eq. 5.70, $ind_c(\cdot)$ is an indicator function defined on the convex set of the constraints $C = \{\mathbf{x} | \, \|\mathbf{Sx}\|_2^2 \leq 1\}$. Similar to Eq. 5.67, we have:

$$\hat{\mathbf{x}}_k = \begin{cases} P(\mathbf{x}_k^{t+1}, \mathbf{x}_k^t) & if\ (\neg s(\mathbf{x}_k)) \wedge (s(\mathbf{A}_k)) = 1 \\ \mathbf{x}_k^{t+1} & otherwise \end{cases}, \tag{5.71}$$

which solve the two coupled variables iteratively. $P(\mathbf{x}_k^{t+1}, \mathbf{x}_k^t)$ is calculated based on $\hat{G}(\mathbf{A}, \mathbf{x})$, which is defined in Eq. 5.52.

5.4.4.4 Network Pruning

Network pruning, particularly convolutional channel pruning, has received increased attention for compressing CNNs. Early works in this area tended to directly prune the kernel based on simple criteria like the norm of kernel weights [40] or use a greedy algorithm [46]. Recent approaches have formulated network pruning as a bilinear optimization problem with soft masks and sparsity regularization [23, 28, 43, 70].

Based on the framework of channel pruning [23, 28, 43, 70], we apply the proposed CoGD for network pruning. To prune a channel of the network, the soft

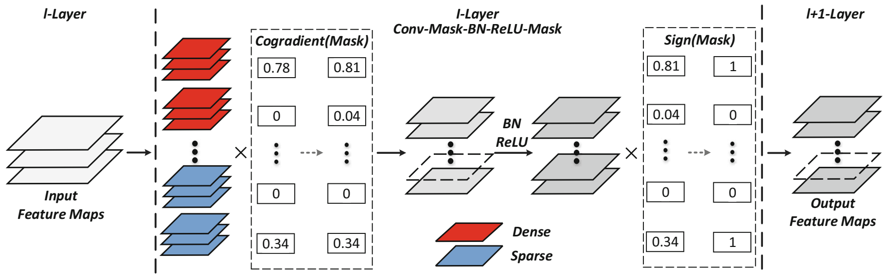


Fig. 5.11 The forward process with the soft mask

mask **m** is introduced after the convolutional layer to guide the output channel pruning. This is defined as a bilinear model as:

$$F_j^{l+1} = f(\sum_i F_i^l \otimes (W_{i,j}^l \mathbf{m}_j)), \tag{5.72}$$

where F_j^l and F_j^{l+1} are the i-th input and the j-th output feature maps at the l-th and $(l+1)$-th layer. $W_{i,j}^l$ are convolutional filters corresponding to the soft mask **m**. $\otimes$ and $f(\cdot)$ respectively refer to convolutional operator and activation.

In this framework shown in Fig. 5.11, the soft mask **m** is learned end-to-end in the back propagation process. To be consistent with other pruning works, we use W and **m** instead of **A** and **x**. A general optimization function for network pruning with a soft mask is defined as:

$$\arg\min_{W,\mathbf{m}} \mathcal{L}(W, \mathbf{m}) + \lambda\|\mathbf{m}\|_1 + R(W), \tag{5.73}$$

where $\mathcal{L}(W, \mathbf{m})$ is the loss function, described in details below. With the sparsity constraint on **m**, the convolutional filters with zero value in the corresponding soft mask are regarded as useless filters. This means that these filters and their corresponding channels in the feature maps have no significant contribution to the network predictions and should be pruned. There is, however, a dilemma in the pruning-aware training in that the pruned filters are not evaluated well before they are pruned, which leads to suboptimal pruning. In particular, the soft mask **m** and the corresponding kernels are not sparse in a synchronous manner, which can prune the kernels still of potentials. To address this problem, we apply the proposed CoGD to calculate the soft mask **m**, by reformulating Eq. 5.67 as:

$$\hat{\mathbf{m}}_j^{l,t+1} = \begin{cases} P(\mathbf{m}_j{}^{l,t+1}, \mathbf{m}_j{}^{l,t}) & if\ (\neg s(\mathbf{m}_j^{l,t})) \wedge s(\sum_i W_{i,j}^l) = 1 \\ \mathbf{m}_j^{l,t+1} & otherwise, \end{cases} \tag{5.74}$$

where $W_{i,j}$ represents the 2D kernel of the i-th input channel of the j-th filter. β, α_W, and $\alpha_{\mathbf{m}}$ are detailed in experiments. The form of $\hat{G}$ is specific for different applications. For CNN pruning, based on Eq. 5.51, we simplify the calculation of $\hat{G}$ as:

$$\hat{G} = \frac{\partial \mathcal{L}}{\partial W_{i,j}} / \mathbf{m}_j. \tag{5.75}$$

Note that the autograd package in deep learning frameworks such as PyTorch [51] can automatically calculate $\frac{\partial \mathcal{L}}{\partial W_{i,j}}$. We then substitute Eq. 5.75 into Eq. 5.62 to train our network and prune CNNs based on the new mask $\hat{\mathbf{m}}$ in Eq. 5.74.

To examine how our CoGD works for network pruning, we use GAL [43] as an example to describe our CoGD for CNN pruning. A pruned network obtained through GAL with ℓ_1-regularization on the soft mask is used to approximate the pre-trained network by aligning their outputs. The discriminator D with weights W_D is introduced to discriminate between the output of the pre-trained network and the pruned network. The pruned network generator G with weights W_G and soft mask $\mathbf{m}$ is learned together with D by using the knowledge from supervised features of the baseline. Accordingly, the soft mask $\mathbf{m}$, the new mask $\hat{\mathbf{m}}$, the pruned network weights W_G, and the discriminator weights W_D are simultaneously learned by solving the optimization problem as follows:

$$\begin{aligned} \arg \min_{W_G, \mathbf{m}} \max_{W_D, \hat{\mathbf{m}}} & \mathcal{L}_{Adv}(W_G, \hat{\mathbf{m}}, W_D) + \mathcal{L}_{data}(W_G, \hat{\mathbf{m}}) \\ & + \mathcal{L}_{reg}(W_G, \mathbf{m}, W_D). \end{aligned} \tag{5.76}$$

where $\mathcal{L}(W, \mathbf{m}) = \mathcal{L}_{Adv}(W_G, \hat{\mathbf{m}}, W_D) + \mathcal{L}_{data}(W_G, \hat{\mathbf{m}})$ and $\mathcal{L}_{reg}(W_G, \mathbf{m}, W_D)$ are related to $\lambda\|\mathbf{m}\|_1 + R(W)$ in Eq. 5.73. $\mathcal{L}_{Adv}(W_G, \hat{\mathbf{m}}, W_D)$ is the adversarial loss to train the two-player game between the pre-trained network and the pruned network that compete with each other.

The advantages of CoGD in network pruning are threefold. First, CoGD that optimizes the bilinear pruning model leads to a synchronous gradient convergence. Second, the process is controllable by a threshold, which makes the pruning rate easy to adjust. Third, the CoGD method for network pruning is scalable, *i.e.*, it can be built upon other state-of-the-art networks for better performance.

CNN Training

The last but not the least, CoGD can be fused with the batch normalization (BN) layer and improve the performance of CNN models. As is known, the BN layer can redistribute the features, resulting in the convergence of the feature and kernel learning in an asynchronous manner. CoGD is then introduced to synchronize their learning speeds to sufficiently train CNN models. Specifically, we backtrack sparse convolutional kernels through evaluating the sparsity of the BN layer, leading to an efficient training process. An interesting application of CoGD is studied in CNN learning. Considering the linearity of the convolutional operation, CNN training can also be considered as a bilinear optimization task as:

$$F_j^{l+1} = f(BN(\sum_i F_i^l \otimes W_{i,j}^l)), \tag{5.77}$$

where F_j^l and F_j^{l+1} are the i-th input and the j-th output feature maps at the l-th and $(l+1)$-th layer, $W_{i,j}^l$ are convolutional filters, and $\otimes$, $BN(\cdot)$, and $f(\cdot)$ refer to convolutional operator, batch normalization, and activation, respectively. However, the convolutional operation is not as efficient as a traditional bilinear model. We instead consider a batch normalization (BN) layer to validate our method, which can be formulated as a bilinear optimization problem as detailed in Sect. 4.2. We use the CoGD to replace SGD to efficiently learn the CNN, with the aim of validating the effectiveness of the proposed method.

To ease presentation, we first copy Eq. 5.77 as:

$$F_j^{l+1} = f(BN(\sum_i F_i^l \otimes W_{i,j}^l)), and \tag{5.78}$$

then redefine the BN model as:

$$\begin{aligned} BN(x) &= \gamma \frac{x - \mu_B}{\sqrt{\sigma_B + \epsilon}} + \beta, \\ \mu_B &= \frac{1}{m}\sum_{i=1}^{m} x_i, \\ \sigma_B &= \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2, \end{aligned} \tag{5.79}$$

where m is the mini-batch size, μ_B and σ_B are mean and variance obtained by feature calculation in the BN layer. γ and β are the learnable parameters, and ϵ is a small number to avoid dividing by zero.

According to Eqs. 5.78 and 5.79, we can easily know that γ and W are bilinear. We use the sparsity of γ instead of the whole convolutional features for kernel backtracking, which simplifies the operation and improves the backtracking efficiency. Similar to network pruning, we also use γ and W instead of $\mathbf{A}$ and $\mathbf{x}$ in this part. A general optimization for CNN training with the BN layer is:

$$\arg\min_{W,\gamma} \mathcal{L}(W, \gamma) + \lambda \|W\|_1, \tag{5.80}$$

where $\mathcal{L}(W, \gamma)$ is the loss function defined on Eqs. 5.78 and 5.79. CoGD is then applied to train CNNs, by reformulating Eq. 5.67 as:

$$\hat{W}_{i,j}^{l,t+1} = \begin{cases} P({W_{i,j}}^{l,t+1}, {W_{i,j}}^{l,t}) & if\ (\neg s(\gamma_j^l)) \wedge s(\sum_i W_{i,j}^l) = 1 \\ {W_{i,j}}^{l,t} & otherwise, \end{cases} \tag{5.81}$$

where γ_j^l is the j-th learnable parameter in the l-th BN layer. $W_{i,j}$ represents the 2D kernel of the i-th input channel of the j-th filter. Similar to network pruning, we define:

$$\hat{G} = \frac{\partial \mathcal{L}}{\partial W_{i,j}} / \boldsymbol{\gamma}_j, \tag{5.82}$$

where $\frac{\partial \mathcal{L}}{\partial W_{i,j}}$ is obtained based on the autograd package in deep learning frameworks such as PyTorch [51]. Similar to network pruning, we substitute Eq. 5.82 into Eq. 5.62, to use CoGD for CNN training.

5.4.4.5 Experiments

In this section, CoGD is first analyzed and compared with classical optimization methods on a baseline problem. It is then validated on the problems of CSC, network pruning, and CNN model training.

Baseline Problem
A baseline problem is first used as an example to illustrate the superiority of our algorithm. The problem is the optimization of Beale function[2] under constraint of $F(x_1, x_2) = beale(x_1, x_2) + \|x_1\| + x_2^2$. The Beale function has the same form as Eq. 5.49 and can be regraded as a bilinear optimization problem with respect to variables $x_1 x_2$. During optimization, the learning rate η_2 is set as 0.001, 0.005, and 0.1 for "SGD," "momentum," and "Adam," respectively. The thresholds α_{x_1} and α_{x_2} for CoGD are set to 1 and 0.5. $\beta = 0.001\eta_2 \mathbf{c}^t$ with $\frac{\partial x_2}{\partial x_1} \approx \frac{\Delta x_2}{\Delta x_1}$, where Δ denotes the difference of variable over the epoch. $\frac{\Delta x_2}{\Delta x_1} = \mathbf{1}$, when Δx_2 or x_2 approaches zero. The total number of iterations is 200.

In Fig. 5.12, we compare the optimization paths of CoGD with those of the three widely used optimization methods – "SGD," "momentum," and "Adam." It can be seen that algorithms equipped with CoGD have shorter optimization paths than their counterparts. Particularly, the ADAM-CoGD algorithm has a much shorter path than ADAM, demonstrating the fast convergence of the proposed CoGD algorithm. The similar convergence with shorter paths means that CoGD facilitates efficient and sufficient training.

[2] $beale(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.62 - x_1 + x_1 x_2^3)^2$.

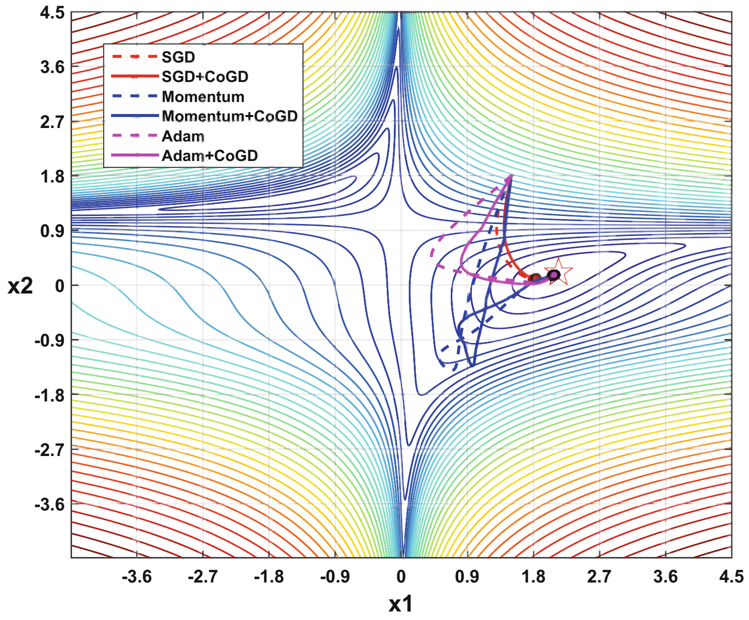


Fig. 5.12 Comparison of classical gradient algorithm with CoGD.The background is the contour map of Beale functions. The algorithms with CoGD have short optimization paths compared with their counterparts, which shows that CoGD facilitates efficient and sufficient training

Convolutional Sparse Coding

Experimental Setting The CoGD for convolutional sparse coding (CSC) is evaluated on two public datasets: the fruit dataset [72] and the city dataset [24, 72], each of which consists of ten images with 100 × 100 resolution. To evaluate the quality of the reconstructed images, we use two standard metrics, the peak signal-to-noise ratio (PSNR, unit: dB), and the structural similarity (SSIM). The higher the PSNR and the SSIM values are, the better the visual quality of the reconstructed image is. The evaluation metrics are defined as:

$$PSNR = 10 \times \log_{10}(\frac{MAX^2}{MSR}), \tag{5.83}$$

where MSE is the mean square error of clean image and noisy image. MAX is the maximum pixel value of the image:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}, \tag{5.84}$$

where μ is the mean of samples. σ is the variance of samples. σ_{xy} is the covariance of the samples. C is a constant, $C_1 = (0.01 \times MAX)^2$, $C_2 = (0.03 \times MAX)^2$

Implementation Details The reconstruction model is implemented based on the conventional CSC method [17], while we introduce the CoGD with the kernelized projection function to achieve a better convergence and higher reconstruction accuracy. One hundred of filters with size 11×11 are set as model parameters. $\alpha_{\mathbf{x}}$ is set to the mean of $\|\mathbf{x}_k\|_1$. $\alpha_{\mathbf{A}}$ is calculated as the median of the sorted results of β_k. As shown in Eq. 5.62, linear and polynomial kernel functions are used in the experiment, which can both improve the performance of our method. For a fair comparison, we use the same hyperparameters (η_2) in both our method and [17]. We also test $\beta = 0.1\eta_2\mathbf{c}^t$, which achieves a similar performance as the linear kernel.

Results The CSC with the proposed CoGD algorithm is evaluated with two tasks including image reconstruction and image inpainting.

For image inpainting, we randomly sample the data with a 75% subsampling rate, to obtain the incomplete data. Like [24], we test our method on contrast-normalized images. We first learn filters from all the incomplete data under the guidance of the soft mask $\mathbf{m}$ and then reconstruct the incomplete data by fixing the learned filters. We show inpainting results of the normalized data in Fig. 5.13. Moreover, to

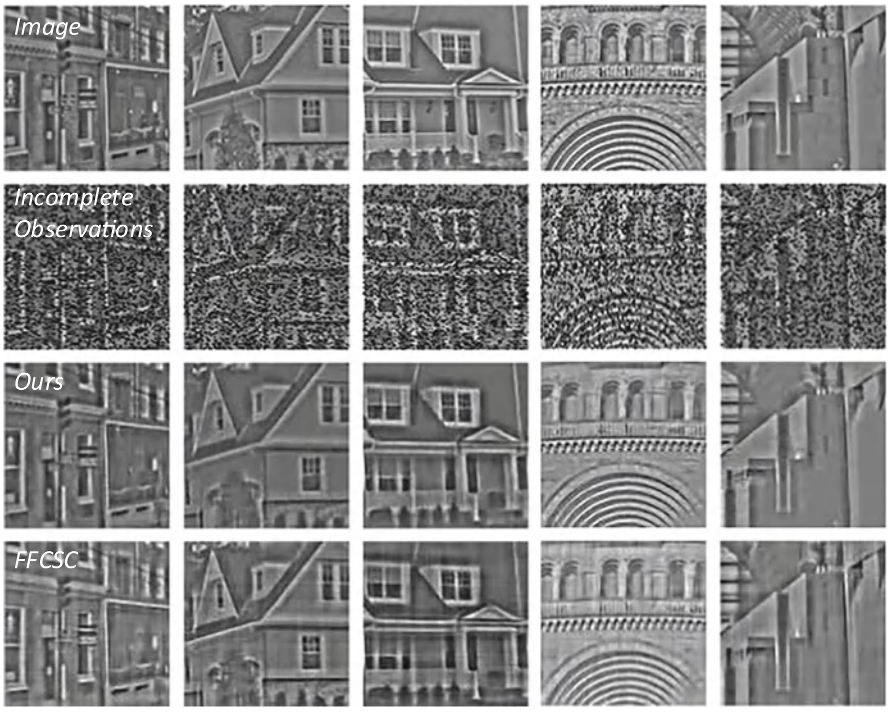


Fig. 5.13 Inpainting for the normalized city dataset. From top to bottom: the original images, incomplete observations, reconstructions with FFCSC [24], and reconstructions with our proposed algorithm

compare with FFCSC, inpainting results on the fruit and city datasets are shown in Table 5.13. It can be seen that our method achieves a better PSNR and SSIM in all cases, while the average PSNR and SSIM improvements are impressively 1.78 and 0.017 db.

For image reconstruction, we reconstruct the images on the fruit and city datasets. One hundred of 11×11 filters are trained and compared with those of FFCSC [24]. Figure 5.14 shows the resulting filters after convergence within the same 20 iterations. It can be seen that the proposed reconstruction method, driven with CoGD, converges with a lower loss. When comparing the PSNR and the SSIM of our method with FFCSC in Table 5.14, we can see that in most cases, our method achieves higher PSNR and SSIM. The average PSNR and SSIM improvements are respectively 1.27 db and 0.005.

Considering that PSNR is calculated with a log function, the performance improvement shown in Tables 5.13 and 5.14 is significant. Such improvements show that the kernelized projection function improves the performance of the algorithm and reveal the nonlinear interaction of the variables.

Network Pruning We have evaluated the proposed CoGD algorithm on network pruning using the CIFAR-10 and ILSVRC12 ImageNet datasets for the image classification tasks. The commonly used ResNets and MobileNetV2 are used as the backbone networks to get the pruned network models.

Experimental Setting

Datasets CIFAR-10 is a natural image classification dataset containing a training set of 50,000 and a testing set of 10,000 32×32 color images distributed over ten classes, including airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The ImageNet classification dataset is more challenging due to the significant increase of image categories, image samples, and sample diversity. For the 1000 categories of images, there are 1.2 million images for training and 50k images for validation. The large data divergence set a ground challenge for the optimization algorithms when pruning network models.

Implementation We use PyTorch to implement our method with 3 NVIDIA TITAN V and 2 Tesla V100 GPUs. The weight decay and the momentum are set to 0.0002 and 0.9, respectively. The hyperparameter $\lambda_{\mathbf{m}}$ is selected through cross-validation in the range $[0.01, 0.1]$ for ResNet and MobileNetv2. The drop rate is set to 0.1. The other training parameters are described on a per experiment basis.

To better demonstrate our method, we denote CoGD-a as an approximated pruning rate of $(1 - a)\%$ for corresponding channels. a is associated with the threshold α_W, which is given by its sorted result. For example, if $a = 0.5$, α_W is the median of the sorted result. $\alpha_{\mathbf{m}}$ is set to be 0.5 for easy implementation. Similarly, $\beta = 0.001\eta_2\mathbf{c}^t$ with $\frac{\partial \mathbf{W}}{\partial m_j} \approx \frac{\Delta \mathbf{W}}{\Delta m_j}$. Note that our training cost is similar to that of [43], since we use our method once per epoch without additional cost.

Table 5.13 Inpainting results for convolutional filters learned with the proposed method and with FFCSC [24]. All reconstructions are performed with 75% data subsampling. The proposed CoGD achieves better PSNR and SSIM in all cases

Dataset	Fruit	1	2	3	4	5	6	7	8	9	10	Average
PSNR (dB)	[24]	25.37	24.31	25.08	24.27	23.09	25.51	22.74	24.10	19.47	22.58	23.65
	CoGD(kernelized, $k = 1$)	26.37	24.45	25.19	25.43	24.91	**27.90**	24.26	25.40	**24.70**	**24.46**	25.31
	CoGD(kernelized, $k = 2$)	27.93	**26.73**	27.19	25.25	**23.54**	25.02	**26.29**	24.12	24.48	24.04	25.47
	CoGD(kernelized, $k = 3$)	**28.85**	26.41	**27.35**	**25.68**	24.44	26.91	25.56	**25.46**	24.51	22.42	**25.76**
SSIM	[24]	0.9118	0.9036	0.9043	0.8975	0.8883	0.9242	0.8921	0.8899	0.8909	0.8974	0.9000
	CoGD(kernelized, $k = 1$)	0.9452	0.9217	**0.9348**	0.9114	**0.9036**	**0.9483**	0.9109	0.9041	**0.9215**	**0.9097**	**0.9211**
	CoGD(kernelized, $k = 2$)	0.9483	**0.9301**	0.9294	0.9061	0.8939	0.9454	**0.9245**	0.8990	0.9208	0.9054	0.9203
	CoGD(kernelized, $k = 3$)	**0.9490**	0.9222	0.9342	**0.9181**	0.8810	0.9464	0.9137	**0.9072**	0.9175	0.8782	0.9168
Dataset	City	1	2	3	4	5	6	7	8	9	10	Average
PSNR (dB)	[24]	26.55	24.48	25.45	21.82	24.29	25.65	19.11	25.52	22.67	27.51	24.31
	CoGD(kernelized, $k = 1$)	26.58	25.75	26.36	25.06	26.57	24.55	21.45	**26.13**	24.71	**28.66**	25.58
	CoGD(kernelized, $k = 2$)	**27.93**	**26.73**	**27.19**	25.83	24.41	25.31	**26.29**	24.70	24.48	24.62	**25.76**
	CoGD(kernelized, $k = 3$)	25.91	25.95	25.21	**26.26**	**26.63**	**27.68**	21.54	25.86	**24.74**	27.69	25.75

(continued)

Table 5.13 (continued)

Dataset	Fruit	1	2	3	4	5	6	7	8	9	10	Average
SSIM	[24]	0.9284	0.9204	0.9368	0.9056	0.9193	0.9202	0.9140	0.9258	0.9027	0.9261	0.9199
	CoGD(kernelized, $k = 1$)	0.9397	0.9269	**0.9433**	**0.9289**	0.9350	0.9217	**0.9411**	0.9298	0.9111	0.9365	0.9314
	CoGD(kernelized, $k = 2$)	**0.9498**	**0.9316**	0.9409	0.9176	0.9189	**0.9454**	0.9360	0.9305	**0.9323**	0.9284	0.9318
	CoGD(kernelized, $k = 3$)	0.9372	0.9291	0.9429	0.9254	**0.9361**	0.9333	0.9373	**0.9331**	0.9178	**0.9372**	**0.9329**

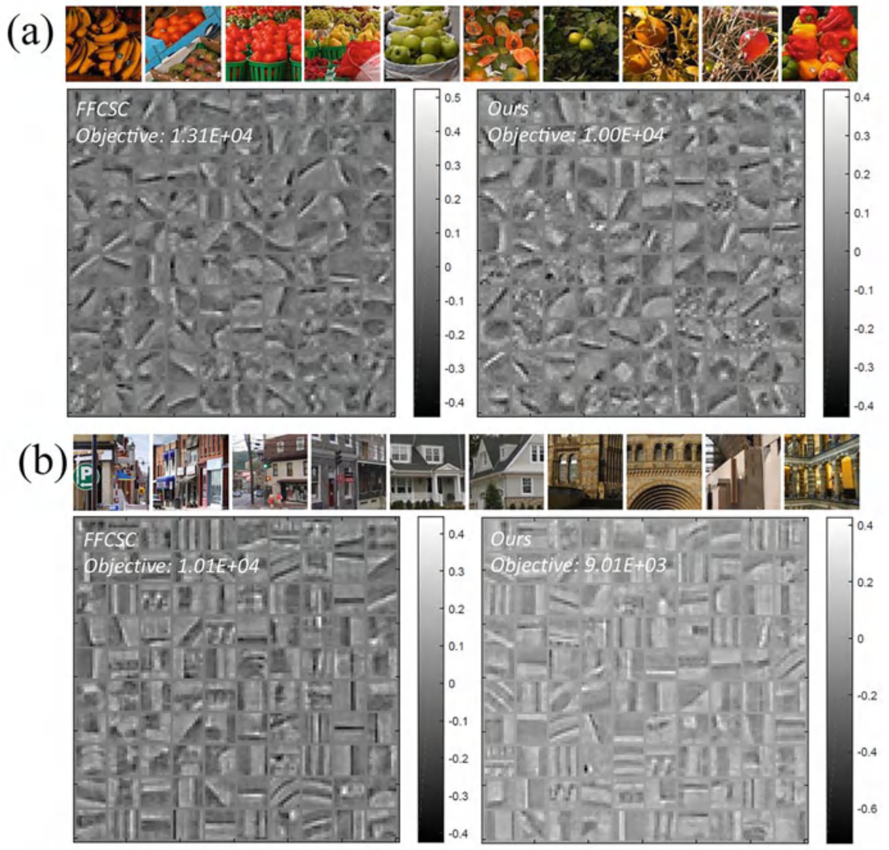


Fig. 5.14 Filters learned on fruit and city datasets. Thumbnails of the datasets along with filters learned with FFCSC [24] (left) and with CoGD (right) are shown. The proposed reconstruction method reports lower objectives. (Best viewed in color with zoom)

CIFAR-10

We evaluated the proposed network pruning method on CIFAR-10 for two popular networks, ResNets and MobileNetV2. The stage kernels are set to 64-128-256-512 for ResNet-18 and 16-32-64 for ResNet-110. For all networks, we add a soft mask only after the first convolutional layer within each block to simultaneously prune the output channel of the current convolutional layer and input channel of the next convolutional layer. The mini-batch size is set to be 128 for 100 epochs, and the initial learning rate is set to 0.01, scaled by 0.1 over 30 epochs.

Fine-tuning In the network fine-tuning after pruning, we only reserve the student model. According to the "zeros" in each soft mask, we remove the corresponding output channels of the current convolutional layer and corresponding input channels of the next convolutional layer. We then obtain a pruned network with fewer

Table 5.14 Reconstruction results for filters learned with the proposed method and with FFCSC [24]. With the exception of six images, the proposed method achieves better PSNR and SSIM

Dataset	Fruit	1	2	3	4	5	6	7	8	9	10	Average
PSNR (dB)	[24]	30.90	**29.52**	26.90	28.09	22.25	27.93	27.10	27.05	23.65	23.65	26.70
	CoGD(kernelized, $k = 1$)	**31.46**	29.12	27.26	**28.80**	25.21	27.35	26.25	**27.48**	**25.30**	**27.84**	27.60
	CoGD(kernelized, $k = 2$)	30.54	28.77	**30.33**	28.64	**25.72**	**30.31**	**28.07**	27.46	25.22	26.14	**28.12**
SSIM	[24]	0.9706	0.9651	0.9625	0.9629	0.9433	0.9712	0.9581	0.9524	0.9608	0.9546	0.9602
	CoGD(kernelized, $k = 1$)	**0.9731**	0.9648	0.9640	0.9607	**0.9566**	0.9717	0.9587	0.9562	0.9642	**0.9651**	0.9635
	CoGD(kernelized, $k = 2$)	0.9705	**0.9675**	**0.9660**	**0.9640**	0.9477	**0.9728**	**0.9592**	**0.9572**	**0.9648**	0.9642	**0.9679**
Dataset	City	1	2	3	4	5	6	7	8	9	10	Average
PSNR (dB)	[24]	30.11	27.86	**28.91**	26.70	27.85	28.62	18.63	28.14	27.20	25.81	26.98
	CoGD(kernelized, $k = 1$)	30.29	**28.77**	28.51	26.29	28.50	**30.36**	21.22	**29.07**	**27.45**	**30.54**	**28.10**
	CoGD(kernelized, $k = 2$)	**30.61**	28.57	27.37	**27.66**	**28.57**	29.87	**21.48**	27.08	26.82	29.86	27.79
SSIM	[24]	0.9704	**0.9660**	**0.9703**	0.9624	0.9619	0.9613	0.9459	0.9647	0.9531	0.9616	0.9618
	CoGD(kernelized, $k = 1$)	**0.9717**	**0.9660**	0.9702	**0.9628**	**0.9627**	**0.9624**	**0.9593**	**0.9663**	**0.9571**	**0.9632**	**0.9642**
	CoGD(kernelized, $k = 2$)	0.9697	0.9646	0.9681	0.962	0.9613	0.9594	0.9541	0.9607	0.9538	0.9620	0.9631

Table 5.15 Pruning results of ResNet-18/ResNet-110 and MobilenetV2 on CIFAR-10. M = million (10^6)

Model	FLOPs (M)	Reduction	Accuracy/+FT (%)
ResNet-18 [20]	555.42	–	95.31
CoGD-0.5	274.74	0.51×	95.11/95.30
CoGD-0.8	423.87	0.24×	95.19/**95.41**
ResNet-56 [20]	125.49	–	93.26
GAL-0.6 [43]	78.30	0.38×	92.98/93.38
GAL-0.8 [43]	49.99	0.60×	90.36/91.58
CoGD-0.5	48.90	0.61×	92.38/92.95
CoGD-0.8	82.85	0.34×	93.16/**93.59**
ResNet-110 [20]	252.89	–	93.68
GAL-0.1 [43]	205.70	0.20×	92.65/93.59
GAL-0.5 [43]	130.20	0.49×	92.65/92.74
CoGD-0.5	95.03	0.62×	93.31/93.45
CoGD-0.8	135.76	0.46×	93.42/93.66
MobileNet-V2 [54]	91.15	–	94.43
CoGD-0.5	50.10	0.45×	94.25/–

parameters and that requires fewer FLOPs. We use the same batch size of 256 for 60 epochs as in training. The initial learning rate is changed to 0.1 and scaled by 0.1 over 15 epochs. Note that a similar fine-tuning strategy was used in GAL [43].

Results Two kinds of networks are tested on the CIFAR-10 database – ResNets and MobileNet-V2. In this section, we only test the linear kernel, which achieves a similar performance as the full-precision model.

Results for **ResNets** are shown in Table 5.15. Compared to the pre-trained network for ResNet-18 with 95.31% accuracy, CoGD-0.5 achieves a 0.51× FLOP reduction with negligible accuracy drop 0.01%. Among other structured pruning methods for ResNet-110, CoGD-0.5 has a larger FLOP reduction than GAL-0.1 (95.03M vs. 205.70M), but with similar accuracy (93.45% vs. 93.59%). These results demonstrate that our method can prune the network efficiently and generate a more compressed model with higher performance.

For MobileNetV2, the pruning results are summarized in Table 5.15. Compared to the pre-trained network, CoGD-0.5 achieves a 0.45× FLOP reduction with a 0.18% accuracy drop. The results indicate that CoGD is easily employed on efficient networks with depth-wise separable convolution, which is worth exploring in practical applications.

ImageNet
For ILSVRC12 ImageNet, we test our CoGD based on ResNet-50. We train the network with a batch size of 256 for 60 epochs. The initial learning rate is set to 0.01 and scaled by 0.1 over 15 epochs. Other hyperparameters follow the settings

Table 5.16 Pruning results of ResNet-50 on ImageNet. B means billion (10^9)

Model	FLOPs (B)	Reduction	Accuracy/+FT (%)
ResNet-50 [20]	4.09	–	76.24
ThiNet-50 [46]	1.71	0.58×	71.01
ThiNet-30 [46]	1.10	0.73×	68.42
CP[23]	2.73	0.33×	72.30
GDP-0.5 [42]	1.57	0.62×	69.58
GDP-0.6 [42]	1.88	0.54×	71.19
SSS-26 [29]	2.33	0.43×	71.82
SSS-32 [29]	2.82	0.31×	74.18
RBP [75]	1.78	0.56×	71.50
RRBP [75]	1.86	0.55×	73.00
GAL-0.1 [43]	2.33	0.43×	–/71.95
GAL-0.5 [43]	1.58	0.61×	–/69.88
CoGD-0.5	2.67	0.35×	75.15/75.62

used on CIFAR-10. The fine-tuning process follows the setting on CIFAR-10 with the initial learning rate 0.00001.

Table 5.16 shows that CoGD achieves state-of-the-art performance on the ILSVRC12 ImageNet. For ResNet-50, CoGD-0.5 further shows a 0.35× FLOP reduction while achieving only a 0.62% drop in accuracy.

5.4.4.6 Ablation Study

We use ResNet-18 on CIFAR-10 for an ablation study to evaluate the effectiveness of our method.

Effect on CoGD We train the pruned network with and without CoGD by using the same parameters. As shown in Table 5.17, we obtain an error rate of 4.70% and a 0.51× FLOP reduction with CoGD, compared to the error rate of 5.19% and a 0.32× FLOP reduction without CoGD, validating the effectiveness of CoGD.

Synchronous convergence In Fig. 5.15, the training curve shows that the convergence of CoGD is similar to that of GAL with SGD-based optimization within an epoch, especially for the last epochs when converging in a similar speed. We theoretically derive CoGD within the gradient descent framework, which provides a theoretical foundation for the convergence, which is validated by the experiments. As a summary, the main differences between SGD and CoGD are twofold. First, we change the initial point for each epoch. Second, we explore the coupling relationship between the hidden factors to improve a bilinear model within the gradient descent framework. Such differences do not change the convergence of CoGD compared with the SGD method.

Table 5.17 Pruning results on CIFAR-10 with CoGD or SGD. M means million (10^6)

Optimizer	Accuracy (%)	FLOPs/Baseline (M)
SGD	94.81	376.12/555.43
CoGD	95.30	274.74/555.43

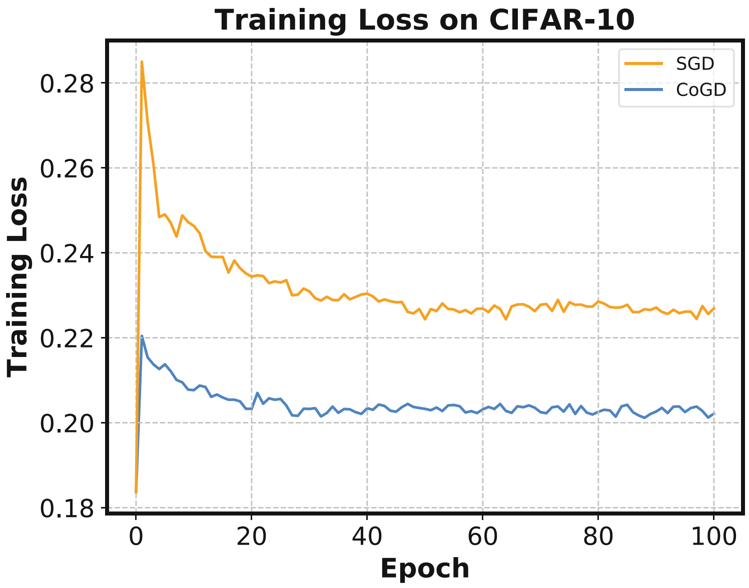
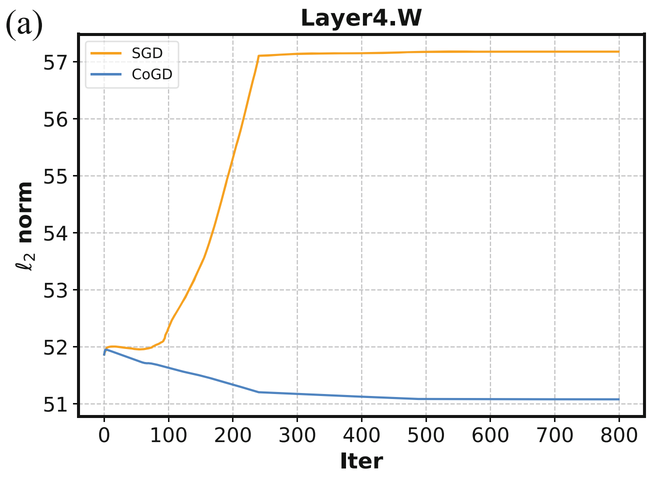


Fig. 5.15 Comparison of training loss on CIFAR-10 with CoGD and SGD

In Fig. 5.16, we show the convergence in a synchronous manner of the 4th layer's variables when pruning CNNs. For better visualization, the learning rate of **m** is enlarged by $100x$. On the curves, we observe that the two variables converge synchronously and that neither variable gets stuck into a local minimum. This validates that CoGD avoids vanishing gradient for the coupled variables.

CNN Training

Similar to network pruning, we have further evaluated CoGD algorithm for CNN model training on CIFAR-10 and ILSVRC12 ImageNet datasets. Specifically, we use ResNet-18 as the backbone CNN to test our algorithm. The network stages are 64-128-256-512. The learning rate is optimized by a cosine updating schedule with an initial learning rate 0.1. The algorithm iterates 200 epochs. The weight decay and momentum are respectively set to 0.0001 and 0.9. The model is trained on 2 GPUs (Titan V) with a mini-batch size of 128. We follow the similar augmentation strategy in [20] and add the cutout method for training. When training the model, horizontal flipping and 32×32 crop are used as data augmentation. The cutout size is set to 16. Similar to CNN pruning, a is set to 0.95 to compute α_γ and α_W. To improve the efficiency, we directly backtrack the corresponding weights. For



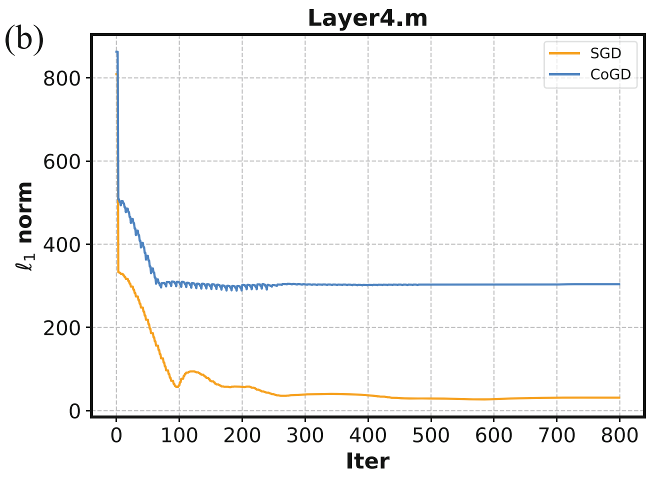


Fig. 5.16 Convergence comparison of the variables in the fourth convolutional layer when pruning CNNs. The curves are obtained using SGD and CoGD-0.5 on CIFAR-10. With CoGD, the two variables converge synchronously while avoiding either variable gets stuck in a local convergence (local minimum of the objective function), which validates that CoGD can avoid vanishing gradient for the coupled variables

ILSVRC12 ImageNet, the initial learning rate is set to 0.01, and the total epochs are 120.

With ResNet-18 backbone, we simply replace the SGD algorithm with the proposed CoGD for model training. In Table 5.18, it can be seen that the performance is improved by 1.25% (70.75% vs. 69.50%) on the large-scale ImageNet dataset. In addition, the improvement is also observed on CIFAR-10. We report the results for different kernels, which show that the performance are relatively stable for $k = 1$

Table 5.18 Results for CNN training on CIFAR-10 and ImageNet

Models	Accuracy(%)	
	CIFAR-10	ImageNet
ResNet-18 (SGD)[20]	95.31	69.50
ResNet-18 (CoGD with k = 1)	95.80	70.30
ResNet-18 (CoGD with k = 2)	96.10	70.75

and $k = 2$. These results validate the effectiveness and generality of the proposed CoGD algorithm.

5.5 Network Pruning on BNNs

5.5.1 Rectified Binary Convolutional Networks with Generative Adversarial Learning

Quantization techniques involve representing network weights and activations using low-bit fixed-point integers, enabling efficient computation with bitwise operations.

Binarization, as proposed in [45, 53], takes quantization to the extreme by using only a single bit to represent both weights and activations, where they are assigned values of either $+1$ or -1. This work focuses on creating compact binary neural networks (BNNs) by combining quantization and network pruning strategies.

Despite advancements in 1-bit quantization and network pruning, only a few studies have merged these methods into a cohesive framework to enhance their synergy. Introducing pruning techniques into 1-bit CNNs becomes necessary because not all filters and kernels have equal significance or warrant identical quantization. One potential solution involves pruning the network first and then applying 1-bit quantization to the remaining weights, resulting in a more compressed network. However, this approach must consider the disparities between binarized and full-precision parameters during pruning. As a promising alternative, one can prune the quantized network directly. Nevertheless, devising a unified framework to combine quantization and pruning remains an open question.

To tackle these challenges, we propose a novel approach called rectified binary convolutional network (RBCN) [44] to train a binary neural network (BNN) using a generative adversarial network (GAN) framework. Our motivation stems from GANs' ability to match two data distributions, namely, the full-precision and 1-bit networks. This can be seen as distilling or exploiting the knowledge from the full-precision model to benefit its 1-bit counterpart.

In the training process of RBCN, the key binarization steps are depicted in Fig. 5.17. Here, the full-precision model and the 1-bit model (generator) generate "real" and "fake" feature maps, respectively, which are then fed to the discriminators. The discriminators aim to distinguish between the "real" and "fake" samples, while the generator attempts to deceive the discriminators. This process results

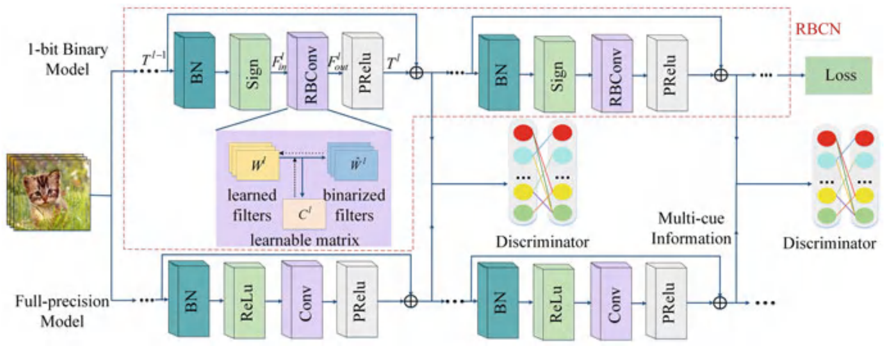


Fig. 5.17 This figure shows the framework for integrating the rectified binary convolutional network (RBCN) with generative adversarial network (GAN) learning. The full-precision model provides "real" feature maps, while the 1-bit model (as a generator) provides "fake" feature maps to discriminators trying to distinguish "real" from "fake." Meanwhile, the generator tries to make the discriminators work improperly. When this process is repeated, both the full-precision feature maps and kernels (across all convolutional layers) are sufficiently employed to enhance the capacity of the 1-bit model. Note that (1) the full-precision model is used only in learning but not in inference; (2) after training, the full-precision learned filters W are discarded, and only the binarized filters $\hat{W}$ and the shared learnable matrices C are kept in RBCN for the calculation of the feature maps in inference

in a rectified operation and a unique architecture that provides a more accurate estimation of the full-precision model.

Furthermore, we explore the application of pruning to enhance the practical usability of the 1-bit model within the GAN framework. To achieve this goal, we seamlessly integrate quantization and pruning into a unified framework.

5.5.1.1 Loss Function

The rectification process involves combining full-precision kernels and feature maps to improve the binarization process. It includes two main components: kernel approximation and adversarial learning.

The learnable kernel approximation results in a unique architecture that provides a precise estimation of the convolutional filters by minimizing the kernel loss. This allows the RBCN to achieve better performance and more accurate representations.

To accomplish this, discriminators denoted as $D(\cdot)$ with filters Y are introduced. Their purpose is to distinguish between feature maps R obtained from the full-precision model and feature maps T generated by the RBCN. The RBCN generator, equipped with filters W and matrices C, is trained using knowledge from the supervised feature maps R.

In summary, the optimization problem involves learning the parameters W, C, and Y by solving the following optimization problem:

$$\arg \min_{W,\hat{W},C} \max_{Y} \mathscr{L} = \mathscr{L}_{Adv}(W, \hat{W}, C, Y) + \mathscr{L}_{S}(W, \hat{W}, C) + \mathscr{L}_{Kernel}(W, \hat{W}, C), \tag{5.85}$$

where $\mathscr{L}_{Adv}(W, \hat{W}, C, Y)$ is the adversarial loss as:

$$\mathscr{L}_{Adv}(W, \hat{W}, C, Y) = log(D(R; Y)) + log(1 - D(T; Y)), \tag{5.86}$$

where $D(\cdot)$ consists of a series of basic blocks, each containing linear and LeakyRelu layers. We also have multiple discriminators to rectify the binarization training process.

In addition, $\mathscr{L}_{Kernel}(W, \hat{W}, C)$ denotes the kernel loss between the learned full-precision filters W and the binarized filters $\hat{W}$ and is defined as:

$$\mathscr{L}_{Kernel}(W, \hat{W}, C) = \lambda_1/2||W - C\hat{W}||^2, \tag{5.87}$$

where λ_1 is a balance parameter. Finally, $\mathscr{L}_S$ is a traditional problem-dependent loss, such as softmax loss. The adversarial, kernel, and softmax loss are regularizations on $\mathscr{L}$.

We also have omitted $log(\cdot)$ and rewritten the optimization in Eq. 5.85 as in Eq. 5.88 for simplicity:

$$\min_{W,\hat{W},C} \mathscr{L}_S(W, \hat{W}, C) + \lambda_1/2 \sum_l \sum_i ||W_i^l - C^l \hat{W}_i^l||^2 + \sum_l \sum_i ||1 - D(T_i^l; Y)||^2. \tag{5.88}$$

where i represents the ith channel and l represents the lth layer. In Eq. 5.88, the objective is to obtain W, $\hat{W}$ and C with Y fixed, which is why the term $D(R; Y)$ in Eq. 5.85 can be ignored. The advantage of our formulation in Eq. 5.88 lies in that the loss function is trainable, which means it can be easily incorporated into existing learning frameworks.

5.5.1.2 Learning RBCNs

In RBCNs, convolution is implemented using W^l, C^l, and F_{in}^l to calculate output feature maps F_{out}^l as:

$$F_{out}^l = RBConv(F_{in}^l; \hat{W}^l, C^l) = Conv(F_{in}^l, \hat{W}^l \odot C^l), \tag{5.89}$$

where $RBConv$ denotes the convolution operation implemented as a new module, F_{in}^l and F_{out}^l are the feature maps before and after convolution, respectively. W^l are full-precision filters, the values of $\hat{W}^l$ are 1 or -1, and $\odot$ is the operation of the element-by-element product.

During the backward propagation process of RBCNs, the full-precision filters W and the learnable matrices C must be learned and updated. These two sets of parameters are jointly learned. We update W first and then C in each convolutional layer.

Update W Let $\delta_{W_i^l}$ be the gradient of the full-precision filter W_i^l. During back propagation, the gradients are first passed to $\hat{W}_i^l$ and then to W_i^l. Thus:

$$\delta_{W_i^l} = \frac{\partial \mathcal{L}}{\partial W_i^l} = \frac{\partial \mathcal{L}}{\partial \hat{W}_i^l} \frac{\partial \hat{W}_i^l}{\partial W_i^l}, \tag{5.90}$$

where

$$\frac{\partial \hat{W}_i^l}{\partial W_i^l} = \begin{cases} 2 + 2W_i^l, & -1 \le W_i^l < 0, \\ 2 - 2W_i^l, & 0 \le W_i^l < 1, \\ 0, & \text{otherwise}, \end{cases} \tag{5.91}$$

which is an approximation of $2\times$ the Dirac delta function [45]. Furthermore:

$$\frac{\partial \mathcal{L}}{\partial \hat{W}_i^l} = \frac{\partial \mathcal{L}_S}{\partial \hat{W}_i^l} + \frac{\partial \mathcal{L}_{Kernel}}{\partial \hat{W}_i^l} + \frac{\partial \mathcal{L}_{Adv}}{\partial \hat{W}_i^l}, \tag{5.92}$$

and:

$$W_i^l \leftarrow W_i^l - \eta_1 \delta_{W_i^l}, \tag{5.93}$$

where η_1 is the learning rate. Then:

$$\frac{\partial \mathcal{L}_{Kernel}}{\partial \hat{W}_i^l} = -\lambda_1 (W_i^l - C^l \hat{W}_i^l) C^l, \tag{5.94}$$

$$\frac{\partial \mathcal{L}_{Adv}}{\partial \hat{W}_i^l} = -2(1 - D(T_i^l; Y)) \frac{\partial D}{\partial \hat{W}_i^l}. \tag{5.95}$$

Update C We further update the learnable matrix C^l with W^l fixed. Let δ_{C^l} be the gradient of C^l. Then we have:

$$\delta_{C^l} = \frac{\partial \mathcal{L}_S}{\partial C^l} + \frac{\partial \mathcal{L}_{Kernel}}{\partial C^l} + \frac{\partial \mathcal{L}_{Adv}}{\partial C^l}, \tag{5.96}$$

and:

$$C^l \leftarrow C^l - \eta_2 \delta_{C^l}, \tag{5.97}$$

where η_2 is another learning rate. Furthermore:

$$\frac{\partial \mathscr{L}_{Kernel}}{\partial C^l} = -\lambda_1 \sum_i (W_i^l - C^l \hat{W}_i^l)\hat{W}_i^l, \tag{5.98}$$

$$\frac{\partial \mathscr{L}_{Adv}}{\partial C^l} = -\sum_i 2(1 - D(T_i^l; Y))\frac{\partial D}{\partial C^l}. \tag{5.99}$$

The derivations presented demonstrate that the rectified process is trainable in an end-to-end manner. During training, we update the other parameters independently while keeping the convolutional layer's parameters fixed. This approach helps to enhance the variety of feature maps in each layer, which accelerates training convergence and fully explores the potential of 1-bit networks.

In our implementation, we replace all the values of C^l with their average during the forward process. This simplification reduces the matrix calculations to scalar operations, leading to faster computation during inference. By utilizing this approach, we achieve a significant speedup in the model's execution without compromising its performance.

5.5.1.3 Network Pruning

After binarizing the CNNs, we further prune the resulting 1-bit CNNs to increase model efficiency and improve the flexibility of RBCNs in practical scenarios. The optimization pruning process is performed under the generative adversarial learning framework using the method described in [43].

To achieve this, we employ a soft mask that allows us to remove specific structures, such as filters, while maintaining performance close to the baseline accuracy. The discriminator $D_p(\cdot)$ with weights Y_p is introduced to distinguish between the output of the baseline network R_p and that of the pruned 1-bit network T_p.

The pruned network is denoted by parameters W_p, $\hat{W}_p$, C_p, and a soft mask M_p. These parameters are learned together with Y_p using knowledge from the supervised features of the baseline network.

We jointly optimize the parameters W_p, $\hat{W}_p$, C_p, M_p, and Y_p by solving the following optimization problem:

$$\begin{aligned} \arg \min_{W_p,\hat{W}_p,C_p,M_p} \max_{Y_p} \mathscr{L}_p &= \mathscr{L}_{Adv\_p}(W_p, \hat{W}_p, C_p, M_p, Y_p) \\ &+ \mathscr{L}_{Kernel\_p}(W_p, \hat{W}_p, C_p) \\ \mathscr{L}_{S\_p}(W_p, \hat{W}_p, C_p) &+ \mathscr{L}_{Data\_p}(W_p, \hat{W}_p, C_p, M_p) + \mathscr{L}_{Reg\_p}(M_p, Y_p), \end{aligned} \tag{5.100}$$

where $\mathscr{L}_p$ is the pruning loss function, and the forms of $\mathscr{L}_{Adv\_p}(W_p, \hat{W}_p, C_p, M_p, Y_p)$ and $\mathscr{L}_{Kernel\_p}(W_p, \hat{W}_p, C_p)$ are:

$$\mathscr{L}_{Adv\_p}(W_p, \hat{W}_p, C_p, M_p, Y_p) = log(D_p(R_p; Y_p)) + log(1 - D_p(T_p; Y_p)), \tag{5.101}$$

$$\mathscr{L}_{Kernel\_p}(W_p, \hat{W}_p, C_p) = \lambda_1/2||W_p - C_p\hat{W}_p||^2. \tag{5.102}$$

$\mathscr{L}_{S\_p}$ is a traditional problem-dependent loss such as softmax loss. $\mathscr{L}_{Data\_p}$ is the data loss between the output features of the baseline and the pruned network and is used to align the output of these two networks. The data loss can then be expressed as the MSE loss:

$$\mathscr{L}_{Data\_p}(W_p, \hat{W}_p, C_p, M_p) = \frac{1}{2n}\left\| R_p - T_p \right\|^2, \tag{5.103}$$

where n is the size of the mini-batch.

$\mathscr{L}_{Reg\_p}(M_p, Y_p)$ is a regularizer on W_p,$\hat{W}_p$,M_p, and Y_p, which can be split into two parts as follows:

$$\mathscr{L}_{Reg\_p}(M_p, Y_p) = \mathscr{R}_\lambda(M_p) + \mathscr{R}(Y_p), \tag{5.104}$$

where $\mathscr{R}(Y_p) = log(D_p(T_p; Y_p))$, $\mathscr{R}_\lambda(M_p)$ is a sparsity regularizer form with parameters λ and $\mathscr{R}_\lambda(M_p) = \lambda||M_p||_{l_1}$.

As with the process in binarization, the update of the discriminators is omitted in the following description. We have also omitted $log(\cdot)$ for simplicity and rewritten the optimization of Eq. 5.100 as:

$$\begin{aligned} \min_{W_p,\hat{W}_p,C_p,M_p} & \lambda_1/2 \sum_l \sum_i ||W^l_{p,i} - C^l \hat{W}^l_{p,i}||^2 + \sum_l \sum_i ||1 - D(T^l_{p,i}; Y)||^2 \\ & + \mathscr{L}_{S\_p}(W_p, \hat{W}_p, C_p) + \frac{1}{2n}\left\| R_p - T_p \right\|^2 + \lambda||M_p||_{l_1}. \end{aligned} \tag{5.105}$$

5.5.1.4 Learning Pruned RBCNs

In pruned RBCNs, the convolution is implemented as:

$$F^l_{out,p} = RBConv(F^l_{in,p}; \hat{W}^l_p \circ M^l_p, C^l_p) = Conv(F^l_{in,p}, (\hat{W}_p \circ M^l_p) \odot C^l_p), \tag{5.106}$$

where $\circ$ is an operator that obtains the pruned weight with mask M_p. The other part of the forward propagation in the pruned RBCNs is the same as in the RBCNs.

In pruned RBCNs, what needs to be learned and updated are full-precision filters W_p, learnable matrices C_p, and soft mask M_p. In each convolutional layer, these three sets of parameters are jointly learned.

Update M_p M_p is updated by FISTA [42] with the initialization of $\alpha_{(1)} = 1$. Then we obtain the following:

$$\alpha_{(k+1)} = \frac{1}{2}(1 + \sqrt{1 + 4\alpha^2_{(k)}}), \tag{5.107}$$

$$y_{(k+1)} = M_{p,(k)} + \frac{a_{(k)} - 1}{a_{(k+1)}}(M_{p,(k)} - M_{p,(k-1)}), \tag{5.108}$$

$$M_{p,(k+1)} = prox_{\eta(k+1)\lambda\|\cdot\|_1}(y_{(k+1)} - \eta_{k+1}\frac{\partial(\mathcal{L}_{Adv\_p} + \mathcal{L}_{Data\_p})}{\partial(y_{(k+1)})}), \tag{5.109}$$

where η_{k+1} is the learning rate in iteration $k+1$ and $prox_{\eta(k+1)\lambda\|\cdot\|_1}(z_i) = sign(z_i) \cdot (|z_i| - \eta_0\lambda)_+$; more details can be found in [43].

Update W_p Let $\delta_{W^l_{p,i}}$ be the gradient of the full-precision filter $W^l_{p,i}$. During back propagation, the gradients pass to $\hat{W}^l_{p,i}$ first and then to $W^l_{p,i}$. Furthermore:

$$\delta_{W^l_{p,i}} = \frac{\partial\mathcal{L}_p}{\partial\hat{W}^l_{p,i}} = \frac{\partial\mathcal{L}_{S\_p}}{\partial\hat{W}^l_{p,i}} + \frac{\partial\mathcal{L}_{Adv\_p}}{\partial\hat{W}^l_{p,i}} + \frac{\partial\mathcal{L}_{Kernel\_p}}{\partial\hat{W}^l_{p,i}} + \frac{\partial\mathcal{L}_{Data\_p}}{\partial\hat{W}^l_{p,i}}, \tag{5.110}$$

and:

$$W^l_{p,i} \leftarrow W^l_{p,i} - \eta_{p,1}\delta_{W^l_{p,i}}, \tag{5.111}$$

where $\eta_{p,1}$ is the learning rate, $\frac{\partial\mathcal{L}_{Kernel\_p}}{\partial\hat{W}^l_{p,i}}$ and $\frac{\partial\mathcal{L}_{Adv\_p}}{\partial\hat{W}^l_{p,i}}$ are:

$$\frac{\partial\mathcal{L}_{Kernel\_p}}{\partial\hat{W}^l_{p,i}} = -\lambda_1(W^l_{p,i} - C^l_p\hat{W}^l_{p,i})C^l_p, \tag{5.112}$$

$$\frac{\partial \mathcal{L}_{Adv\_p}}{\partial \hat{W}^l_{p,i}} = -2(1 - D(T^l_{p,i}; Y_p))\frac{\partial D_p}{\partial \hat{W}^l_{p,i}}. \tag{5.113}$$

And:

$$\frac{\partial \mathcal{L}_{Data\_p}}{\partial \hat{W}^l_{p,i}} = -\frac{1}{n}(R_p - T_p)\frac{\partial T_p}{\partial \hat{W}^l_{p,i}}, \tag{5.114}$$

Update C_p We further update the learnable matrix C^l_p with W^l_p and M^l_p fixed. Let $\delta_{C^l_p}$ be the gradient of C^l_p. Then we have:

$$\delta_{C^l_p} = \frac{\partial \mathcal{L}_p}{\partial \hat{C}^l_p} = \frac{\partial \mathcal{L}_{S\_p}}{\partial \hat{C}^l_p} + \frac{\partial \mathcal{L}_{Adv\_p}}{\partial \hat{C}^l_p} + \frac{\partial \mathcal{L}_{Kernel\_p}}{\partial \hat{C}^l_p} + \frac{\partial \mathcal{L}_{Data\_p}}{\partial \hat{C}^l_p}, \tag{5.115}$$

and:

$$C^l_p \leftarrow C^l_p - \eta_{p,2}\delta_{C^l_p}. \tag{5.116}$$

and $\frac{\partial \mathcal{L}_{Kernel\_p}}{\partial C^l_p}$ and $\frac{\partial \mathcal{L}_{Adv\_p}}{\partial C^l_p}$ are:

$$\frac{\partial \mathcal{L}_{Kernel\_p}}{\partial C^l_p} = -\lambda_1 \sum_i (W^l_{p,i} - C^l_p \hat{W}^l_{p,i})\hat{W}^l_{p,i}, \tag{5.117}$$

$$\frac{\partial \mathcal{L}_{Adv\_p}}{\partial C^l_p} = -\sum_i 2(1 - D_p(T^l_{p,i}; Y_p))\frac{\partial D_p}{\partial C^l_p}. \tag{5.118}$$

Furthermore:

$$\frac{\partial \mathcal{L}_{Data\_p}}{\partial C^l_p} = \frac{1}{n}\sum_i (R_p - T_p)\frac{\partial T_p}{\partial C^l_p}. \tag{5.119}$$

The complete training process is summarized in Algorithm 10, including the update of the discriminators.

5.5.1.5 Ablation Study

This section investigates the contributions of kernel approximation, GAN, and the update strategy in improving the performance of RBCNs, using CIFAR-100 dataset and ResNet-18 with different kernel stages.

Algorithm 10: Pruned RBCN

```
Input: The training dataset, the pre-trained 1-bit CNNs model, the feature maps $R_p$ from the
    pre-trained model, the pruning rate, and the hyper-parameters, including the initial learning
    rate, weight decay, convolution stride, and padding size.
Output: The pruned RBCN with updated parameters $W_p$, $\hat{W}_p$, $M_p$ and $C_p$.
 1: repeat
 2:     Randomly sample a mini-batch;
 3:     // Forward propagation
 4:     Training a pruned architecture // Using Eqs. 5.17–5.22
 5:     for all $l = 1$ to $L$ convolutional layer do
 6:         $F^l_{out,p} = Conv(F^l_{in,p}, (\hat{W}^l_p \circ M_p) \odot C^l_p)$;
 7:     end for
 8:     // Backward propagation
 9:     for all $l = L$ to 1 do
10:         Update the discriminators $D^l_p(\cdot)$ by ascending their stochastic gradients:
11:         $\nabla_{D^l_p}(log(D^l_p(R^l_p; Y_p)) + log(1 - D^l_p(T^l_p; Y_p)) + log(D^l_p(T_p; Y_p)))$;
12:         Update soft mask $M_p$ by FISTA // Using Eqs. 5.24–5.26
13:         Calculate the gradients $\delta_{W^l_p}$; // Using Eqs. 5.27–5.31
14:         $W^l_p \leftarrow W^l_p - \eta_{p,1}\delta_{W^l_p}$; // Update the weights
15:         Calculate the gradient $\delta_{C^l_p}$; // Using Eqs. 5.32–5.36
16:         $C^l_p \leftarrow C^l_p - \eta_{p,2}\delta_{C^l_p}$; // Update the learnable matrix
17:     end for
18: until the maximum epoch
19: $\hat{W} = \text{sign}(W)$.
```

Table 5.19 Performance (accuracy, %) contributions of the components in RBCNs on CIFAR-100, where Bi = Bi-Real Net, R = $RBConv$, G = GAN, and B = update strategy. The numbers in bold represent the best results

	Kernel stage	Bi	R	R+G	R+G+B
RBCN	32-32-64-128	54.92	56.54	59.13	**61.64**
RBCN	32-64-128-256	63.11	63.49	64.93	**65.38**
RBCN	64-64-128-256	63.81	64.13	65.02	**66.27**

1. We replace the convolution in Bi-Real Net with our kernel approximation ($RBConv$) and compare the results. The comparison is shown in the "Bi" and "R" columns of Table 5.19. RBCN achieves an accuracy improvement of 1.62% over Bi-Real Net (56.54% vs. 54.92%) using the same network structure as in ResNet-18. This substantial improvement validates the effectiveness of the learnable matrices.
2. Incorporating GAN into RBCN results in a further performance boost of 2.59% (59.13% vs. 56.54%) with the kernel stage of 32-32-64-128. This demonstrates that GAN helps to mitigate the problem of getting stuck in poor local minima during training, leading to better overall performance.
3. We enhance RBCNs by updating the batch normalization (BN) layers with fixed W and C after each epoch. This strategy further increases the accuracy by

2.51% (61.64% vs. 59.13%) in CIFAR-100 with 32-32-64-128 kernel stage. This improvement shows the effectiveness of the update strategy and its ability to contribute to the model's performance.

In summary, the kernel approximation, GAN, and the update strategy play crucial roles in enhancing the accuracy of RBCNs, and their combined effect results in significant improvements over the baseline Bi-Real Net, making RBCNs a powerful choice for image classification tasks on CIFAR-100 dataset.

5.5.2 BONN: Bayesian Optimized Binary Neural Network

Bayesian learning is a statistical modeling paradigm based on Bayes' theorem. It provides practical learning algorithms and facilitates understanding of other learning methods. Bayesian learning is particularly advantageous in solving probabilistic graphical models, enabling information exchange between perception and inference tasks, handling conditional dependencies in high-dimensional data, and effective uncertainty modeling. Bayesian neural networks (BayesNNs) have been extensively studied, with recent developments in their efficacy [4, 36, 41, 57].

Estimating the posterior distribution is essential in Bayesian inference as it represents uncertainties for both data and parameters. However, obtaining an exact analytical solution for the posterior distribution is challenging due to the large number of parameters in neural networks. To address this, various approaches have been proposed, including optimization-based techniques like variational inference (VI) and sampling-based methods such as Markov chain Monte Carlo (MCMC). MCMC provides sampling-based estimates of the posterior distribution but is computationally expensive for large datasets. On the other hand, VI tends to converge faster and has been applied to various Bayesian models, including BayesNNs [5, 58].

Despite the progress in 1-bit quantization and network pruning, few works have integrated both in a unified framework to enhance each other. However, introducing pruning techniques into 1-bit CNNs is crucial. Not all filters and kernels are equally important or suitable for quantization, as verified in subsequent experiments.

One potential approach is to perform pruning first, removing less important filters or parameters from the network, and then apply 1-bit quantization to the remaining network to achieve further compression.

However, pruning a 1-bit CNN requires special considerations due to the difference between binarized and full-precision parameters. While 1-bit CNNs tend to be more redundant before and after binarization, the pruning process must carefully account for the impact of quantization on the remaining parameters.

Alternatively, conducting pruning over Bayesian neural networks (BNNs) is a promising alternative. BNNs have been shown to provide better uncertainty modeling and representation ability, making them suitable candidates for pruning while preserving performance.

However, developing a unified framework to first calculate a 1-bit network and then prune it remains an open challenge. The representation ability of 1-bit networks may deteriorate due to quantization, affecting the backpropagation process and rendering existing optimization schemes ineffective.

To tackle the challenge of designing a unified framework for pruning 1-bit CNNs, Bayesian learning, a well-established global optimization scheme [5, 49], is leveraged to prune 1-bit CNNs [16].

The Bayesian learning approach begins by binarizing the full-precision kernels to two quantization values (centers), resulting in 1-bit CNNs. The quantization error is minimized by modeling the full-precision kernels as a Gaussian mixture, where each Gaussian is centered on its corresponding quantization value.

Using the two centers for 1-bit CNNs, a mixture model is constructed to represent the full-precision kernels. Subsequently, the Bayesian learning framework introduces a novel pruning operation for 1-bit CNNs. Filters are divided into two groups, with the assumption that filters within each group follow the same Gaussian distribution. The weights of filters in one group are then replaced with their average, effectively pruning the network and reducing its complexity.

The general framework for this approach is illustrated in Fig. 5.18, and it incorporates three innovative elements in the learning procedure of 1-bit CNNs with compression: (1) minimizing the reconstruction error of the parameters before and after quantization, (2) modeling the parameter distribution as a Gaussian mixture with two modes centered on the binarized values, and (3) pruning the quantized network by maximizing a posterior probability.

Further analysis leads to the development of three new losses and their corresponding learning algorithms, namely, the *Bayesian kernel loss*, *Bayesian feature loss*, and *Bayesian pruning loss*. These losses are jointly applied with the conventional cross-entropy loss within the same back propagation pipeline, inheriting the advantages of Bayesian learning during model quantization and pruning. Additionally, the proposed losses comprehensively supervise the 1-bit CNN training process concerning kernel and feature distributions.

In conclusion, the application of Bayesian learning in pruning 1-bit CNNs presents a promising direction for improving the compressed model's applicability in practical applications.

5.5.2.1 Bayesian Formulation for Compact 1-Bit CNNs

The state-of-the-art methods for learning 1-bit CNNs, such as [15, 39, 53], involve optimization in both continuous and discrete spaces. Training a 1-bit CNN typically comprises three steps: a forward pass (inference) using binarized weights ($\hat{\boldsymbol{x}}$), a backward pass involving gradient calculations, and a parameter update that leads to full-precision weights ($\boldsymbol{x}$).

The crucial factor influencing the performance of a quantized network, as demonstrated in [15, 39, 53], is how to connect the binarized weights $\hat{\boldsymbol{x}}$ with the full-precision weights $\boldsymbol{x}$. This connection determines the effectiveness of the

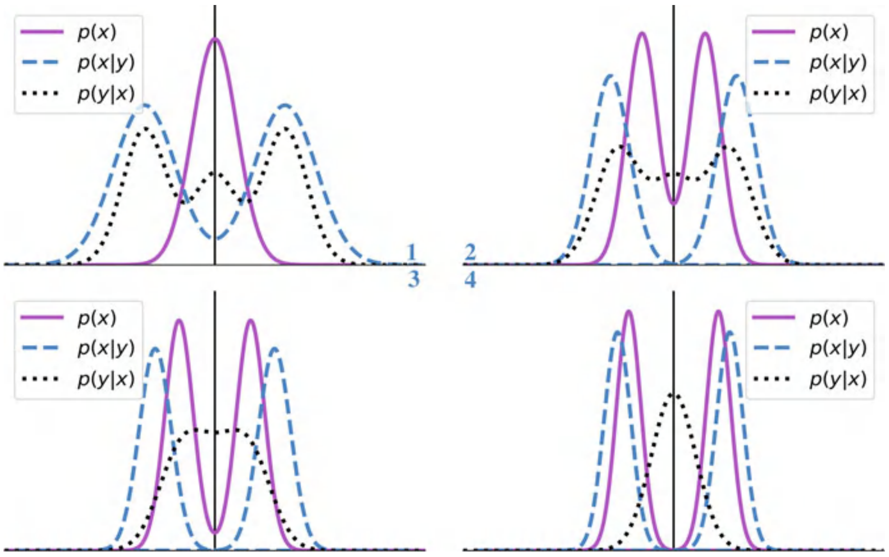


Fig. 5.18 The evolution of the prior $p(\boldsymbol{x})$, the distribution of the observation $\boldsymbol{y}$, and the posterior $p(\boldsymbol{x}|\boldsymbol{y})$ during learning, where $\boldsymbol{x}$ is the latent variable representing the full-precision parameters and $\boldsymbol{y}$ is the quantization error. Initially, the parameters $\boldsymbol{x}$ are initialized according to a single-mode Gaussian distribution. When our learning algorithm converges, the ideal case is that (i) $p(\boldsymbol{y})$ becomes a Gaussian distribution $\mathcal{N}(0, \nu)$, which corresponds to the minimum reconstruction error, and (ii) $p(\boldsymbol{x}|\boldsymbol{y}) = p(\boldsymbol{x})$ is a Gaussian mixture distribution with two modes where the binarized values $\hat{\boldsymbol{x}}$ and $-\hat{\boldsymbol{x}}$ are located

quantized model. In this paper, the authors propose to address this challenge using a probabilistic framework to learn optimal 1-bit CNNs.

5.5.2.2 Bayesian Learning Losses

Bayesian kernel loss Given a network weight parameter $\boldsymbol{x}$, its quantized code should be as close to its original (full-precision) code as possible so that the quantization error is minimized. We then define:

$$\boldsymbol{y} = \boldsymbol{w}^{\mathbf{-1}} \circ \hat{\boldsymbol{x}} - \boldsymbol{x}, \tag{5.120}$$

where $\boldsymbol{x}, \hat{\boldsymbol{x}} \in \mathbf{R}^n$ are the full-precision and quantized vectors, respectively, $\boldsymbol{w} \in \mathbf{R}^n$ denotes the learned vector to reconstruct $\boldsymbol{x}$, $\circ$ represents the Hadamard product, and $\boldsymbol{y} \sim G(0, \nu)$ is the reconstruction error that is assumed to obey a Gaussian prior with zero mean and variance ν. Under the most probable $\boldsymbol{y}$ (corresponding to $\boldsymbol{y} = \mathbf{0}$ and $\boldsymbol{x} = \boldsymbol{w}^{-1} \circ \hat{\boldsymbol{x}}$, i.e., the minimum reconstruction error), we maximize $p(\boldsymbol{x}|\boldsymbol{y})$ to optimize $\boldsymbol{x}$ for quantization (e.g., 1-bit CNNs) as:

$$\max p(\boldsymbol{x}|\boldsymbol{y}), \tag{5.121}$$

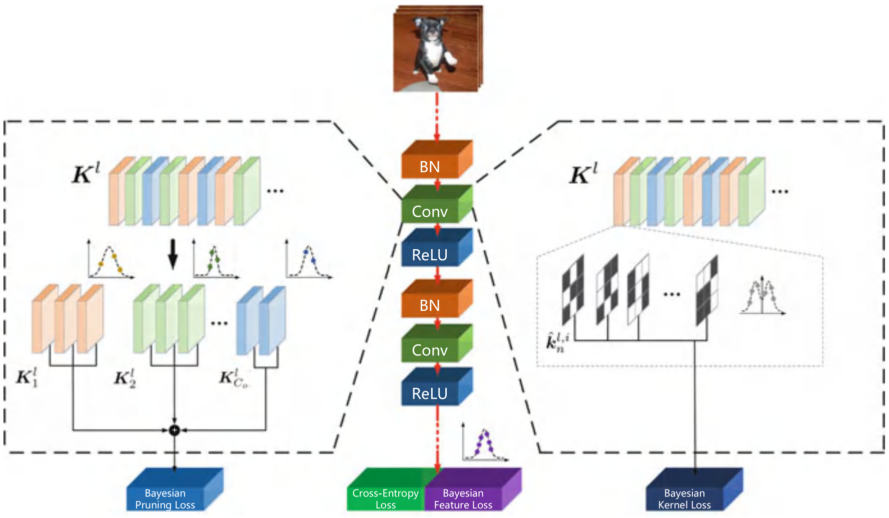


Fig. 5.19 By considering the prior distributions of the kernels and features in the Bayesian framework, we achieve three new Bayesian losses to optimize the 1-bit CNNs. The Bayesian kernel loss improves the layer-wise kernel distribution of each convolutional layer, the Bayesian feature loss introduces the intraclass compactness to alleviate the disturbance induced by the quantization process, and the Bayesian pruning loss centralizes channels following the same Gaussian distribution for pruning. The Bayesian feature loss is applied only to the fully connected layer

which can be solved based on Bayesian learning that uses Bayes' theorem to determine the conditional probability of a hypothesis given limited observations. We note that the calculation of BNNs is still based on optimizing $\boldsymbol{x}$, as shown in Fig. 5.19, where the binarization is performed based on the sign function. Equation 5.121 is complicated and difficult to solve due to the unknown w^{-1} as shown in Eq. 5.120. From a Bayesian learning perspective, we resolve this problem via maximum a posteriori (MAP):

$$\begin{aligned} \max p(\boldsymbol{x}|\boldsymbol{y}) &= \max p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x}) \\ &= \min ||\hat{\boldsymbol{x}} - \boldsymbol{w} \circ \boldsymbol{x}||_2^2 - 2\nu \log\big(p(\boldsymbol{x})\big), \end{aligned} \tag{5.122}$$

where

$$p(\boldsymbol{y}|\boldsymbol{x}) \propto \exp(-\frac{1}{2\nu}||\boldsymbol{y}||_2^2) \propto \exp(-\frac{1}{2\nu}||\hat{\boldsymbol{x}} - \boldsymbol{w} \circ \boldsymbol{x}||_2^2). \tag{5.123}$$

In Eq. 5.123, we assume that all components of the quantization error $\boldsymbol{y}$ are i.i.d., thus resulting in a simplified form. As shown in Fig. 5.19, for 1-bit CNNs, $\boldsymbol{x}$ is usually quantized to two numbers with the same absolute value. We neglect the overlap between the two numbers, and thus $p(\boldsymbol{x})$ is modeled as a Gaussian mixture with two modes:

$$
\begin{aligned}
p(\boldsymbol{x}) = & \frac{1}{2}(2\pi)^{-\frac{N}{2}} \det(\boldsymbol{\Psi})^{-\frac{1}{2}} \Big\{ \exp\big(-\frac{(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Psi}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}{2}\big) \\
& + \exp\big(-\frac{(\boldsymbol{x}+\boldsymbol{\mu})^T \boldsymbol{\Psi}^{-1}(\mathbf{x}+\boldsymbol{\mu})}{2}\big)\Big\} \\
\approx & \frac{1}{2}(2\pi)^{-\frac{N}{2}} \det(\boldsymbol{\Psi})^{-\frac{1}{2}} \Big\{ \exp\big(-\frac{(\boldsymbol{x}_+-\boldsymbol{\mu}_+)^T \boldsymbol{\Psi_+}^{-1}(\boldsymbol{x}_+-\boldsymbol{\mu}_+)}{2}\big) \\
& + \exp\big(-\frac{(\boldsymbol{x}_-+\boldsymbol{\mu}_-)^T \boldsymbol{\Psi_-}^{-1}(\boldsymbol{x}_-+\boldsymbol{\mu}_-)}{2}\big)\Big\},
\end{aligned}
\tag{5.124}
$$

where $\boldsymbol{x}$ is divided into $\boldsymbol{x}_+$ and $\boldsymbol{x}_-$ according to the signs of the elements in $\boldsymbol{x}$, and N is the dimension of $\boldsymbol{x}$. Accordingly, Eq. 5.122 can be rewritten as:

$$
\begin{aligned}
& \min ||\hat{\boldsymbol{x}} - \boldsymbol{w} \circ \boldsymbol{x}||_2^2 + \nu(\boldsymbol{x}_+ - \boldsymbol{\mu}_+)^T \boldsymbol{\Psi}_+^{-1}(\boldsymbol{x}_+ - \boldsymbol{\mu}_+) \\
& \quad + \nu(\boldsymbol{x}_- + \boldsymbol{\mu}_-)^T \boldsymbol{\Psi}_-^{-1}(\boldsymbol{x}_- + \boldsymbol{\mu}_-) + \nu \log\big(\det(\boldsymbol{\Psi})\big),
\end{aligned}
\tag{5.125}
$$

where $\boldsymbol{\mu}_-$ and $\boldsymbol{\mu}_+$ are solved independently. $\det(\boldsymbol{\Psi})$ is accordingly set to be the determinant of the matrix $\boldsymbol{\Psi}_-$ or $\boldsymbol{\Psi}_+$. We call Eq. 5.125 the Bayesian kernel loss.

Bayesian feature loss We also design a Bayesian feature loss to alleviate the disturbance caused by the extreme quantization process in 1-bit CNNs. Considering the intraclass compactness, the features $\boldsymbol{f}_m$ of the m-th class supposedly follow a Gaussian distribution with the mean $\boldsymbol{c}_m$ as revealed in the center loss [64]. Similarly to the Bayesian kernel loss, we define $\boldsymbol{y}_f^m = \boldsymbol{f}_m - \boldsymbol{c}_m$ and $\boldsymbol{y}_f^m \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\sigma}_m)$, and we have:

$$
\min ||\boldsymbol{f}_m - \boldsymbol{c}_m||_2^2 + \sum_{n=1}^{N_f} \Big[\sigma_{m,n}^{-2}(f_{m,n} - c_{m,n})^2 + \log(\sigma_{m,n}^2) \Big],
\tag{5.126}
$$

which is called the Bayesian feature loss. In Eq. 5.126, $\sigma_{m,n}$, $f_{m,n}$ and $c_{m,n}$ are the n-th elements of $\boldsymbol{\sigma}_m$, $\boldsymbol{f}_m$ and $\boldsymbol{c}_m$, respectively. We take the latent distributions of kernel weights and features into consideration in the same framework and introduce Bayesian losses to improve the capacity of 1-bit CNNs.

5.5.2.3 Bayesian Pruning

After binarizing CNNs, we prune the 1-bit CNNs using a Bayesian learning framework. The idea is to group similar channels together and then replace filters within each group with their average during optimization. The representation of the kernel weights of the l-th layer is a tensor $\boldsymbol{K}^l$ with dimensions $C_o^l \times C_i^l \times H^l \times W^l$, where C_o^l and C_i^l are the numbers of output and input channels, respectively, and H^l and W^l represent the height and width of the kernels.

To simplify the notation, we define $\boldsymbol{K}^l$ as a concatenation of individual filters $\boldsymbol{K}_i^l$ for $i = 1, 2, \ldots, C_o^l$, where $\boldsymbol{K}_i^l$ is a three-dimensional filter with dimensions $C_i^l \times H^l \times W^l$.

The pruning process begins by using the K-means algorithm to divide the filters into different groups based on similarity. The assumption is that filters within each group follow the same Gaussian distribution during training. The goal is to find the average $\overline{\boldsymbol{K}}$ that can replace all $\boldsymbol{K}_i$'s within the same group, which effectively assimilates similar filters into a single one.

This pruning problem leads to a similar formulation as in Eq. 5.122, and it involves learning the average filter $\overline{\boldsymbol{K}}$ with a Gaussian distribution constraint. This type of learning process with a Gaussian distribution constraint has been widely considered in other works [18].

Accordingly, Bayesian learning is used to prune 1-bit CNNs. We denote $\boldsymbol{\epsilon}$ as the difference between a filter and its mean, i.e., $\boldsymbol{\epsilon} = \boldsymbol{K} - \overline{\boldsymbol{K}}$, following a Gaussian distribution for simplicity. To calculate $\overline{\boldsymbol{K}}$, we minimize $\boldsymbol{\epsilon}$ based on MAP in our Bayesian framework, and we have:

$$\overline{\boldsymbol{K}} = \arg\max_{\boldsymbol{K}} p(\boldsymbol{K}|\boldsymbol{\epsilon}) = \arg\max_{\boldsymbol{K}} p(\boldsymbol{\epsilon}|\boldsymbol{K}) p(\boldsymbol{K}), \tag{5.127}$$

$$p(\boldsymbol{\epsilon}|\boldsymbol{K}) \propto \exp(-\frac{1}{2\nu}||\boldsymbol{\epsilon}||_2^2) \propto \exp(-\frac{1}{2\nu}||\boldsymbol{K} - \overline{\boldsymbol{K}}||_2^2), \tag{5.128}$$

and $p(\boldsymbol{K})$ is similar to Eq. 5.124 but with one mode. Thus, we have:

$$\begin{aligned} &\min||\boldsymbol{K} - \overline{\boldsymbol{K}}||_2^2 + \nu(\boldsymbol{K} - \overline{\boldsymbol{K}})^T \boldsymbol{\Psi}^{-1}(\boldsymbol{K} - \overline{\boldsymbol{K}}) \\ &\quad + \nu \log\big(\det(\boldsymbol{\Psi})\big), \end{aligned} \tag{5.129}$$

which is called the Bayesian pruning loss. In summary, the proposed Bayesian pruning approach is more general, assuming that similar kernels follow a Gaussian distribution and are represented by their centers for pruning. This results in a more flexible and suitable pruning method for binary neural networks compared to existing techniques. We introduce Bayesian losses and Bayesian pruning within the same framework, considering the latent distributions of kernel weights, features, and filters. This enhances the capacity of 1-bit CNNs and captures uncertainties, leading to improved performance. Experimental results demonstrate that the proposed Bayesian optimization-based neural networks (BONNs) outperform existing pruning methods.

5.5.2.4 BONNs

We employ the three Bayesian losses to optimize 1-bit CNNs, which form our Bayesian optimized 1-bit CNNs (BONNs). To do this, we reformulate the first two Bayesian losses for 1-bit CNNs as:

$$
\begin{aligned}
L_B = \frac{\lambda}{2} \sum_{l=1}^{L} \sum_{i=1}^{C_o^l} \sum_{n=1}^{C_i^l} \Big\{ & ||\hat{\boldsymbol{k}}_n^{l,i} - \boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i}||_2^2 \\
& + \nu(\boldsymbol{k}_n^{l,i}{}_+ - \boldsymbol{\mu}_{i+}^l)^T (\boldsymbol{\Psi}_{i+}^l)^{-1} (\boldsymbol{k}_n^{l,i}{}_+ - \boldsymbol{\mu}_{i+}^l) \\
& + \nu(\boldsymbol{k}_n^{l,i}{}_- + \boldsymbol{\mu}_{i-}^l)^T (\boldsymbol{\Psi}_{i-}^l)^{-1} (\boldsymbol{k}_n^{l,i}{}_- + \boldsymbol{\mu}_{i-}^l) \\
& + \nu \log(\det(\boldsymbol{\Psi}^l)) \Big\} + \frac{\theta}{2} \sum_{m=1}^{M} \Big\{ ||\boldsymbol{f}_m - \boldsymbol{c}_m||_2^2 \\
& + \sum_{n=1}^{N_f} \Big[\sigma_{m,n}^{-2} (f_{m,n} - c_{m,n})^2 + \log(\sigma_{m,n}^2) \Big] \Big\},
\end{aligned}
\tag{5.130}
$$

where $\boldsymbol{k}_n^{l,i}, l \in \{1, \ldots, L\}, i \in \{1, \ldots, C_o^l\}, n \in \{1, \ldots, C_i^l\}$, is the vectorization of the i-th kernel matrix at the l-th convolutional layer, $\boldsymbol{w}^l$ is a vector used to modulate $\boldsymbol{k}_n^{l,i}$, and $\boldsymbol{\mu}_i^l$ and $\boldsymbol{\Psi}_i^l$ are the mean and covariance of the i-th kernel vector at the l-th layer, respectively. And we term L_B the Bayesian optimization loss. Furthermore, we assume that the parameters in the same kernel are independent. Thus, $\boldsymbol{\Psi}_i^l$ becomes a diagonal matrix with the identical value $(\sigma_i^l)^2$, where $(\sigma_i^l)^2$ is the variance of the i-th kernel of the l-th layer.

In order to speed up the calculation of the inverse of $\boldsymbol{\Psi}_i^l$, all elements of $\boldsymbol{\mu}_i^l$ are made identical and equal to μ_i^l. Additionally, during the forward process in the implementation, all elements of $\boldsymbol{w}^l$ are replaced by their average. This optimization results in only a scalar instead of a matrix being involved in the inference, leading to significantly accelerated computation.

After training 1-bit CNNs, the Bayesian pruning loss L_P is utilized for the optimization of feature channels. The expression for L_P is given by:

$$
\begin{aligned}
L_P = \sum_{l=1}^{L} \sum_{j=1}^{J_l} \sum_{i=1}^{I_j} \Big\{ & ||\boldsymbol{K}_{i,j}^l - \overline{\boldsymbol{K}}_j^l||_2^2 \\
& + \nu(\boldsymbol{K}_{i,j}^l - \overline{\boldsymbol{K}}_j^l)^T (\boldsymbol{\Psi}_j^l)^{-1} (\boldsymbol{K}_{i,j}^l - \overline{\boldsymbol{K}}_j^l) + \nu \log\big(\det(\boldsymbol{\Psi}_j^l)\big) \Big\},
\end{aligned}
\tag{5.131}
$$

where J_l is the number of Gaussian clusters (groups) of the l-th layer, and $\boldsymbol{K}_{i,j}^l$ for $i = 1, 2, \ldots, I_j$, are the $\boldsymbol{K}_i^l$'s that belong to the j-th group. In the implementation,

J_l is defined as $\text{int}(C_o^l \times \epsilon)$, where ϵ is a predefined pruning rate, and one ϵ is used for all layers.

Notably, when the j-th Gaussian has only one sample $\boldsymbol{K}_{i,j}^l$, $\overline{\boldsymbol{K}}_j^l = \boldsymbol{K}_{i,j}^l$, and $\boldsymbol{\Psi}_j$ becomes a unit matrix.

In the BONN framework, the total loss L is a combination of three individual losses: the cross-entropy loss L_S, the Bayesian optimization loss L_B, and the Bayesian pruning loss L_P. The expression for the total loss is given as:

$$L = L_S + L_B + \zeta L_P, \tag{5.132}$$

where ζ is a hyperparameter that is set to 0 during binarization training and 1 during pruning. The Bayesian optimization loss L_B constrains the distribution of the convolution kernels to a symmetric Gaussian mixture with two modes. It ensures that the quantization error is minimized through the term $||\hat{\boldsymbol{k}}_n^{l,i} - \boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i}||_2^2$, where $\hat{\boldsymbol{k}}_n^{l,i}$ is the quantized kernel and $\boldsymbol{w}^l$ is the learnable vector used to reconstruct the full-precision kernel $\boldsymbol{k}_n^{l,i}$. The Bayesian feature loss works to modify the distribution of the features, reducing intraclass variation for improved classification performance. Finally, the Bayesian pruning loss drives the kernels toward their means, effectively compressing the 1-bit CNNs further by assimilating similar filters into single ones.

5.5.2.5 Forward Propagation

During forward propagation in BONNs, the binarized kernels and activations significantly accelerate the convolution computation. The reconstruction vector, denoted as $\boldsymbol{w}$ in Eq. 5.120, plays a crucial role in 1-bit CNNs. $\boldsymbol{w}^l$ becomes a scalar $\overline{w}^l$ in each layer, where $\overline{w}^l$ is the mean of $\boldsymbol{w}^l$ and is calculated online. The convolution process can be represented as:

$$\boldsymbol{O}^{l+1} = ((\overline{w}^l)^{-1}\hat{\boldsymbol{K}}^l) * \hat{\boldsymbol{O}}^l = (\overline{w}^l)^{-1}(\hat{\boldsymbol{K}}^l * \hat{\boldsymbol{O}}^l), \tag{5.133}$$

where $\hat{\boldsymbol{O}}^l$ denotes the binarized feature map of the l-th layer, and O^{l+1} is the feature map of the $(l+1)$-th layer. As shown in Eq. 5.133, the actual convolution is still binary, and O^{l+1} is obtained by simply multiplying $(\overline{w}^l)^{-1}$ and the binarization convolution. For each layer, only one floating-point multiplication is added, which is negligible for BONNs.

In addition, we consider the Gaussian distribution in the forward process of Bayesian pruning, which updates every filter in one group based on its mean. Specifically, we replace each filter $\boldsymbol{K}_{i,j}^l = (1-\gamma)\boldsymbol{K}_{i,j}^l + \gamma\overline{\boldsymbol{K}}_j^l$ during pruning.

Algorithm 11: Optimizing 1-bit CNNs with Bayesian learning

```
Input:
     The full-precision kernels $\boldsymbol{k}$, the reconstruction vector $\boldsymbol{w}$, the learning rate $\eta$,
     regularization parameters $\lambda$, $\theta$ and variance $\nu$, and the training dataset.
Output:
     The BONN with the updated $\boldsymbol{k}$, $\boldsymbol{w}$, $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$, $\boldsymbol{c}_m$, $\boldsymbol{\sigma}_m$.
 1: Initialize $\boldsymbol{k}$ and $\boldsymbol{w}$ randomly, and then estimate $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$ based on the average and variance of
    $\boldsymbol{k}$, respectively;
 2: repeat
 3:     // Forward propagation
 4:     for $l = 1$ to $L$ do
 5:         $\hat{\boldsymbol{k}}_i^l = \boldsymbol{w}^l \circ \text{sign}(\boldsymbol{k}_i^l), \forall i$; // Each element of $\boldsymbol{w}^l$ is replaced by the average of all
            elements $\overline{w}^l$.
 6:         Perform activation binarization; // Using the sign function
 7:         Perform 2D convolution with $\hat{\boldsymbol{k}}_i^l, \forall i$;
 8:     end for
 9:     // Backward propagation
10:     Compute $\delta_{\hat{\boldsymbol{k}}_i^l} = \frac{\partial L_s}{\partial \hat{\boldsymbol{k}}_i^l}, \forall l, i$;
11:     for $l = L$ to 1 do
12:         Calculate $\delta_{\boldsymbol{k}_i^l}, \delta_{\boldsymbol{w}^l}, \delta_{\mu_i^l}, \delta_{\sigma_i^l}$; // using Eqs. 5.134∼5.141
13:         Update parameters $\boldsymbol{k}_i^l$, $\boldsymbol{w}^l$, $\mu_i^l$, $\sigma_i^l$ using SGD;
14:     end for
15:     Update $\boldsymbol{c}_m, \boldsymbol{\sigma}_m$;
16: until convergence
```

5.5.2.6 Asynchronous Backward Propagation

To minimize Eq. 5.130, we update $\boldsymbol{k}_n^{l,i}$, $\boldsymbol{w}^l$, μ_i^l, σ_i^l, $\boldsymbol{c}_m$, and $\boldsymbol{\sigma}_m$ using stochastic gradient descent (SGD) in an asynchronous manner, which updates $\boldsymbol{w}$ instead of $\overline{w}$ as elaborated below.

Updating $\boldsymbol{k}_n^{l,i}$ We define $\delta_{\boldsymbol{k}_n^{l,i}}$ as the gradient of the full-precision kernel $\boldsymbol{k}_n^{l,i}$, and we have:

$$\delta_{\boldsymbol{k}_n^{l,i}} = \frac{\partial L}{\partial \boldsymbol{k}_n^{l,i}} = \frac{\partial L_S}{\partial \boldsymbol{k}_n^{l,i}} + \frac{\partial L_B}{\partial \boldsymbol{k}_n^{l,i}}. \tag{5.134}$$

For each term in Eq. 5.134, we have:

$$\begin{aligned}\frac{\partial L_S}{\partial \boldsymbol{k}_n^{l,i}} &= \frac{\partial L_S}{\partial \hat{\boldsymbol{k}}_n^{l,i}} \frac{\partial \hat{\boldsymbol{k}}_n^{l,i}}{\partial (\boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i})} \frac{\partial (\boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i})}{\partial \boldsymbol{k}_n^{l,i}} \\ &= \frac{\partial L_S}{\partial \hat{\boldsymbol{k}}_n^{l,i}} \circ \mathbb{1}_{-1 \le \boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i} \le 1} \circ \boldsymbol{w}^l,\end{aligned} \tag{5.135}$$

$$\begin{aligned}\frac{\partial L_B}{\partial \boldsymbol{k}_n^{l,i}} &= \lambda\{\boldsymbol{w}^l \circ \left[\boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i} - \hat{\boldsymbol{k}}_n^{l,i}\right] \\ &+ \nu[(\boldsymbol{\sigma}_i^l)^{-2} \circ (\boldsymbol{k}_{i+}^l - \boldsymbol{\mu}_{i+}^l) \\ &+ (\boldsymbol{\sigma}_i^l)^{-2} \circ (\boldsymbol{k}_{i-}^l + \boldsymbol{\mu}_{i-}^l)],\end{aligned} \tag{5.136}$$

where $\mathbb{1}$ is the indicator function that is widely used to estimate the gradient of nondifferentiable parameters [53], and $(\boldsymbol{\sigma}_i^l)^{-2}$ is a vector whose elements are all equal to $(\sigma_i^l)^{-2}$.

Updating $\boldsymbol{w}^l$ Unlike the forward process, $\boldsymbol{w}$ is used in back propagation to calculate the gradients. This process is similar to the way to calculate $\hat{\boldsymbol{x}}$ from $\boldsymbol{x}$ asynchronously. Specifically, $\delta_{\boldsymbol{w}^l}$ is composed of the following two parts:

$$\delta_{\boldsymbol{w}^l} = \frac{\partial L}{\partial \boldsymbol{w}^l} = \frac{\partial L_S}{\partial \boldsymbol{w}^l} + \frac{\partial L_B}{\partial \boldsymbol{w}^l}. \tag{5.137}$$

For each term in Eq. 5.137, we have:

$$\begin{aligned}\frac{\partial L_S}{\partial \boldsymbol{w}^l} &= \sum_{i=1}^{I_l}\sum_{n=1}^{N_{I_l}} \frac{\partial L_S}{\partial \hat{\boldsymbol{k}}_n^{l,i}} \frac{\partial \hat{\boldsymbol{k}}_n^{l,i}}{\partial(\boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i})} \frac{\partial(\boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i})}{\partial \boldsymbol{w}^l} \\ &= \sum_{i=1}^{I_l}\sum_{n=1}^{N_{I_L}} \frac{\partial L_S}{\partial \hat{\boldsymbol{k}}_n^{l,i}} \circ \mathbb{1}_{-1\le \boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i} \le 1} \circ \boldsymbol{k}_n^{l,i},\end{aligned} \tag{5.138}$$

$$\frac{\partial L_B}{\partial \boldsymbol{w}^l} = \lambda \sum_{i=1}^{I_l}\sum_{n=1}^{N_{I_l}} (\boldsymbol{w}^l \circ \boldsymbol{k}_n^{l,i} - \hat{\boldsymbol{k}}_n^{l,i}) \circ \boldsymbol{k}_n^{l,i}. \tag{5.139}$$

Updating μ_i^l and σ_i^l Note that we use the same μ_i^l and σ_i^l for each kernel (see Sect. 3.2). So, the gradients here are scalars. The gradients $\delta_{\mu_i^l}$ and $\delta_{\sigma_i^l}$ are calculated as:

$$\begin{aligned}\delta_{\mu_i^l} &= \frac{\partial L}{\partial \mu_i^l} = \frac{\partial L_B}{\partial \mu_i^l} \\ &= \frac{\lambda\nu}{C_i^l \times H^l \times W^l} \sum_{n=1}^{C_i^l}\sum_{p=1}^{H^l \times W^l} \begin{cases} (\sigma_i^l)^{-2}(\mu_i^l - k_{n,p}^{l,i}), & k_{n,p}^{l,i} \ge 0, \\ (\sigma_i^l)^{-2}(\mu_i^l + k_{n,p}^{l,i}), & k_{n,p}^{l,i} < 0, \end{cases}\end{aligned} \tag{5.140}$$

$$\begin{aligned}\delta_{\sigma_i^l} &= \frac{\partial L}{\partial \sigma_i^l} = \frac{\partial L_B}{\partial \sigma_i^l} \\ &= \frac{\lambda \nu}{C_i^l \times H^l \times W^l} \sum_{n=1}^{C_i^l} \sum_{p=1}^{H^l \times W^l} \begin{cases} -(\sigma_i^l)^{-3}(k_{n,p}^{l,i} - \mu_i^l)^2 + (\sigma_i^l)^{-1}, k_{n,p}^{l,i} \geq 0, \\ -(\sigma_i^l)^{-3}(k_{n,p}^{l,i} + \mu_i^l)^2 + (\sigma_i^l)^{-1}, k_{n,p}^{l,i} < 0, \end{cases}\end{aligned} \tag{5.141}$$

where $k_{n,p}^{l,i}$, $p \in \{1, \ldots, H^l \times W^l\}$, denotes the p-th element of $\boldsymbol{k}_n^{l,i}$. In the fine-tuning process, we update $\boldsymbol{c}_m$ using the same strategy as center loss [64]. The update of $\sigma_{m,n}$ based on L_B is straightforward and is not elaborated here for brevity.

Updating $\boldsymbol{K}_{i,j}^l$ In pruning, we aim to converge the filters to their mean gradually. So we replace each filter $\boldsymbol{K}_{i,j}^l$ with its corresponding mean $\overline{\boldsymbol{K}}_{i,j}^l$. The gradient of the mean is represented as follows:

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{K}_{i,j}^l} &= \frac{\partial L_S}{\partial \boldsymbol{K}_{i,j}^l} + \frac{\partial L_B}{\partial \boldsymbol{K}_{i,j}^l} + \frac{\partial L_P}{\partial \boldsymbol{K}_{i,j}^l} \\ &= \frac{\partial L_S}{\partial \overline{\boldsymbol{K}}_j^l} \frac{\partial \overline{\boldsymbol{K}}_j^l}{\partial \boldsymbol{K}_{i,j}^l} + \frac{\partial L_B}{\partial \overline{\boldsymbol{K}}_j^l} \frac{\partial \overline{\boldsymbol{K}}_j^l}{\partial \boldsymbol{K}_{i,j}^l} + \frac{\partial L_P}{\partial \boldsymbol{K}_{i,j}^l} \\ &= \frac{1}{I_j}\left(\frac{\partial L_S}{\partial \overline{\boldsymbol{K}}_j^l} + \frac{\partial L_B}{\partial \overline{\boldsymbol{K}}_j^l}\right) + 2(\boldsymbol{K}_{i,j}^l - \overline{\boldsymbol{K}}_j) \\ &\quad + 2\nu(\boldsymbol{\Psi}_j^l)^{-1}(\boldsymbol{K}_{i,j}^l - \overline{\boldsymbol{K}}_j),\end{aligned} \tag{5.142}$$

where $\overline{\boldsymbol{K}}_j^l = \frac{1}{I_j}\sum_{i=1}^{I_j} \boldsymbol{K}_{i,j}^l$ that is used to update the filters in a group by mean $\overline{\boldsymbol{K}}_j^l$. We leave the first filter in each group to prune redundant filters and remove the others. However, such an operation changes the distribution of the input channel of the batch norm layer, resulting in a dimension mismatch for the next convolutional layer. To solve the problem, we keep the size of the batch norm layer, whose values correspond to the removed filters, set to zero. In this way, the removed information is retained to the greatest extent. In summary, we show that the proposed method is trainable from end to end. The learning procedure is detailed in Algorithms 11 and 12 (Figs. 5.20 and 5.21).

5.5.2.7 Ablation Study

Hyperparameter Selection In this section, we conduct evaluations to study the effects of hyperparameters on the performance of BONNs, specifically focusing on λ and θ. These hyperparameters are used to balance the Bayesian kernel loss and the Bayesian feature loss, respectively, and are crucial in adjusting the distributions of

Algorithm 12: Pruning 1-bit CNNs with Bayesian learning

```
Input:
    The pre-trained 1-bit CNN model with parameters $\boldsymbol{K}$, the reconstruction vector $\boldsymbol{w}$, the
    learning rate $\eta$, regularization parameters $\lambda, \theta$, variance $\nu$ and convergence rate $\gamma$ and
    the training dataset.
Output:
    The pruned BONN with updated $\boldsymbol{K}, \boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{c}_m, \boldsymbol{\sigma}_m$.
 1: repeat
 2:     // Forward propagation
 3:     for $l = 1$ to $L$ do
 4:         $\boldsymbol{K}^l_{i,j} = (1-\gamma)\boldsymbol{K}^l_{i,j} + \gamma\overline{\boldsymbol{K}}^l_j$;
 5:         $\hat{\boldsymbol{k}}^l_i = \boldsymbol{w}^l \circ \text{sign}(\boldsymbol{k}^l_i), \forall i$; // Each element of $\boldsymbol{w}^l$ is replaced by the average of all
            elements $\overline{w}^l$.
 6:         Perform activation binarization; // Using the sign function
 7:         Perform 2D convolution with $\hat{\boldsymbol{k}}^l_i, \forall i$;
 8:     end for
 9:     // Backward propagation
10:     Compute $\delta_{\hat{\boldsymbol{k}}^l_i} = \frac{\partial L_s}{\partial \hat{\boldsymbol{k}}^l_i}, \forall l, i$;
11:     for $l = L$ to $1$ do
12:         Calculate $\delta_{\boldsymbol{k}^l_i}, \delta_{\boldsymbol{w}^l}, \delta_{\mu^l_i}, \delta_{\sigma^l_i}$; // using Eqs. 5.137~5.142
13:         Update parameters $\boldsymbol{k}^l_i, \boldsymbol{w}^l, \mu^l_i, \sigma^l_i$ using SGD;
14:     end for
15:     Update $\boldsymbol{c}_m, \boldsymbol{\sigma}_m$;
16: until Filters in the same group are similar enough
```

kernels and features for better performance.We use wide residual networks (WRN-22 and WRN-40) for our evaluations. The implementation details are provided below.

In Table 5.20, we vary λ while setting θ to zero to understand the influence of the Bayesian kernel loss on the kernel distribution. The results show that incorporating the Bayesian kernel loss effectively improves the accuracy on CIFAR-10. However, simply increasing λ does not lead to higher accuracy. Instead, finding an appropriate value of λ is essential to strike the right balance between the cross-entropy loss and the Bayesian kernel loss. For instance, when λ is set to $1e-4$, we achieve the best classification accuracy, indicating an optimal balance.

Next, we study the effect of the hyperparameter θ on the intraclass variations of features using different values of θ. Similar to the observations with λ, the classification accuracy varies with θ, demonstrating that the Bayesian feature loss can contribute to better classification accuracy when an appropriate value of θ is chosen.

Furthermore, we evaluate the convergence performance of our method in comparison to other methods using ResNet-18 on ImageNet ILSVRC12. The training curve of XNOR-Net shows vigorous oscillations, which suggests suboptimal learning. In contrast, our BONN achieves better training and test accuracy, indicating improved convergence performance.

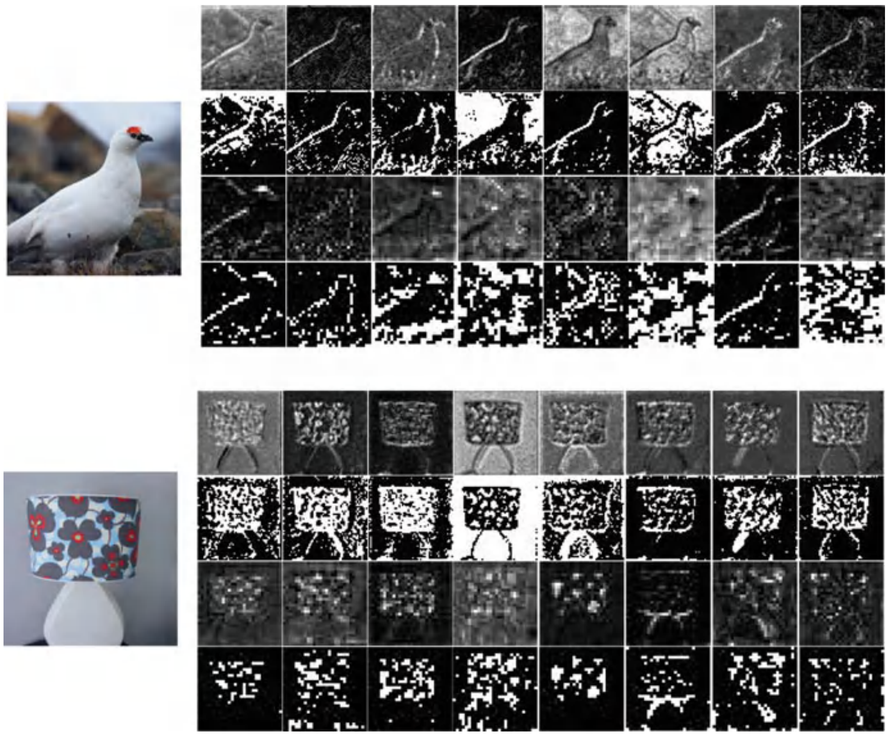
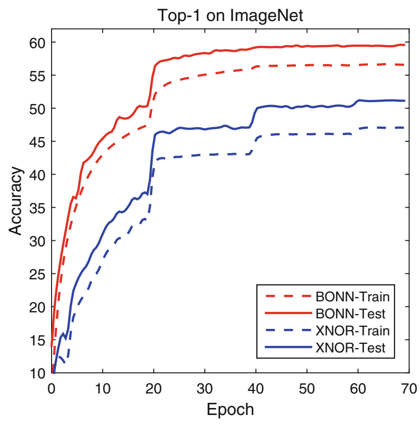


Fig. 5.20 The images on the left are the input images chosen from the ImageNet ILSVRC12 dataset. Right images are feature maps and binary feature maps from different layers of BONNs. The first and third rows are feature maps for each group, while the second and fourth rows are corresponding binary feature maps. Although binarization of the feature map causes information loss, BONNs could extract essential features for accurate classification

Effectiveness of Bayesian Binarization on ImageNet ILSVRC12 We experimented with examining how each loss affects performance better to understand Bayesian losses on the large-scale ImageNet ILSVRC12 dataset. Based on the experiments described above, we set λ to $1e-4$ and θ to $1e-3$ if they are used. As shown in Table 5.21, both the Bayesian kernel loss and Bayesian feature loss can independently improve the accuracy on ImageNet. When applied together, the Top-1 accuracy reaches the highest value of 59.3%.

Weight Distribution Figure 5.22 further illustrates the distribution of the kernel weights, with λ fixed to $1e-4$. During the training process, the distribution gradually approaches the two-mode GMM, as assumed previously, confirming the effectiveness of the Bayesian kernel loss in a more intuitive way. We also compare the kernel weight distribution between XNOR-Net and BONN. As shown in Fig. 5.23, the kernel weights learned in XNOR-Net are tightly distributed around



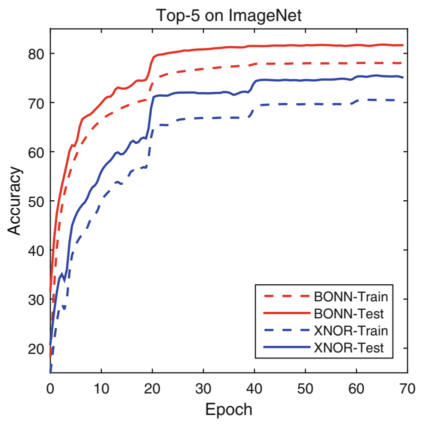


Fig. 5.21 Training and test accuracies on ImageNet when $\lambda = 1e-4$ shows the superiority of the proposed BONN over XNOR-Net. The backbone of the two networks is ResNet-18

Table 5.20 With different λ and θ, we evaluate the accuracies of BONNs based on WRN-22 and WRN-40 on CIFAR-10/CIFAR-100. When varying λ, the Bayesian feature loss is not used ($\theta = 0$). However, when varying θ, we choose the optimal loss weight ($\lambda = 1e-4$) for the Bayesian kernel loss

		WRN-22 (BONN)		WRN-40 (BONN)	
Hyper-param.		CIFAR-10	CIFAR-100	CIFAR-10	CIFAR-100
λ	$1e-3$	85.82	59.32	85.79	58.84
	$1e-4$	**86.23**	**59.77**	**87.12**	**60.32**
	$1e-5$	85.74	57.73	86.22	59.93
	0	84.97	55.38	84.61	56.03
θ	$1e-2$	**87.34**	60.31	87.23	60.83
	$1e-3$	86.49	60.37	87.18	**61.25**
	$1e-4$	86.27	**60.91**	**87.41**	61.03
	0	86.23	59.77	87.12	60.32

Table 5.21 Effect of Bayesian losses on the ImageNet dataset. The backbone is ResNet-18

Bayesian kernel loss		✗	✓	✗	✓
Bayesian feature loss		✗	✗	✓	✓
Accuracy	Top-1	56.3	58.3	58.4	**59.3**
	Top-5	79.8	80.8	80.8	**81.6**

the threshold value, but those in BONN are regularized in a two-mode GMM style. Figure 5.24 shows the evolution of the binarized values during the training process of XNOR-Net and BONN. The two different patterns indicate that the binarized values learned in BONN are more diverse.

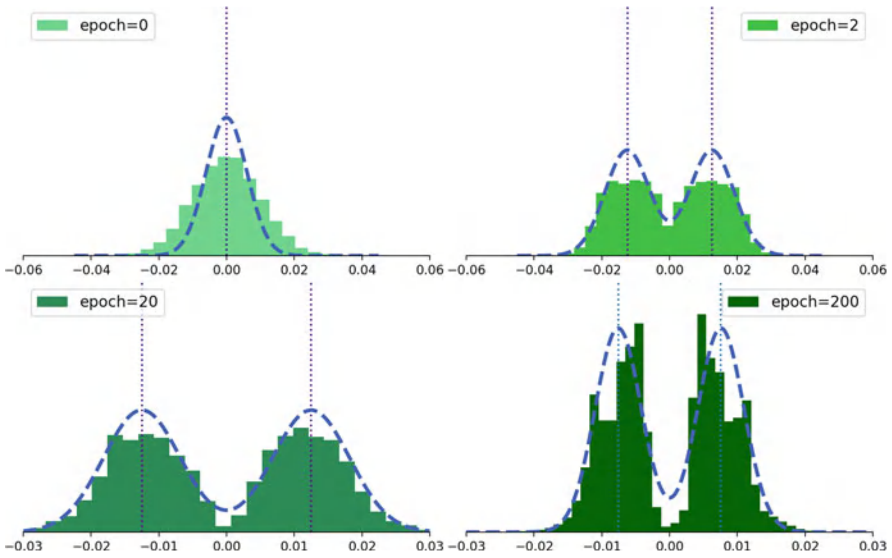


Fig. 5.22 We demonstrate the kernel weight distribution of the first binarized convolutional layer of BONNs. Before training, we initialize the kernels as a single-mode Gaussian distribution. From the 2-th epoch to the 200-th epoch, with λ fixed to $1e-4$, the distribution of the kernel weights becomes more and more compact with two modes, which confirms that the Bayesian kernel loss can regularize the kernels into a promising distribution for binarization

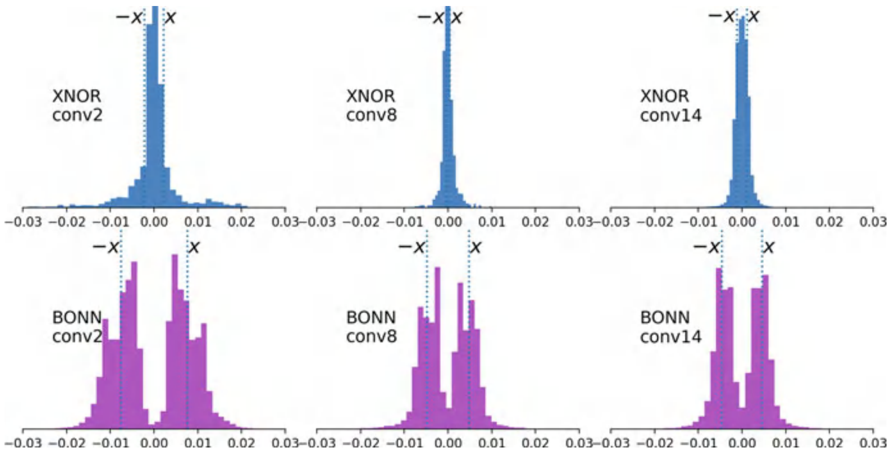


Fig. 5.23 The weight distributions of XNOR and BONN are based on WRN-22 (2nd, 8th, and 14th convolutional layers) after 200 epochs. The weight distribution difference between XNOR and BONN indicates that the kernels are regularized across the convolutional layers with the proposed Bayesian kernel loss

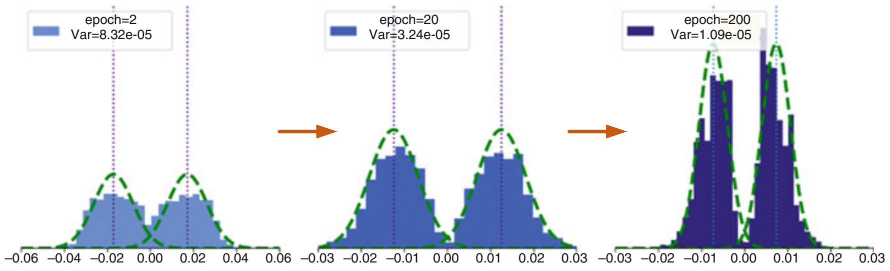


Fig. 5.24 Evolution of the binarized values, $|x|$s, during the XNOR and BONN training process. They are both based on WRN-22 (2nd, 3rd, 8th, and 14th convolutional layers), and the curves do not share the same y-axis. The binarized values of XNOR-Net tend to converge to small and similar values, but these of BONN are learned diversely

Table 5.22 Effect of Bayesian feature loss on the ImageNet dataset. The core is ResNet-18 and ResNet-50 with real value

Model		ResNet-18		ResNet-50	
Bayesian feature loss		✗	✓	✗	✓
Accuracy	Top-1	69.3	**69.9**	76.6	**77.0**
	Top-5	89.2	**89.8**	92.4	**92.7**

Effectiveness of Bayesian Feature Loss on Real-Valued Models

We have applied our Bayesian feature loss on real-valued models, specifically ResNet-18 and ResNet-50 [19]. During retraining, we incorporated our Bayesian feature loss for 70 epochs, setting the hyperparameter θ to $1e-3$. The SGD optimizer was used with an initial learning rate of 0.1, and a learning rate schedule that decreases to 10% every 30 epochs.

The results, as shown in Table 5.22, demonstrate that our Bayesian feature loss significantly improves the performance of models with real values. Specifically, the Top-1 accuracies of ResNet-18 and ResNet-50 are boosted by 0.6% and 0.4%, respectively.

References

1. Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *Advances in neural information processing systems*, pages 2270–2278, 2016.
2. Jose M Alvarez and Mathieu Salzmann. Compression-aware training of deep networks. In *Advances in Neural Information Processing Systems*, pages 856–867, 2017.
3. Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.
4. Christopher M Bishop. Bayesian neural networks. *Journal of the Brazilian Computer Society*, 4(1):61–68, 1997.
5. Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural network. In *Proceedings of the International Conference on Machine Learning*, pages 1613–1622, 2015.

6. Neill DF Campbell, George Vogiatzis, Carlos Hernández, and Roberto Cipolla. Using multiple hypotheses to improve depth-maps for multi-view stereo. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 766–779. Springer, 2008.
7. Robert T Collins. A space-sweep approach to true multi-image matching. In *Proceedings CVPR IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 358–363. IEEE, 1996.
8. Xiaohan Ding, Guiguang Ding, Yuchen Guo, and Jungong Han. Centripetal sgd for pruning very deep convolutional networks with complicated structure. In *CVPR*, pages 4943–4953, 2019.
9. Xuanyi Dong, Junshi Huang, Yi Yang, and Shuicheng Yan. More is less: A more complicated network with less inference complexity. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5840–5848, 2017.
10. Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multiview stereopsis. *IEEE transactions on pattern analysis and machine intelligence*, 32(8):1362–1376, 2009.
11. Silvano Galliani, Katrin Lasinger, and Konrad Schindler. Massively parallel multiview stereopsis by surface normal diffusion. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 873–881, 2015.
12. Andre Gaschler, Darius Burschka, and Gregory Hager. Epipolar-based stereo tracking without explicit 3d reconstruction. In *2010 20th International Conference on Pattern Recognition*, pages 1755–1758. IEEE, 2010.
13. Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 315–323, 2011.
14. Tom Goldstein, Christoph Studer, and Richard Baraniuk. A field guide to forward-backward splitting with a fasta implementation. *arXiv preprint arXiv:1411.3406*, 2014.
15. Jiaxin Gu, Ce Li, Baochang Zhang, Jungong Han, Xianbin Cao, Jianzhuang Liu, and David Doermann. Projection convolutional neural networks for 1-bit cnns via discrete back propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.
16. Jiaxin Gu, Junhe Zhao, Xiaolong Jiang, Baochang Zhang, Jianzhuang Liu, Guodong Guo, and Rongrong Ji. Bayesian optimized 1-bit cnns. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4909–4917, 2019.
17. Shuhang Gu, Wangmeng Zuo, Qi Xie, Deyu Meng, Xiangchu Feng, and Lei Zhang. Convolutional sparse coding for image super-resolution. In *ICCV*, pages 1823–1831, 2015.
18. Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.
19. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
20. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.
21. Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2234–2240, 2018.
22. Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *CVPR*, pages 4340–4349, 2019.
23. Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, pages 1398–1406, 2017.
24. Felix Heide, Wolfgang Heidrich, and Gordon Wetzstein. Fast and flexible convolutional sparse coding. In *CVPR*, pages 5135–5143, 2015.
25. Andrew Howard. Real-time stereo visual odometry for autonomous ground vehicles. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3946–3952. IEEE, 2008.

26. Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *IEEE International Conference on Computer Vision (ICCV)*, pages 1314–1324, 2019.
27. Po-Han Huang, Kevin Matzen, Johannes Kopf, Narendra Ahuja, and Jia-Bin Huang. Deepmvs: Learning multi-view stereopsis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2821–2830, 2018.
28. Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks. In *ECCV*, pages 304–320, 2018.
29. Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks. In *ECCV*, pages 304–320, 2018.
30. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
31. Rasmus Jensen, Anders Dahl, George Vogiatzis, Engin Tola, and Henrik Aanæs. Large scale multi-view stereopsis evaluation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 406–413, 2014.
32. Mengqi Ji, Juergen Gall, Haitian Zheng, Yebin Liu, and Lu Fang. Surfacenet: An end-to-end 3d neural network for multiview stereopsis. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2307–2315, 2017.
33. Alex Kendall, Hayk Martirosyan, Saumitro Dasgupta, Peter Henry, Ryan Kennedy, Abraham Bachrach, and Adam Bry. End-to-end learning of geometry and context for deep stereo regression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 66–75, 2017.
34. Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
35. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.
36. Jouko Lampinen and Aki Vehtari. Bayesian approach for neural networks—review and case studies. *Neural networks*, 14(3):257–274, 2001.
37. Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. In *Artificial intelligence and statistics*, pages 562–570, 2015.
38. Thomas Lemaire, Cyrille Berger, Il-Kyun Jung, and Simon Lacroix. Vision-based slam: Stereo and monocular approaches. *International Journal of Computer Vision*, 74(3):343–364, 2007.
39. Cong Leng, Zesheng Dou, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3466–3473, 2018.
40. Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.
41. Faming Liang, Qizhai Li, and Lei Zhou. Bayesian neural networks for selection of drug sensitive genes. *Journal of the American Statistical Association*, 113(523):955–972, 2018.
42. Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and Baochang Zhang. Accelerating convolutional networks via global & dynamic filter pruning. In *IJCAI*, pages 2425–2432, 2018.
43. Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning. In *CVPR*, 2019.
44. Chunlei Liu, Wenrui Ding, Yuan Hu, Baochang Zhang, Jianzhuang Liu, Guodong Guo, and David Doermann. Rectified binary convolutional networks with generative adversarial learning. *International Journal of Computer Vision*, 129:998–1012, 2021.
45. Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018.
46. Jianhao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *ICCV*, pages 5068–5076, 2017.

47. Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research*, 11(Jan):19–60, 2010.
48. M. Menze and A. Geiger. Object scene flow for autonomous vehicles. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3061–3070, 2015.
49. Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.
50. Neal Parikh, Stephen Boyd, et al. Proximal algorithms. *Foundations and Trends® in Optimization*, 1(3):127–239, 2014.
51. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.
52. Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7(15):510, 2008.
53. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.
54. Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, pages 4510–4520, 2018.
55. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
56. Pravendra Singh, Vinay Kumar Verma, Piyush Rai, and Vinay Namboodiri. Leveraging filter correlations for deep model compression. In *The IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 835–844, 2020.
57. Shengyang Sun, Changyou Chen, and Lawrence Carin. Learning structured weight uncertainty in bayesian neural networks. In *Proceedings of the Artificial Intelligence and Statistics*, pages 1283–1292, 2017.
58. Shengyang Sun, Guodong Zhang, Jiaxin Shi, and Roger Grosse. Functional variational bayesian neural networks. In *Proceedings of the International Conference on Learning Representations*, pages 1–22, 2019.
59. Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.
60. Engin Tola, Christoph Strecha, and Pascal Fua. Efficient large-scale multi-view stereo for ultra high-resolution image sets. *Machine Vision and Applications*, 23(5):903–920, 2012.
61. Hanzi Wang, Daniel Mirota, Masaru Ishii, and Gregory D Hager. Robust motion estimation and structure recovery from endoscopic image sequences with an adaptive scale kernel consensus estimator. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–7. IEEE, 2008.
62. Runqi Wang, Baochang Zhang, Li'an Zhuo, Qixiang Ye, and David Doermann. Cogradient descent for dependable learning. *arXiv preprint arXiv:2106.10617*, 2021.
63. Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016.
64. Yandong Wen, Kaipeng Zhang, Zhifeng Li, and Yu Qiao. A discriminative feature learning approach for deep face recognition. In *European Conference on Computer Vision (ECCV)*, pages 499–515, 2016.
65. Xiang Xiang. A brief review on visual tracking methods. In *Third Chinese Conference on Intelligent Visual Surveillance*, 2011.
66. Xiang Xiang, Daniel Mirota, Austin Reiter, and Gregory D Hager. Is multi-model feature matching better for endoscopic motion estimation? In *International Workshop on Computer-Assisted and Robotic Endoscopy*, pages 88–98. Springer, 2014.
67. Xiang Xiang, Zhiyuan Wang, Shan Lao, and Baochang Zhang. Pruning multi-view stereo net for efficient 3d reconstruction. *Isprs Journal of Photogrammetry and Remote Sensing*, 168:17–27, 2020.

68. Sheng Xu, Hanlin Chen, Xuan Gong, Kexin Liu, Jinhu Lu, and Baochang Zhang. Efficient structured pruning based on deep feature stabilization. *Neural Computing and Applications*, 33:7409–7420, 2021.
69. Yao Yao, Zixin Luo, Shiwei Li, Tian Fang, and Long Quan. Mvsnet: Depth inference for unstructured multi-view stereo. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 767–783, 2018.
70. Jianbo Ye, Xin Lu, Zhe Lin, and James Z Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *ICLR*, 2018.
71. Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2018.
72. Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *CVPR*, pages 2528–2535. IEEE, 2010.
73. Ziming Zhang and Venkatesh Saligrama. Rapid: Rapidly accelerated proximal gradient algorithms for convex minimization. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3796–3800, 2015.
74. et al. Zhang K, Schölkopf B. Learning causality and causality-related learning: some recent progress. *National science review*, 2018.
75. Yuefu Zhou, Ya Zhang, Yanfeng Wang, and Qi Tian. Accelerate cnn via recursive bayesian pruning. In *ICCV*, pages 3306–3315, 2019.
76. Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation. In *Deep Learning in Medical Image Analysis and Multimodalf Learning for Clinical Decision Support*, pages 3–11. Springer, 2018.
77. Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, pages 1–16, 2017.

Chapter 6
Applications

6.1 Introduction

The success of binary neural networks makes it possible to apply deep learning models to edge computing. Neural network models have been used in various real tasks with the help of these binary methods, including image classification, image classification, speech recognition, and object detection and tracking. In this section, we introduce the applications of binary neural networks in these fields.

In this chapter, we introduce the applications of binary neural networks in computer vision. Specifically, we introduce the vision tasks, including object detection, speech recognition, person reidentification, and 3D point cloud processing.

6.2 Image Classification

Image classification aims to group images into different semantic classes together. Many works regard the completion of image classification as the criterion for the success of binary neural networks. Five datasets are commonly used for image classification tasks: MNIST [42], SVHN, CIFAR-10 [26], CIFAR-100, and ImageNet [52]. Among them, ImageNet is the most difficult to train and consists of 100 classes of images. Table 2.1 shows the experimental results of some of the most popular binary methods on ImageNet.

6.3 Speech Recognition

Speech recognition is a technique or capability that enables a program or system to process human speech. We can use binary methods to complete speech recognition tasks in edge computing devices.

Xiang et al. [67] applied binary DNNs to speech recognition tasks. Experiments on TIMIT phone recognition and 50-hour Switchboard speech recognition show that binary DNNs can run about four times faster than standard DNNs during inference, with roughly 10.0% relative accuracy reduction.

Zheng et al. [74] and Yin et al. [71] also implement binarized convolutional neural network-based speech recognition tasks.

6.3.1 1-Bit WaveNet: Compression of a Generative Neural Network in Speech Recognition with Two Binarized Methods

Instead of traditional speech recognition applications on remote servers, speech recognition is gradually becoming popular on mobile devices. However, significant memory and computational resource requirements restrict full-precision neural networks. Before solving the hardware deployment problem on mobile devices, we needed more parameters to run or store these DCNNs. To resolve these challenges, Rastegari et al. [47] use binary operations to approximate convolutions using binarized kernel weights and input. In recent years, Zhou et al. presented DoReFa-Net [76] that could speed training and inference with 1-bit convolution filters with low bit width parameter gradients. Lin et al. [33] binarized multiple activations and weights to approximately replace the weights of real value. Consequently, the degradation of prediction accuracy is decreased. Zhuang et al. proposed a two-stage optimization method to quantize weights and activations and then train a 4-bit model. This results in no performance reduction compared to its real value counterpart at the baseline. McDonnell [39] achieves equivalent binarized results compared to the basic baseline by applying scaling factors to balance each layer with constant unlearned values and standard deviations specific to the initial layer. In contrast to baseline models, Wang et al. [61] proposed MCNs that only substitute full-precision kernels with binarized parameters and obtain excellent performance.

Although these BNNs save considerable memory and computational power, the accuracy of vision or speech tasks is reduced. The main explanations are as follows: (1) In previous work, they rarely resolved the process of CNN binarization by discrete optimization [10]. (2) Considering binarized and filter losses, existing approaches could not trade them off well.

In this work, considering the outstanding achievements of BNNs in computer vision and existing binary RNN research in speech recognition [46], we propose a new binarization application through two technologies (Bi-Real Net [37] and

PCNN [15]) on WaveNet to accomplish our speech keyword recognition mission and acquire the closest-to-baseline accuracy with subtle numerical error. We demonstrate the principle of our new 1-bit WaveNet via extraordinary dilated causal convolutions and residual blocks, which compress the baseline up to a third of its original size with similar accuracy. Three technical novelties of our work include the following: (1) the entire framework of our new 1-bit WaveNet [11] based on binary dilated causal convolutions, which enlarge receptive fields, is presented in our speech keyword recognition tasks to save memory and computational resources; (2) a new application in speech recognition is proposed by binarization of 1D convolution; and (3) an audio keywords dataset that could be tested by our model and prepared to facilitate future research is collected and labeled.

6.3.1.1 Network Architecture

In this work, we propose a 1-bit neural network model [23] based on WaveNet that has achieved exceptional performance on the raw audio waveform. WaveNet is a deep autoregressive neural network with a point-by-point sampling method, and it could achieve high-quality audio via a conditional probability formula as follows:

$$P(X) = \prod_{n=1}^{N} P\left(x_n \mid x_1, \ldots, x_{n-1}\right) \tag{6.1}$$

where past speech samples from previous steps generate each x_n.

Figure 6.1 shows the 1-bit WaveNet architecture, which contains a preprocessing data module that converts raw clear keyword spectrogram data into Mel frequency cepstrum coefficient (MFCC) and then inputs these data into the main network, the principal part of this WaveNet, which is composed of several residual blocks, and a DenseNet that ensures that the outputs could be distributed as a categorical form to facilitate the calculation, and, meanwhile, it could solve the overflow problem in the model.

Dilated Causal Convolutions This WaveNet model is based on PixelCNN [56], which discarded the pooling layers in the architecture but used a unique 1D convolution many times, called the causal convolution. The modeling process could certainly be in the correct time sequence, that is, the output $P\left(x_{n+1} \mid x_1, \ldots, x_n\right)$ produced by the model can only be generated from the present steps, but not from the predictions of the future. After predicting each audio sample, the model receives and applies it to the next prediction. Causal convolution-trained complicated sequences save more time than traditional RNNs such as LSTMs or GRUs [44] due to cutoff recurrent connections. Furthermore, convolutions with holes (dilated convolutions) include a new hyperparameter named the dilation rate to increase kernels' reception field efficiently. Similarly to subsampled layers, the output and the input of dilated convolutions have an equivalent size. Our model could use only several layers to enlarge the receptive fields with considerable input resolution

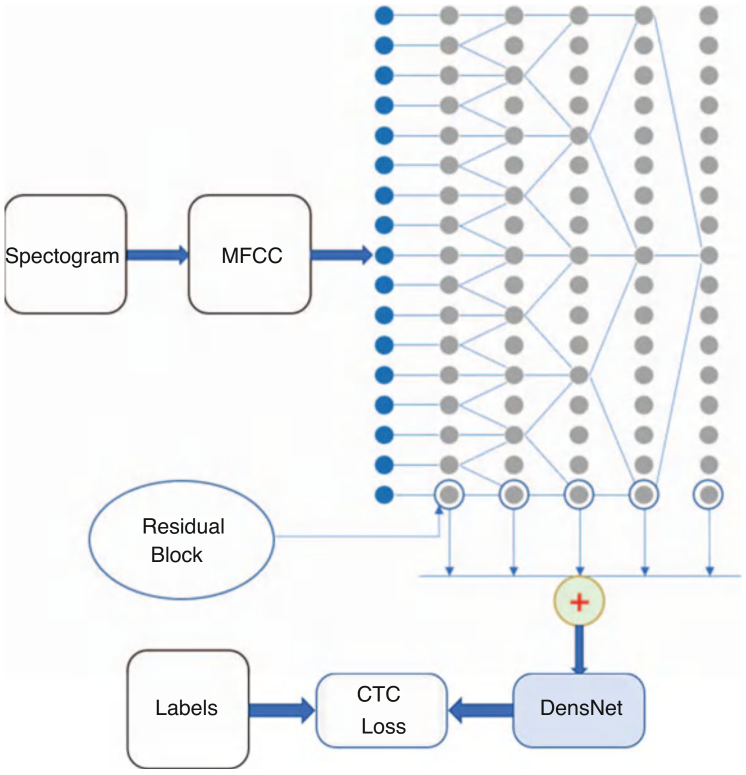


Fig. 6.1 In the 1-bit WaveNet, a new binarized application via Bi-Real Net and PCNN is used to compress the speech recognition model. This flowchart illustrates the network architecture with all techniques in this work. See the text for a detailed description of the model

and reasonable computational resources. In our work, we apply five layers with the dilation from 2^0 to 2^4.

Gated Activation Units and Residual Blocks We used the same gated activation unit as the original WaveNet [44]:

$$\mathbf{z} = \tanh\left(W_{f,k} * \mathbf{x}\right) \odot \sigma\left(W_{g,k} * \mathbf{x}\right) \tag{6.2}$$

where x is the output of dilated causal convolutions, W is a trainable convolution kernel, $\odot$ denotes the Hadamard product, $*$ denotes the convolutional operation, and $\sigma(\cdot)$ is a sigmoid function whose nonlinearity works better than other activation functions in speech recognition tasks [41].

This model uses three residual blocks to accelerate the convergence process when training deep convolutional neural networks. Figure 6.2 illustrates more details about one of our residual blocks.

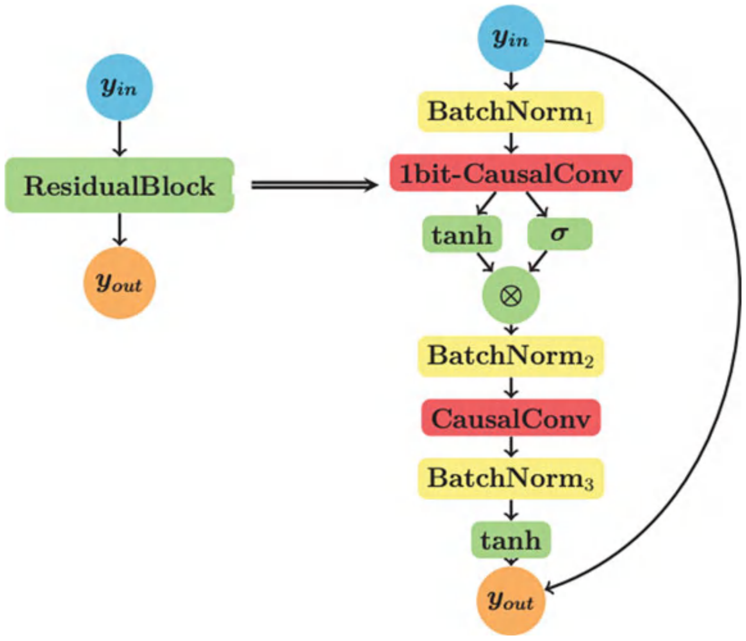


Fig. 6.2 The proposed residual block modified from WaveNet [44], which adds some of the BatchNorm layers and our new 1-bit causal convolution.$\otimes$ denotes the element-wise multiplication operator

6.3.1.2 Bi-Real Net Binarization

In our neural network, we binarized weights through a sign function in 1-bit dilated causal convolutions:

$$w_b = \text{Sign}(w_r) = \begin{cases} -1 & \text{if } w_r < 0 \\ +1 & \text{otherwise} \end{cases} \tag{6.3}$$

where w_r denotes the real weight. In the backward propagation of the training process, we will use the real weight to update the binary weights called the magnitude-aware gradient regarding the weights [37] to update the binary weights, i.e., $\mathbf{W}_b^l \in \{-1, +1\}$. Because convolutions are one-dimensional in our 1-bit WaveNet, $\mathbf{W}_b^l$ in the following equations is a one-dimensional vector. Traditional gradient descent is too small to update binary weights, so Courbariaux proposed a training approach that used a full-precision weight and a sign function [5]. Therefore, $\mathbf{W}_r^l$ will be changed in the back propagation as follows:

$$\mathbf{W}_r^{l,t+1} = \mathbf{W}_r^{l,t} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_r^{l,t}} = \mathbf{W}_r^{l,t} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_b^{l,t}} \frac{\partial \mathbf{W}_b^{l,t}}{\partial \mathbf{W}_r^{l,t}} \tag{6.4}$$

$$\frac{\partial \mathbf{W}_b^{l,t}(i,j)}{\partial \mathbf{W}_r^{l,t}(i,j)} = \begin{cases} 1 & \text{if } \mathbf{W}_r^{l,t}(i,j) \in [-1,1] \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

where $\frac{\partial \mathbf{W}_b^{l,t}(i,j)}{\partial \mathbf{W}_r^{l,t}(i,j)}$ denotes the element-wise derivative and $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_b^{l,t}}$ is derived from the chain rule. Given that, we proposed a new magnitude-aware function to substitute the sign function as follows:

$$\overline{\mathbf{W}}_b^{l,t} = \frac{\left\| \mathbf{W}_r^{l,t} \right\|_{1,1}}{\left| \mathbf{W}_r^{l,t} \right|} \operatorname{Sign}\left(\mathbf{W}_r^{l,t} \right) \tag{6.6}$$

Hence, W_r^l updates to:

$$\mathbf{W}_r^{l,t+1} = \mathbf{W}_r^{l,t} - \eta \frac{\partial \mathcal{L}}{\partial \overline{\mathbf{W}}_b^{l,t}} \frac{\partial \overline{\mathbf{W}}_b^{l,t}}{\partial \mathbf{W}_r^{l,t}} \tag{6.7}$$

where

$$\frac{\partial \overline{\mathbf{W}}_b^{l,t}}{\partial \mathbf{W}_r^{l,t}} \approx \frac{\left\| \mathbf{W}_r^{l,t} \right\|_{1,1}}{\left| \mathbf{W}_r^{l,t} \right|} \tag{6.8}$$

$$\frac{\partial \operatorname{Sign}\left(\mathbf{W}_r^{l,t} \right)}{\partial \mathbf{W}_r^{l,t}} \approx \mathbf{1}_{\left| \mathbf{W}_r^{l,t} \right| < 1} \tag{6.9}$$

and $\bar{\theta}^{l,t}$ is related to the magnitude of $W_b^{l,t}$. Eventually, as we show in Fig. 6.3, this magnitude-aware function changes the weight's sign obviously so that the stochastic gradient descent (SGD) could not achieve this significant effect [37].

6.3.1.3 Projection Convolutional Neural Network Binarization

Projection Loss Figure 6.4 shows the projection convolutional neural network (PCNN) strategy that uses a discrete back propagation via projection [15] to compress our model. The projection loss obtained by optimization is as follows:

$$L_p = \frac{\lambda}{2} \sum_{l,i}^{L,I} \sum_{j}^{J} \left\| \hat{C}_{i,j}^{l,[k]} - \widetilde{W}_j^{l,[k]} \otimes \left(C_i^{l,[k]} + \eta \delta_{\hat{C}_{i,j}^l,[k]} \right) \right\|^2 \tag{6.10}$$

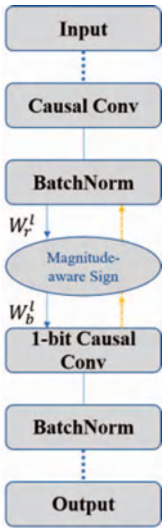


Fig. 6.3 Illustration of the training architecture of 1-bit WaveNet by Bi-Real Net. Note that W is weighted; the superscript l means the lth block, which includes magnitude-aware sign, 1-bit causal convolution, and BatchNorm; and the subscript r and b denote real values and binary values, respectively

where $\otimes$ denotes the element-wise multiplication operator, J is the total number of projections, l denotes the layer index, $[k]$ is the iteration index, and i, j is the kernel index. The projection index and λ, respectively, denote a trade-off scaler for the projection loss. In speech recognition, W is generally a 1D projection vector. Specifically, in the kth iteration, $C_i^{l,[k]}$ means the ith filter vector of the lth convolutional layer and $\hat{C}_{i,j}^{l,[k]} = P_\Omega^{l,j}\left(\widetilde{W}_j^{l,[k]} \otimes C_i^{l,[k]}\right)$ denotes the binarized kernel of $C_i^{l,[k]}$ that includes a duplicated dimensional-corresponding projection vector $\widetilde{W}_j^{l,[k]}$. $\eta\delta_{\hat{C}_{i,j}^{l},[k]}$ means the gradient of $\hat{C}_{i,j}^{l,[k]}$ from the loss of CTC at the beginning [14]. Figure 6.4 shows the principle of projection loss. We omit $[k]$ in the following content for convenience. In our 1-bit WaveNets, we should calculate both the CTC loss and projection loss as the total loss as follows:

$$L = L_C + L_P \tag{6.11}$$

Forward Propagation In PCNNs, we concatenate each $\hat{C}_{i,j}^l$ that is binarized from the relevant real value filter $C_{i,j}^l$to define the kernel D_i^l.

$$D_i^l = \hat{C}_{i,1}^l \oplus \hat{C}_{i,2}^l \oplus \cdots \oplus \hat{C}_{i,J}^l \tag{6.12}$$

Note that $\oplus$ is the convolutional concatenating operation. From D^l and F^l, we achieve the projection convolution and then calculate the feature map F^{l+1} of the next layer:

$$F^{l+1} = \mathrm{Conv}\,1D\left(F^l, D^l\right) \tag{6.13}$$

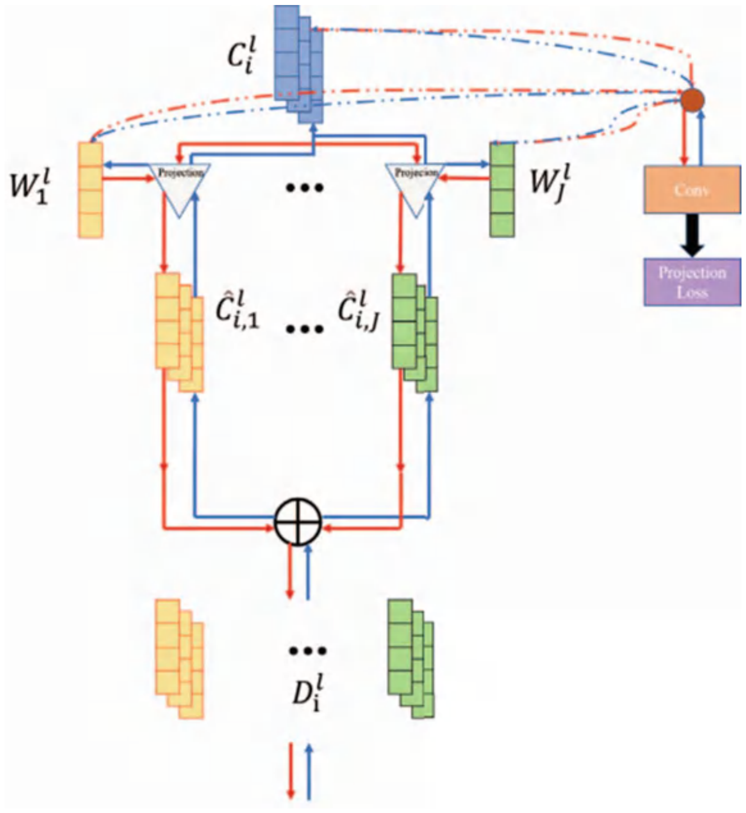


Fig. 6.4 In one-dimensional PCNNs, we propose a discrete back propagation through projection to binarize our WaveNet end-to-end [15]. By using the projection, we binarize the convolutional filters of real value C_i^l to the binary counterpart $hatC_{i,j}^l$. Solid and dashed lines indicate the paths of L_C and L_P. Note that $\oplus$ is the convolutional concatenation operation in the network

where Conv $1D$ is the ordinary 1D convolution. We use this method to fit the dimensional difference between 1D convolutions and 2D convolutional filters:

$$F_{h,j}^{l+1} = \sum_{i,h} F_h^l \circ D_{i,j}^l \tag{6.14}$$

$$F_h^{l+1} = F_{h,1}^l \oplus \cdots \oplus F_{h,J}^l \tag{6.15}$$

where $\circ$ denotes the convolutional operation and h is the index of the feature map.

Backward Propagation Taking into account projection loss, we should train and update the real value filters C_i^l and the projection matrix W^l by these formulas as follows:

$$\delta_{C_i^l} = \frac{\partial L}{\partial C_i^l} = \frac{\partial L_S}{\partial C_i^l} + \frac{\partial L_P}{\partial C_i^l} \tag{6.16}$$

$$C_i^l \leftarrow C_i^l - \eta_1 \delta_{C_i^l} \tag{6.17}$$

$$\delta_{W_j^l} = \frac{\partial L}{\partial W_j^l} = \frac{\partial L_S}{\partial W_j^l} + \frac{\partial L_P}{\partial W_j^l} \tag{6.18}$$

$$W_j^l \leftarrow W_j^l - \eta_2 \delta_{W_j^l} \tag{6.19}$$

Note that η_1, η_2 is the learning rate of the convolutional filters and W_j^l. We could substitute PCNNs for ordinary convolutions due to the dimensional consistency of two continuous layers [15]. 1-bit WaveNets achieve a decrease in computational resources using this method.

6.4 Object Detection and Tracking

Object detection is the process of finding a target from a scene, while object tracking is the follow-up of a target in consecutive frames in a video. Deep learning-based object detection can generally be classified into two categories: two-stage and single-stage object detection. Two-stage detectors, for example, Faster R-CNN [49], FPN [29], and Cascade R-CNN [4], generate region proposals in the first stage and refine them in the second. In localization, R-CNN [13] utilizes the L2 norm between predicted and target offsets as the object function, which can cause gradient explosions when errors are significant. Fast R-CNN [12] and Faster R-CNN [49] proposed a smooth loss of L1 that keeps the gradient of large prediction errors consistent. One-stage detectors, e.g., RetinaNet [30] and YOLO [48], classify and regress objects concurrently, which are highly efficient but suffer from lower accuracy. Recent methods [51, 72] have been used to improve localization accuracy using IoU-related values (insertion over union) as regression targets. IoU loss [72] utilized the negative log of IoU as object functions directly, which incorporates the dependency between box coordinates and adapts to multi-scale training. GIoU [51] extends the IoU loss to nonoverlapping cases by considering the shape properties of the compared objects. CIoU loss [75] incorporates more geometric measurements, that is, overlap area, central point distance, and aspect ratio, and achieves better convergence.

Table 6.1 Results reported in Liu et al. [34]

Dataset	Index	SiamFC	XNOR	RB-SF
GOT-10K	AO	0.348	0.251	0.327
	SR	0.383	0.230	0.343
OTB50	Precision	0.761	0.457	0.706
	SR	0.556	0.323	0.496
OTB100	Precision	0.808	0.541	0.786
	SR	0.602	0.394	0.572
UAV123	Precision	0.745	0.547	0.688
	SR	0.528	0.374	0.497

Sun et al. [54] propose a fast object detection algorithm based on BNNs. Compared to full-precision convolution, this new method results in 62 times faster convolutional operations and 32 times memory saving in theory.

Liu et al. [34] experiment on object tracking after proposing RBCNs. They used the SiamFC network as the backbone for object tracking and binarized the SiamFC as the rectified binary convolutional SiamFC network (RB-SF). They evaluated RBSF in four datasets, GOT-10K [22], OTB50 [65], OTB100 [66], and UAV123 [40], using accuracy occupy (AO) and success rate (SR). The results are shown in Table 6.1.

Yang et al. [69] propose a new method to optimize a deep neural network based on YOLO-based object tracking simultaneously using approximate weight binarization, trainable threshold group binarization activation function, and separable convolution methods according to depth, significantly reducing the complexity of computation and model size.

6.4.1 Data-Adaptive Binary Neural Networks for Efficient Object Detection and Recognition

One of the ugliest aspects of 1-bit CNNs lies in the gap between full-precision weights and their quantization counterpart. The focus of the existing methods is to minimize the gap. To this end, the convolutional kernel is usually divided into two parts, the amplitude and direction, while the feature maps are only in the direction for efficient calculation. The existing binarization methods can be formulated in a unified framework where (1) D_i^l are the directions of the full-precision kernels W_i^l of the ith channel in the lth convolutional layer, $i \in \{1, \cdots, I\}$, $l \in \{1, \cdots, N\}$; (2) A^l is shared by all D_i^l represents the amplitude of the lth convolutional layer; and (3) $\hat{A}_i^l$ and A_i^l are of the same size and all the elements of $\hat{A}_i^l$ are equal to the average of the elements of A_i^l. $\hat{X}_i^l$ denotes the direction of input X_i^l. In the forward pass, $\hat{A}^l$ is used instead of the full-precision weights A^l. The full-precision weights A^l are only used for back propagation during training. Note that the formulation can represent XNOR based on a scalar [47], and also simplified PCNN [15] whose scalar is learnable as a projection matrix, or even XNOR++, which decomposes tensor $\hat{A}$ into vectors in the three directions of the channel, height, and width [3]. We represent $\hat{W}$ by the amplitude and direction as:

$$\hat{W} = D \odot \hat{A}, \tag{6.20}$$

where $\odot$ denotes the element-wise multiplication between matrices. We can calculate the binary convolution output $\hat{O}$ as:

$$\hat{O} = \hat{X} * \hat{W} = (\hat{X} \circledast D) \odot \hat{A}, \tag{6.21}$$

where $*$ and $\circledast$ denote convolution (floating-point multiplication and addition) and bit convolution (bitwise XNOR and pop-count operations), respectively.

6.4.1.1 Data-Adaptive Amplitude Method

Existing methods fail to calculate the data-adaptive amplitude to better approximate the full-precision feature maps. This explains the primary reason for the performance gap between 1-bit CNNs and their full-precision counterpart. Without considering the amplitude of X, there is an inevitable gap between $\hat{O}$ and O because the fixed $\hat{A}$ is irrelevant to input X. To address this issue, an intuitive idea is to let $\hat{A}$ become a function $\hat{A}(X)$ with X as the input. In 1-bit CNNs, we use $\hat{X} \circledast D$ to substitute for X because $\hat{X} \circledast D$ contains the information of both X and W, which will have a better representation capacity. Because the amplitude $\hat{A}$ is not fixed but adaptive to the input data, we call our method data-adaptive binary neural network (DA-BNN) [73]. And we have:

$$\hat{O} = (\hat{X} \circledast D) \odot \hat{A}(\hat{X} \circledast D), \tag{6.22}$$

where $\hat{A}(.)$ is related to the input and will burden the computation of CNNs. To address this problem, we use attention-based methods [60, 64] and introduce a lightweight module to implement $\hat{A}(.)$. The module is designed by considering both channel and spatial (height $\times$ width) levels. For simplicity, we denote $\hat{X} \circledast D$ as $\hat{M}$ in the following sections. To solve the problem, we introduce the attention method to calculate a data-adaptive amplitude for better performance. We lead two data-adaptive amplitude methods: channel-based and spatial-based.

6.4.1.2 Data-Adaptive Channel Amplitude

To calculate the channel amplitude $\hat{A}_C(\hat{M})$, we consider the feature maps from two perspectives, within and between channels, similar to the attention mechanism [21, 60]. We introduce the global average pooling layer to reduce the other two dimensions and extract features within channels. Compared with the convolution,

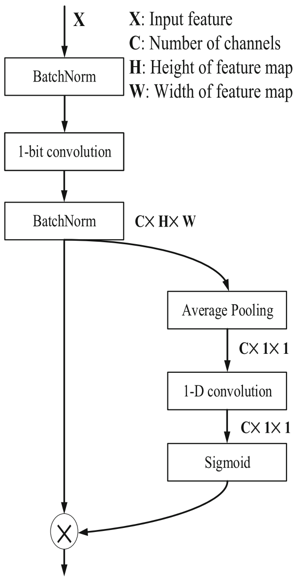


Fig. 6.5 The calculation of the channel amplitude

the global average pooling layer adds no extra parameters and fewer calculations. Considering the cross-channel interaction, a 1D convolution is applied to fuse the information of each channel with its neighbors.

However, because real-valued convolution parameters are often nearly zero, easily influenced by the weight decay, the binarization of parameters always means an amplification compared with the real-value convolution. The result of the binary convolution is usually much more significant when compared with the corresponding real-valued convolution [36, 53]. Thus, the amplitude $\hat{A}(\hat{M})$ should be a small value, solved by a sigmoid function that maps the amplitude into (0, 1). Furthermore, the sigmoid function is also used to guarantee $\hat{A}(.)$ merely learns the amplitude information, not the direction. By doing so, we represent the channel amplitude $\hat{A}_C(\hat{M})$ as:

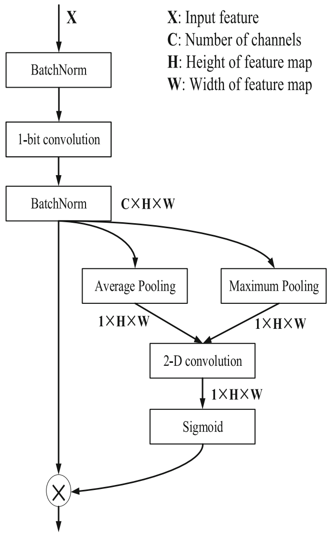
$$\hat{A}_C(\hat{M}) = \sigma(k_c * AvgPool(\hat{M})), \tag{6.23}$$

where σ denotes the sigmoid function and k_c is the kernel of 1D convolution. A specific module is illustrated in Fig. 6.5.

6.4.1.3 Data-Adaptive Spatial Amplitude

Similar to calculating the attention network, we use pooling layers and convolution to calculate the spatial amplitude. In Fig. 6.6, we demonstrate the corresponding structure. We utilize the average and maximum pooling together and then use a

Fig. 6.6 The calculation of spatial amplitude



3×3 convolution instead of the 1D convolution to deal with the spatial data. We calculate the spatial amplitude $\hat{A}_S(\hat{M})$ as:

$$\hat{A}_S(\hat{M}) = \sigma(k_s * \left[AvgPool(\hat{M}); MaxPool(\hat{M})\right]). \tag{6.24}$$

However, when the features are binarized in the next block, the amplification information will be eliminated, and only the direction information will be retained. To keep the amplitude information, we redistribute features using an additional BN, added before the binarization of the feature map. By doing so, the amplitude will be partially converted into the direction information by improving the feature distribution.

6.4.1.4 Experiment on Object Recognition

We use ImageNet [8] to train our models. Considering the size of the dataset, we apply ResNet-18 [19] on ImageNet for a fair comparison with other quantization networks.

ILSVRC12 ImageNet is a large-scale dataset that contains over 1.2 million training images and 50K validation images from 1000 categories. To train ResNet-18 on ImageNet, models are trained in a two-step training method, similar to [36–38]. The training process is divided into two stages. In the first stage, we train a full-precision network that keeps the weights and activations real-valued for 60 epochs. Networks are optimized using stochastic gradient descent (SGD) to stabilize the pre-trained model. At this stage, we set the weight decay to 3e−4, the momentum to 0.9, and the learning rate to 0.1. In the second stage, the network loads the parameters and binarizes the weights and activations. An Adam optimizer is used to sufficiently train the binary model, following the settings of [37]. The learning rate is set to 1e−3, and the weight decay is fixed to 0. In both stages, the batch size is 360, and the learning rate is adjusted following a cosine schedule until annealing down to 0. Following the settings in [38], we use PReLU activations [18] instead of the ReLU activations and keep the real-valued downsample and double skip connections.

6.4.1.5 Ablation Study on Object Recognition

In this section, we evaluate the effects of data-adaptive amplitude on the performance of 1-bit CNNs.

The BN layer in ResNet-18 is set after the convolution. We, however, add an extra BN layer in front of the 1-bit convolution to redistribute the features and turn the information of amplitude into the direction to cope with the information loss in the binarization process. We test the effectiveness of the addition of BN on ImageNet by ResNet-18. The channel amplitude and spatial amplitude are used in parallel. The specific structure is demonstrated in Fig. 6.7 and the results are shown in Table 6.2:

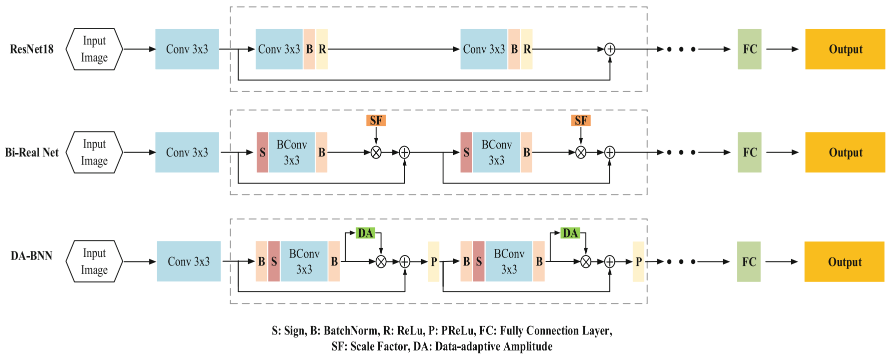


Fig. 6.7 Network architectures of ResNet-18, Bi-Real Net on ResNet-18, and DA-BNN on ResNet-18. Note that the scale factor is added to the convolution in Bi-Real Net, and we adjust its position to make the comparison clear with the same principle

Table 6.2 Different structures of binary neural networks are tested on ImageNet ILSVRC12. “BN” refers to the use of the front batch normalization layer. All the models are based on ResNet-18

Binary	BN	Amplitude	Acc.
×	×	×	69.30
√	×	×	57.60
√	√	×	59.32
√	√	√	**63.08**

The bolds denote the best results

Table 6.3 We apply the channel and spatial adaptive amplitude, respectively, to evaluate their effectiveness. All models are trained in a two-step method. The accuracy of step 1 corresponds to the real-valued model, while the accuracy of step 2 corresponds to the binarization counterpart

Method	Acc.of step1	Acc. Of step2
Baseline	67.73	57.60
Spatial	67.36	60.18
Channel	69.32	61.63
Channel+spatial	**69.41**	62.48
Channel & spatial	69.29	**63.08**

The bolds denote the best results

directly using the additional front BN, an increase of 1.72% in accuracy is observed compared to the normal binarization in ResNet-18. If we add the adaptive amplitude, the accuracy of networks can be improved by about 4%, which proves that the additional BN is helpful for further improving the performance of binary neural networks.

In Eq. 6.22, the data-adaptive amplitude $\hat{A}(.)$ can be calculated using the channel amplitude and spatial amplitude in sequence or parallel. To this end, we test three data-adaptive amplitude combinations of the channel amplitude and spatial amplitude: sequential channel-spatial, sequential spatial-channel, and parallel, using both amplitude modules. Scale factor methods are also tested for comparison. We evaluate the performance of data-adaptive amplitude on ImageNet, based on ResNet-18. Unlike the experiments in Table 6.3, a two-step training method is used here. In the first stage, we train the model with real-valued weights and feature maps as a pre-training step. In the second step, the model loads the parameters trained in the first step and binarizes the weights and features corresponding to the binary models. We record the best performance of the model at each stage.

Figure 6.8 shows the curves for the Top-1 accuracy of different methods. The sudden drop of the curves denotes the switch of training stages from full-precision to binarized models. The best accuracy of different methods is illustrated in Table 6.3: spatial amplitude has little influence on the real-valued model but increases the performance of binary neural network by about 2.5%, whereas channel amplitudes can improve the accuracy of both full-precision model and binarized model by 2% and 4%, respectively. Different configurations of adaptive amplitude methods influence the performance of full-precision models and binary models differently. The sequential channel and spatial method performs better on full-precision models, while the one in parallel performs better for binary models. These results verify that a proper arrangement of the amplitude methods is essential to further improve the performance.

6.4.1.6 Network Accuracy Comparison on ImageNet

To evaluate the performance of DA-BNN, we compare its performance with other state-of-the-art quantized networks, including BWN [47] DoReFa-Net [76],

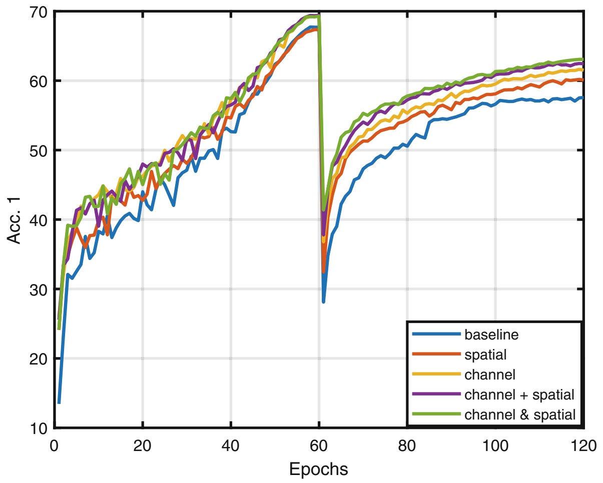


Fig. 6.8 The convergence curves of DA-BNN. The "baseline" label means no extra modules are used. We use our proposed modified structure. The "channel" and "spatial" labels denote applying the corresponding amplitude modules. The "channel + spatial" label and "channel & spatial" represent different combination methods. It should be noted that the sudden drop in 60 epochs is caused by switching the training steps when both weights and activations are binarized

TBN [57], BNN [6], XNOR-Net [47], ABC-Net [33], Bi-Real Net [37], PCNN [15], BONN [17], CI-Net [62], BinaryDuo [25], real-to-binary [38], and ReActNet [36] and reported Top-1 and Top-5 accuracies in Table 6.4. Note that all models are based on ResNet-18 with 69.3% Top-1 accuracy on the full-precision model.

We first applied our DA-BNN based on Bi-Real Net, achieving outstanding performance among neural networks with binary weights and activations. However, Bi-Real Net focuses more on optimizing binarization and ignores the significance of amplitude in 1-bit convolution, just using the mean of the real-valued weights as the scale factor. In contrast, our method focuses on the adaptive amplitude to improve the representation capacity, an essential enhancement to Bi-Real Net. By applying our DA-BNN on Bi-Real Net, we use our adaptive amplitude instead of the scale factor and modify those above. Above 6% improvement is achieved under the Bi-Real Net framework, which exceeds most binarization methods. It is also worth mentioning that DoReFa-Net and TBN use more than 1-bit to quantify activations, yet we still perform better.

However, our DA-BNN is not limited to a specific quantization method and can be combined with other binarization methods for more significant improvement. To further evaluate the potential of our DA-BNN, we combine it with ReActNet, which achieves the highest binary accuracy based on ResNet-18 on ImageNet, to the best of our knowledge. Note that for a fair comparison, we remove the scale factor used in ReActNet to ensure the used amplitude is learned in our adaptive methods. Based on the ReActNet, it obtains even higher accuracy, with just a 3% gap to the full-precision model.

Table 6.4 Accuracy of state-of-the-art quantization networks and our DA-BNN on ImageNet. "W" and "A" refer to the weight and activation of the bit width, respectively. All the models are based on ResNet-18

Model	W/A(bit)	Top-1	Top-5
ResNet-18	32/32	69.3	89.2
BWN	1/32	60.8	83.0
DoReFa-Net	1/4	59.2	81.5
TBN	1/2	55.6	79.0
BNN	1/1	42.2	67.1
XNOR-Net	1/1	51.2	73.2
ABC-Net	1/1	42.7	67.6
Bi-Real Net[a]	1/1	56.4	79.5
PCNN	1/1	57.3	80.0
BONN	1/1	59.3	81.6
CI-Net	1/1	59.9	84.2
BinaryDuo[a]	1/1	60.9	82.6
DA-BNN[a] (based on Bi-Real Net)	1/1	**63.1**	**84.3**
Real-to-binary[a]	1/1	65.4	–
ReActNet[a]	1/1	65.5	–
DA-BNN[a] (based on ReActNet)	1/1	**66.3**	**86.7**

The bolds denote the best results

[a]A real-valued or partly real-valued (just binarized activation) model is used for pre-training. Because our method is not specific to the quantitative process, we use two different binarization frameworks, Bi-Real Net and ReActNet. Note that the first DA-BNN is based on Bi-Real Net, and the experimental settings refer to the description above. As for ReActNet, we keep all the settings the same except for the scale factor change with our data-adaptive amplitude

In short, we achieve a new state-of-the-art performance compared to other BNNs, and a much closer performance to full-precision models, which validates the superiority of DA-BNN for the BNN calculation.

6.4.1.7 Experiment on Object Detection

We evaluate our method on the PASCAL VOC dataset, composed of natural images from 20 classes. We train our model on the VOC 2007 and VOC 2012 trainval sets, which consist of around 16k images, and we evaluate our method on the VOC 2007 test set, including about 5k images. Following the setting of [9], we use the mean average precision (mAP) as the evaluation criterion.

We train our DA-BNN with the Faster R-CNN [50] detection framework with the ResNet-18 backbone [19] aforementioned. Following implementing binary neural networks in [37], we remain the first and last layer in the detection networks' real-valued. The same pipeline as [50] is utilized when training our DA-BNN with a Faster R-CNN detector. For efficient object detection, we binarize all the 3×3 convolution operations in the following models, except the first convolution and full-connected layer in Faster R-CNN, following the same settings as XNOR-Net [47] and BiDet [63]. We modify the architecture of ResNets following [63].

As for the details of training settings, we pre-train the binary backbone network in DA-BNN fashion on the ImageNet dataset, as depicted in Sect. 6.4.1.4. Then we fine-tune the backbone and detection parts collaboratively for the object detection task. The batch size is assigned to be 16, with the SGD optimizer applied. The number of epochs is 12, and the learning rate varies according to the framework and backbone. A multistep learning rate schedule is employed for the Faster-RCNN, which decays twice by multiplying by 0.1 at the 8th and 11th epoch of the 12 epochs (Table 6.5 and Fig. 6.9).

6.4.1.8 Performance Comparison on PASCAL VOC

In this section, we compare the proposed DA-BNN with state-of-the-art 1-bit neural networks, including XNOR-Net [47], Bi-Real Net [37], and BiDet [63] for the task of object detection on the PASCAL VOC datasets.

Table 6.5 Comparison of mAP (%) with state-of-the-art BNNs in Faster R-CNN frameworks with ResNet-18 on VOC `test2007`. The detector with the real-valued and multi-bit backbone is given for reference. Input resolution is set as 600 × 1000. The bold denotes the best result

Quantization method	W/A(bit)	mAP(%)
Full-precision	32/32	74.5
XNOR-Net	1/1	48.9
Bi-Real Net	1/1	58.2
BiDet	1/1	59.5
DA-BNN	1/1	**63.5**

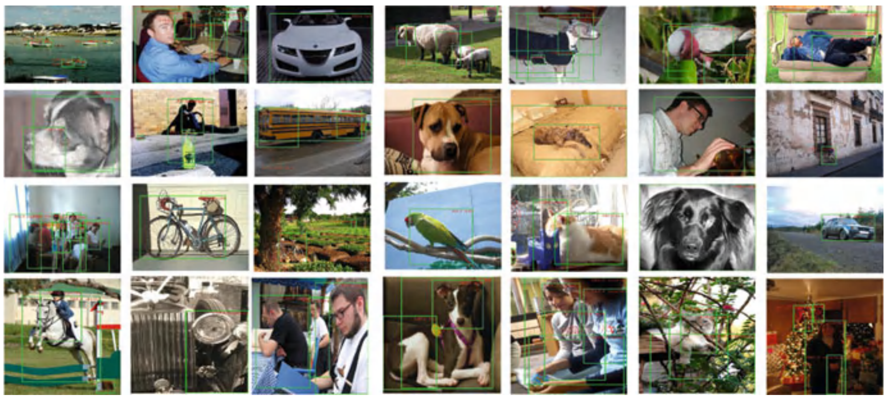


Fig. 6.9 Qualitative results on PASCAL VOC `test2007` (best viewed in color)

Compared to other 1-bit methods, we observe a significant performance advantage over other state of the arts. With the ResNet-18 backbone, we achieve 63.5% mAP, outperforming XNOR-Net, Bi-Real Net, and BiDet by 14.6%, 5.3%, and 4.0% mAP with the similar memory usage and FLOPs.

In short, we achieved a new state-of-the-art performance compared to other BNNs on PASCAL VOC. We are also much closer in performance to full-precision models, as demonstrated in experiments, validating the superiority of DA-BNN.

6.4.1.9 Computation and Storage Analysis

Inevitably, using a data-adaptive amplitude will increase the computation and storage for a more accurate approximation to real-valued convolution. However, the additional structures are lightweight and efficient. The additional part is negligible compared to the computation and storage of 1-bit convolution. In detail, the additional storage in adaptive amplitude is the weight of the convolution. In channel amplitude, we use a simple 1D convolution with a size of three, and thus the number of additional parameters is 3×32 (32 denotes 32 bits). A 3×3 convolution is used in the spatial amplitude with two input and one output channels. Thus, its storage is $9 \times 2 \times 32$. Both are far less than the storage of corresponding 1-bit convolution due to the large numbers of convolution channels. The primary source of the extra computation comes from the structure modification, where just a few parameters are introduced compared to the whole model. So the storage increase has almost no influence on the storage of original 1-bit networks.

We calculate the computational and storage complexity compared to BNN networks and full-precision networks to show the ignorable addition of memory and speed up during inferences. The memory usage is represented by the storage for parameters of networks, which is calculated as the summation of 32-bit times real-valued parameters and 1-bit times binary parameters. We use FLOPs to measure computational complexity. Referring to [37, 47], the acceleration of 1-bit convolution is about 64 times the real-valued convolution. We follow these methods and calculate corresponding FLOPs.

Table 6.6 compares computational complexity, and storage cost, across different quantization methods on ResNet-18 and Faster R-CNN frameworks. The proposed DA-BNN saves the storage cost by $11.04\times$ and reduces the computation by $10.80\times$ in ResNet-18. On Faster R-CNN, as a result of the decrease of full-precision parameters in the fully connected layer, better performance of saving the storage and computation by $18.62\times$ and $15.77\times$, respectively, has been achieved, which keeps the same level as other 1-bit CNN methods. In summary, our adaptive amplitude introduces negligible storage (less than 1%) and little computation (less than 8%) but can significantly enhance BNNs' performance.

Table 6.6 We show the memory usage as well as the flops of the DA-BNN. The calculation method is the same as Bi-Real Net

Model	Method	Memory usage	Memory saving	FLOPs ($\times 10^8$)
ResNet-18	Full-precision	374.1 Mbit	–	18.26
	XNOR-Net	33.7 Mbit	11.10×	1.67
	Bi-Real Net	33.6 Mbit	11.14×	1.63
	Channel	33.9 Mbit	11.04×	1.65
	Spatial	33.9 Mbit	11.04×	1.67
	DA-BNN	33.9 Mbit	11.04×	1.69
Faster R-CNN	Full-precision	379.9 Mbit	–	360.14
	XNOR-Net	20.2 Mbit	18.81×	21.29
	Bi-Real Net	20.1 Mbit	18.90×	21.27
	BiDet	20.1 Mbit	18.90×	21.27
	DA-BNN	20.4 Mbit	18.62×	22.84

6.4.2 Amplitude Suppression and Direction Activation in Networks for Faster Object Detection

6.4.2.1 Methodology

In this paper, we propose an amplitude suppression and direction activation in the Faster R-CNN framework (ASDA-FRCNN) [68] to compress DCNNs for highly efficient object detection. The shared amplitude between full-precision and quantized kernels is significantly suppressed during binarization, which can lead to a new simple but effective loss. The concept of ASDA is generic and flexible and can be easily incorporated into existing DCNNs such WideResNets and ResNets and applied to many vision tasks including object classification.

Problem Formulation

The inference process of any binary neural network (BNN) model is based on the binarized kernels. This means that the kernels must be binarized in the forward step (corresponding to the inference) during training, so that the training loss is calculated based on the binarized filters. Unlike the forward process, during back propagation, the resulting kernels do not need to be binarized and can be full-precision. In this case, the full-precision kernels are binarized to gradually bridge the binarization gap during training. Therefore, the learning of most BNN models involves both discrete and continuous spaces, which poses a great challenge in practice.

To address these challenges and improve the optimization of binarizing CNNs, we decouple the full-precision kernel X and represent it by the amplitude and direction as:

$$\hat{X} = A \cdot D, \tag{6.25}$$

where A and D respectively denote the amplitude and the direction of X. D is the ℓ_1-normalized matrix and calculated by sign(X) as $-\frac{1}{size(X)}$ for negative X and $\frac{1}{size(X)}$ for positive X. A is a scalar.

Corollary 6.1 *To obtain an optimized BNN, we solve:*

$$X = \hat{X} = A \cdot D, \tag{6.26}$$

based on the assumption that X and $\hat{X}$ share similar amplitude.

This corollary is a bilinear problem, where A and D need to be calculated simultaneously. Existing methods tend to split the problem into easily solved subproblems, and then solve them using the alternating direction method of multipliers (ADMM) [20, 70], which might be less efficient for the BNN calculation. To simplify the process, we proposed to calculate the amplitude A based on the back propagation algorithm since D can be solved based on the sign(.) function. In addition, due to the shared amplitude between the full-precision kernels and binarized kernels, we can easily suppress it and thus lead to a highly efficient detector.

Forward Propagation in ASDA-FRCNN

In order to achieve binarized weights, we design a new loss function in ASDA-FRCNN. Note that only the kernels of ASDA-FRCNN are binarized, while for 1-bit ASDA-FRCNN, both the kernels and the activations are binarized. These are briefly described at the end of Sect. 6.4.2.2. Here we define D, A, and A as follows. $D^l_{i,j}$ is the direction of the full-precision kernel $X^l_{i,j}$. $X^l_{i,j}$ denotes the i-th kernel in the j-th filter at l-th convolutional layer, $l \in \{1, \cdots, N\}$; A^l shared by all $D^l_{i,j}$ represents the amplitude of the l-th convolutional layer; A^l and A^l are of the same size and all the elements of A^l are equal to the average of the elements of A^l. In the forward pass, A^l is used instead of the full-precision A^l. In this situation, A^l can be considered a scalar. The full-precision A^l is only used for back propagation during training. This process is the same as the way of calculating $\hat{X}$ from X in an asynchronous manner, which is also illustrated in Fig. 5.5.

Accordingly, Eq. 6.25 is represented for ASDA-FRCNN at l-th layer as:

$$D^l_{i,j} = \frac{\text{sign}(X^l_{i,j})}{size(X^l_{i,j})}, \tag{6.27}$$

$$\hat{X}^l_{i,j} = A^l \cdot D^l_{i,j}, \tag{6.28}$$

where $D^l_{i,j}$ represents the binarized kernel, *i.e.*, direction. $size(X^l_{i,j})$ is the number of weights of $size(X^l_{i,j})$. With the i-th binary kernel in j-th filter at l-th layer reconstructed, we can formulate the forward path of feature maps as:

$$\begin{aligned} F_j^{l+1} &= \mathcal{H}_j^{l+1}(\boldsymbol{F}^l, A^l, \boldsymbol{D}^l) \\ &= \sum_i F_i^l \otimes \hat{X}_{i,j}^l \\ &= A^l \sum_i F_i^l \otimes D_{i,j}^l, \end{aligned} \tag{6.29}$$

where we use $\mathcal{H}^{l+1}$ to denote the mapping at $l+1$-th layer in the abstract sense and $\mathcal{H}_j^{l+1}$ is the j-th output feature map. $\otimes$ denotes the convolution operation; F_j^{l+1} is the j-th feature map in the $(l+1)$ aligned convolutional layer. $\boldsymbol{F}^l$, $\boldsymbol{D}^l$ denotes the aggregate of feature maps and directions at l-th layer, respectively. F_i^l denotes the i-th feature map in the lth convolutional layer.

Loss Function of ASDA-FRCNN
We then define an amplitude loss function to reconstruct the full-precision kernels as:

$$\begin{aligned} L_A &= \sum_l \sum_j \sum_i \|X_{i,j}^l - \hat{X}_{i,j}^l\|_2^2 \\ &= \sum_l A^l \sum_j \sum_i \|\mathcal{X}_{i,j}^l - D_{i,j}^l\|_2^2, \end{aligned} \tag{6.30}$$

$\mathcal{X}_{i,j}^l$ is normalized by dividing $\|X_{i,j}^l\|_1$. Under Corollary 6.1, X and $\hat{X}$ share similar amplitude, thus formulating a strong supervision to minimize the reconstruction error. Then we also need a loss to monitor the detection process as:

$$\begin{aligned} L_S =& \frac{1}{S}\sum_k L_{cls}(p_k, p_k^{gt}) + \lambda \frac{1}{M}\sum_k p_k^{gt} L_{reg}(t_k, t_k^{gt}) \\ =& \frac{1}{S}\sum_k -\log\left[p_k^{gt}\cdot p_k + (1-p_k^{gt})\cdot(1-p_k)\right] \\ &+ \lambda\frac{1}{M}\sum_k p_k^{gt} \text{smooth}_{\ell_1}(t_k, t_k^{gt}), \end{aligned} \tag{6.31}$$

where S denotes the mini-batch size and M denotes the anchor locations. p_k, t_k, are a positive prediction and a vector presenting four coordinates of anchor k. Their detailed definitions are:

$$p_k = P_s\left[\mathcal{H}^N(\boldsymbol{F}^{N-1}, A^{N-1}, \boldsymbol{D}^{N-1})\right] \tag{6.32}$$

$$t_k = T_k\left[\mathcal{H}^N(\boldsymbol{F}^{N-1}, A^{N-1}, \boldsymbol{D}^{N-1})\right], \tag{6.33}$$

p_k^{gt} and t_k^{gt} are their ground truth labels, respectively. P_k and T_k denote obtaining the probability and location information of k-th anchor from last layer. N is the total number of layers and the function $\text{smooth}_{\ell_1}(x)$ is defined as:

$$\text{smooth}_{\ell_1}(x) = \begin{cases} 0.5 \cdot x^2, & \text{if } |x| < 1 \\ |x| - 0.5, & \text{else} \end{cases} \tag{6.34}$$

Finally, the overall loss function L is applied to supervise the training of ASDA-FRCNN in the back propagation algorithm and is defined as:

$$L = L_S + \mu L_A, \tag{6.35}$$

6.4.2.2 Back Propagation

In ASDA-FRCNN, what needs to be learned and updated are the full-precision kernels X_i and the amplitude A. The kernels and the matrices are jointly optimized. In each convolutional layer, ASDA-FRCNN updates the full-precision kernels and then the amplitude. In what follows, the layer index l is omitted for simplicity.

Updating X

We denote δ_{X_i} as the gradient of the full-precision kernel $X_{i,j}^l$ and have:

$$X_{i,j}^l \leftarrow X_{i,j}^l - \eta_1 \delta_{X_{i,j}^l}, \tag{6.36}$$

where η_1 is a learning rate. $\delta_{X_{i,j}^l}$ is calculated as:

$$\begin{aligned} \delta_{X_{i,j}^l} &= \frac{\partial L_S}{\partial X_{i,j}^l} + \frac{\partial L_A}{\partial X_{i,j}^l} \\ &= \frac{\partial L_S}{\partial \hat{X}_{i,j}^l} \cdot \frac{\partial \hat{X}_{i,j}^l}{\partial X_{i,j}^l} + 2 \cdot A(\mathcal{X}_{i,j}^l - D_{i,j}^l) \frac{\partial \mathcal{X}_{i,j}^l}{\partial X_{i,j}^l} \\ &= A \cdot \left[\frac{\partial L_S}{\partial \hat{X}_{i,j}^l} \cdot \mathbb{1} + 2(\mathcal{X}_{i,j}^l - D_{i,j}^l) \frac{\partial \mathcal{X}_{i,j}^l}{\partial X_{i,j}^l} \right], \end{aligned} \tag{6.37}$$

$X_{i,j}^l$ is the full-precision convolutional kernel corresponding to $D_{i,j}^l$, and $\mathbb{1}$ is the indicator function [47] widely used to estimate the gradient of the non-differentiable function.

6.4.2.3 Amplitude Calculation and Suppression

After updating X, we update the amplitude A. Let δ_A be the gradient of A. According to Eq. 6.35, we have:

$$A^l \leftarrow |A^l - \eta_2 \delta_{A^l}|, \tag{6.38}$$

where η_2 is another learning rate. And δ_{A^l} is calculated as:

$$\begin{aligned}\delta_{A^l} &= \frac{\partial L_S}{\partial A^l} + \frac{\partial L_A}{\partial A^l} \\ &= \sum_j \sum_i \left[\frac{\partial L_S}{\partial \hat{X}^l_{i,j}} \cdot D^l_{i,j} + \|X^l_{i,j} - D^l_{i,j}\|_2^2 \right],\end{aligned} \tag{6.39}$$

Note that the amplitudes are always set to nonnegative. By setting a very small μ in Eq. 6.35, we actually suppress amplitude A^l directly. The parameter evaluation is extensively explored in the experimental section, which shows that such suppression is highly effective. On the contrary, the direction information is always used in the forward process.

Our 1-bit ASDA-FRCNN is also based on binarizing the kernels and activations simultaneously as in [7, 47]. These derivations show that ASDA-FRCNN is learnable with our BP algorithm. We summarize the training procedure in Algorithm 13.

Algorithm 13: Optimized ASDA-FRCNN via back propagation

```
Input:
    The training dataset; the full-precision kernels $X$; the amplitude scalar $A$; the learning
    rates $\eta_1$ and $\eta_2$.
Output:
    The ASDA-FRCNN with the learned $X$, $A$.
 1: Initialize $X$ and $A$ randomly;
 2: repeat
 3:     // Forward propagation
 4:     for $l = 1$ to $L$ do
 5:         Compute $\hat{X}^l_{i,j}$ from Eqs. 6.27 to 6.28;
 6:         if 1 bit feature maps then
 7:             $F^l_{i,j}$ =sign($F^l_{i,j}$);
 8:         end if
 9:         Compute $F^{l+1}_j$ via Eq. 6.29, $\forall i, j$;
10:     end for
11:     // Backward propagation
12:     for $l = L$ to 1 do
13:         Calculate $\delta_{X^l_{i,j}}$, $\delta_{A^l}$; // using Eqs. 6.36~6.39
14:         Update parameters $X^l_{i,j}$ and $A^l$ using back propagation;
15:     end for
16: until the algorithm converges.
```

6.4.2.4 Experiments

Datasets and Implementation Details

We evaluated our ASDA-FRCNN method on two most widely applied detection datasets: PASCAL VOC and MS COCO. PASCAL VOC 2007 [9] dataset consists of about 5k train/val images and 5k test images over 20 object categories. We also provide results by training on PASCAL VOC 2007+2012 train/val and testing on PASCAL VOC 2007 test. More experiments are deployed on MS COCO 2014 [31], which consists of 240k train/val images, 5k minival images, and 40k test-dev images over 80 object categories. Furthermore, as our approach shows great feasibility, we deploy ASDA ResNet-18 on ImageNet ILSVRC2012 [27] in ResNet-18 [19].

We implemented the training process plotted in Algorithm 13 on 3 NVIDIA TITAN Xp GPUs with 128GB of RAM via PyTorch [45]. The weight decay, momentum, and hyperparameter λ are set as 0.0001, 0.9, and 10, respectively. W and A are the weight and activation, respectively. Full-precision model is implemented with 32-bit weight and 32-bit activation. And 1-bit ASDA Faster is implemented with 1-bit weight and 1-bit activation. We modify the architecture of ResNet-18 and ResNet-34 following [37] by substituting ReLU with PReLU [18], and the final results of our ASDA Res-18 are fine-tuned based on the pre-trained models with only kernel weights binarized, halving the learning rate during training. NOTE: 1-bit ASDA-FRCNN is employed in ablation study; thus, W and A are 1-bit.

6.4.2.5 Ablation Study

Parameter μ

As mentioned above, the proposed loss has the ability to control the process of quantization. Hyperparameter μ is introduced in Eq. 6.35 to balance the loss and suppress the influence of the amplitude. To evaluate the influence of μ, we deploy

Table 6.7 Test mAP on PASCAL VOC 2007 dataset in ResNet-18 backbone. Training method includes VOC2007 only and VOC2007+2012. The bolds represent the best results

	μ			
Model	$1e-4$	$5e-5$	$2e-5$	$1e-5$
1-bit ASDA-FRCNN VOC07	47.4	51.1	**54.6**	48.6
1-bit ASDA-FRCNN VOC07+12	56.3	61.5	**63.4**	61.1

Table 6.8 Test mAP on PASCAL VOC 2007 dataset in ResNet-34 backbone. Training method is VOC2007+2012

	μ			
Model	$5e-5$	$2e-5$	$1e-5$	$5e-6$
1-bit ASDA-FRCNN VOC07+12	54.1	60.2	**65.5**	61.7

The bolds denote the best results

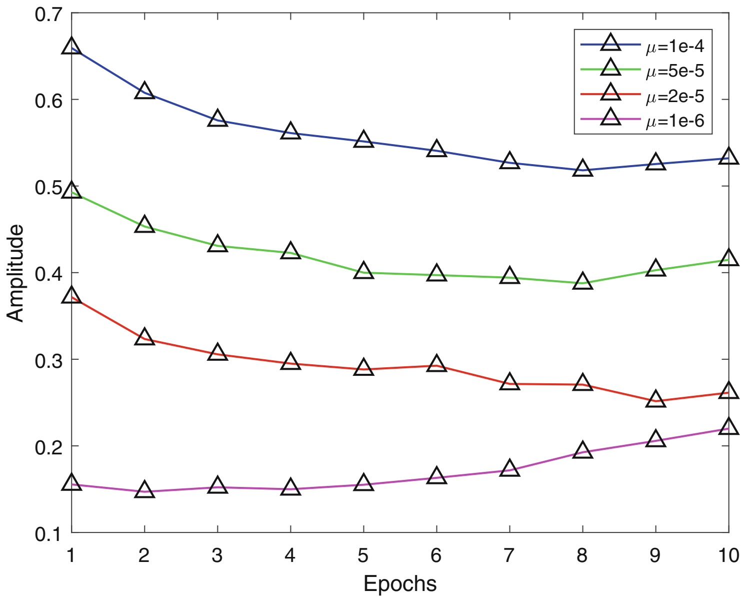


Fig. 6.10 A^l at first Conv layer in the sixth block with different μ. Training method is VOC2007+2012. Model is ResNet-18

controlled experiments on ResNet-18 on PASCAL VOC 2007. Results are shown in Tables 6.7 and 6.8.

In ResNet-18, it is observed that the network achieve the better performance as we suppress μ. Thus, we fix μ to $5e-4$ in the following experiments in ResNet-18. In addition, we analyze the amplitude of a certain layer in ResNet-18. As plotted in Fig. 6.10, A^l is suppressed more deeply as the μ becomes smaller, which subtly demonstrates our intuition in Sect. 6.4.2.3.

In ResNet-34, L_A increases as the model size expends. And the hyperparameter μ should be lower to suppress the amplitude more. As shown in Table 6.8, the network obtains the best performance when μ is set to $1e-5$. Thus, we fix μ to $1e-5$ in the following experiments in ResNet-34.

Learning Convergence

Figure 6.11 plots the L_S curve with different μ. Obviously, when μ is set to $2e-5$, L_S can converge to a lower level, which shows the network obtains a better performance.

Experimental Verification of Corollary 6.1

As plotted in Fig. 6.12, ℓ_2-norm summation of kernels in the first and second layer in 6-th block is similar to the corresponding amplitudes as the scatters distribute uniformly around the positive scale curve. This ablation result strongly proves our Corollary 6.1 (Fig. 6.13).

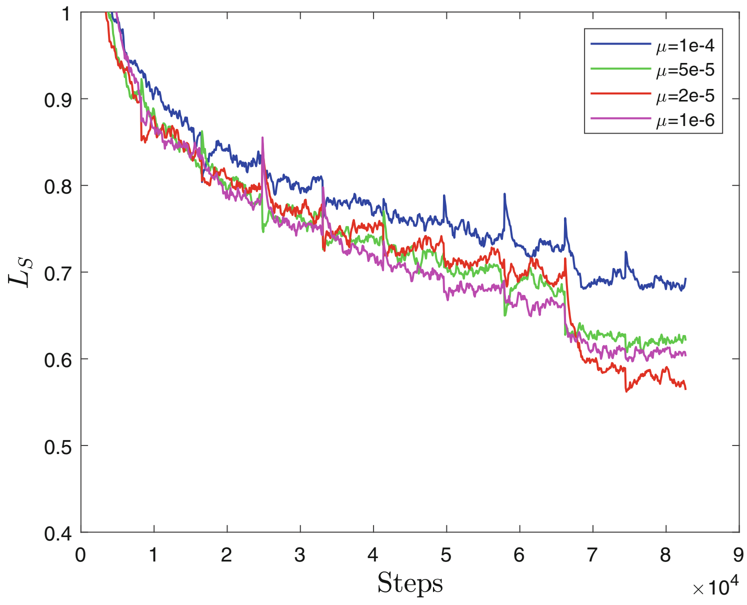


Fig. 6.11 Training L_S with different μ. Training method is VOC2007+2012. Model is ResNet-18

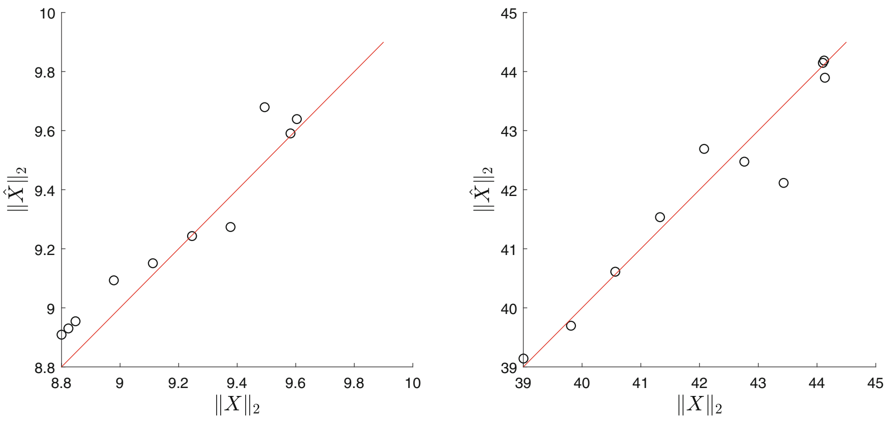


Fig. 6.12 Training $\|X\|_2$ and $\|\hat{X}\|_2$ scatter with epochs. Red line is the positive scale curve. The left one is the first layer in the sixth block and the right one is the second layer

6.4.2.6 Object Detection

Results on PASCAL VOC Datasets

We compare the performance of our results with other state-of-the-art binary methods such as XNOR [47], TBN [57], and Bi-Real [37]. The comparison results for object detection are illustrated in Tables 6.9 and 6.10.

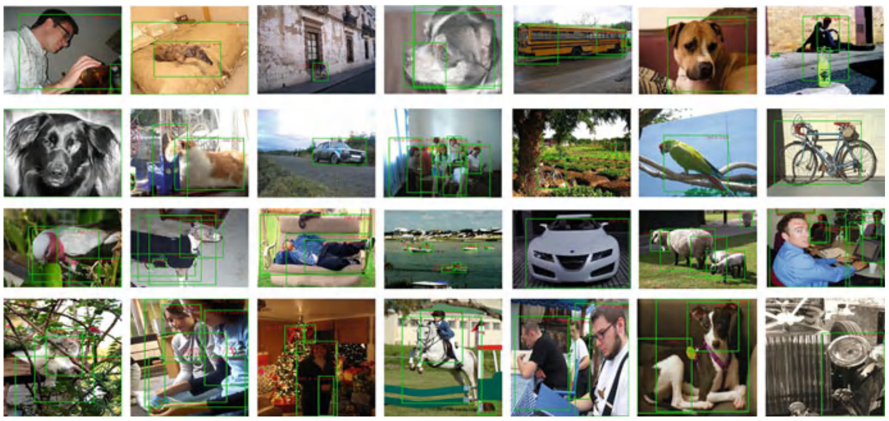


Fig. 6.13 Detection results on PASCAL VOC 2007 test

Table 6.9 Test mAP on PASCAL VOC 2007 dataset in ResNet-18 backbone. Training method is VOC2007 only and VOC2007+2012. "W" and "A" refer to the weight and activation bit width, respectively

Model	W	A	mAP	FPS
VOC2007 only				
Faster R-CNN-Res18	32	32	67.8	12.26
Bi-Real [37]	1	1	51.0	12.26[a]
ASDA-FRCNN	1	32	**56.6**	**12.26**
1-bit ASDA-FRCNN	1	1	**54.6**	**12.26**[a]
VOC2007+2012				
Faster R-CNN-Res18	32	32	73.2	12.26
Bi-Real [37]	1	1	60.6	12.26[a]
ASDA-FRCNN	1	32	**66.4**	**12.26**
1-bit ASDA-FRCNN	1	1	**63.4**	**12.26**[a]

The bolds denote the best results
[a]Due to hardware constraints, binary acceleration cannot be reflected on the PC, but theoretically it can accelerate 58 times. So we estimate FPS as 711

Table 6.10 Test mAP on PASCAL VOC 2007 dataset in ResNet-34 backbone. Training method is VOC2007+2012

Model	W	A	mAP	FPS
Faster R-CNN-Res34	32	32	75.6	8.01
XNOR [47]	1	2	54.7	–
TBN [57]	1	2	59.0	–
ASDA-FRCNN	1	32	**TBD**	**8.01**
1-bit ASDA-FRCNN	1	1	**65.5**	**8.01**[a]

The bolds denote the best results
[a] Estimated 464 FPS

It is observed that at least a 6.5% mAP as well as 1.45× acceleration improvement is gained with our 1-bit ASDA-FRCNN over TBN in ResNet-34. When μ is set to $2e-5$, the detection performance is the highest. In ResNet-18, we deploy Bi-Real [37] in the same experimental settings for contrast. Our 1-bit ASDA-FRCNN outperforms the other two methods with the same compression ratio.

We further plot the test AP of every class in the PASCAL VOC 2007 test. As is shown in Fig. 6.14, ASDA-FRCNN achieves a higher AP on all 20 classes than Bi-Real, and 1-bit ASDA-FRCNN outperforms Bi-Real in 16 out of 20 classes. Hence, we can conclude that ASDA Faster R-CNN achieves the better performance than Bi-Real.

Results on MS COCO Datasets
We use the μ value of $2e-5$ empirically. Then we compare the performance of our results with other state-of-the-art algorithms including one-stage fast object detection methods SSD [35], YOLO [48], RetinaNet [30] and CenterNet [77]. The comparison results for object detection are illustrated in Table 6.11.

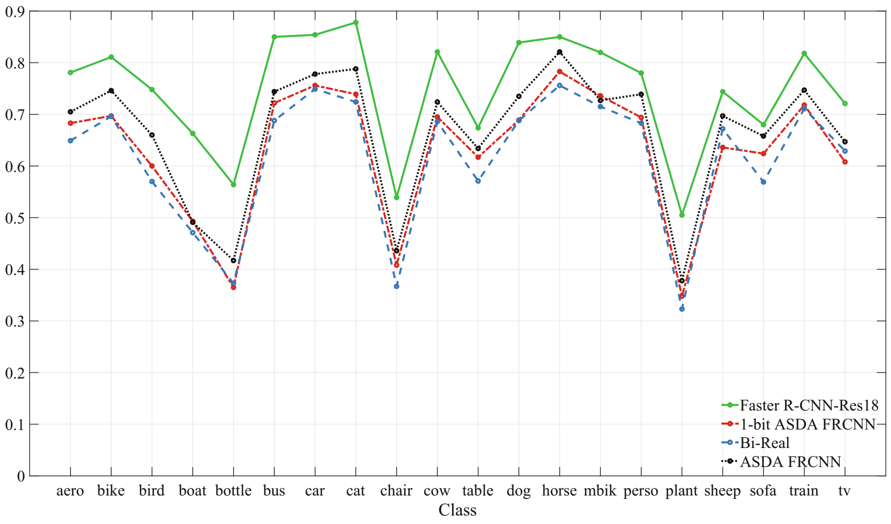


Fig. 6.14 APs of every class on PASCAL VOC 2007 test

Table 6.11 Test mAP@.5 and mAP@[.5, .95] on MS COCO test-dev in ResNet-18 backbone. Training method includes MS COCO Train+Val. Note that only ASDA-FRCNN is the binary approach

Model	W/A	mAP @.5	mAP @[.5, .95]	FPS
SSD321 [35]	32/32	45.4	28.0	61
YOLOv3-320 [48]	32/32	59.0	28.2	45
RetinaNet-50-500 [30]	32/32	59.0	32.5	14
CenterNet-Res18 [77]	32/32	44.9	28.1	142
Faster R-CNN [50]	32/32	42.7	21.9	6.25
ASDA-FRCNN	1/32	**41.5**	**21.4**	**6.25**
1-bit ASDA-FRCNN	1/1	**37.5**	**19.4**	**6.25**[a]

The bolds denote the best results
[a]Estimated 362 FPS

It is observed that our 1-bit ASDA-FRCNN is faster than the other one-stage detectors. It can process estimated 362 images per second, which is far more than other state-of-the-art methods. These results are very meaningful on mobile and embedded devices.

6.4.2.7 Image Classification

For ImageNet [27], we employ two data augmentation techniques sequentially: (1) randomly cropping patches of 224×224 from the original image and (2) horizontally flipping the extracted patches in the training. While in the testing, the Top-1 and Top-5 accuracies on the validation set with single center crop are measured.

In Table 6.12, we compare our ASDA ResNet-18 with several other state-of-the-art models. The first part of the comparison is based on ResNet-18 with 69.3% Top-1 accuracy on the full-precision model. Although BWN [47] and DoReFa-Net [76] achieve Top-1 accuracy with degradation of less than 10%, it should be noted that they apply full-precision and 4-bit activations, respectively. With both of the weights and activations binarized, the BNN model in [7], ABC-Net [33], and XNOR-Net [47] fail to maintain the accuracy and are inferior to our 1-bit ASDA Res-18. For example, compared with the result of PCNN [16], 1-bit ASDA Res-18 increases the Top-1 accuracy by 2.29%. Note that our algorithm still works very well on the classification task, which further validates the significance of our method. In short, we achieved a new state-of-the-art performance compared to other BNNs, which clearly validate the superiority of our method for the BNN computing.

Memory Usage and Efficiency Analysis
Memory use is analyzed by comparing our approach with the state-of-the-art XNOR-Net [47] and the corresponding full-precision network. The memory usage is computed as the sum of 32 bits multiplied by the number of full-precision kernels and 1 bit times the number of the binary kernels in the networks. As shown in Table 6.13, our proposed ASDA-FRCNN reduces the memory usage by 10.2× and 14.4× compared with the full-precision Faster R-CNN based on ResNet-18 and

Table 6.12 Test Top-1 and Top-5 accuracy on ImageNet ILSVRC2012 in ResNet-18. The bolds represent the best results

Model	W	A	Top-1	Top-5
ResNet-18 [19]	32	32	69.3	89.2
BWN [47]	1	32	60.8	83.0
DoReFa-Net [76]	1	4	59.2	81.5
TBN [57]	1	2	55.6	79.0
XNOR-Net [47]	1	1	51.2	73.2
BNN [7]	1	1	42.2	67.1
ABC-Net [33]	1	1	42.7	67.6
Bi-Real Net [37]	1	1	56.4	79.5
PCNN [16]	1	1	57.3	80.0
1-bit ASDA Res-18	1	1	**59.59**	**82.11**

Table 6.13 Memory usage and efficiency of convolution comparison on detection and classification binary and full-precision models

Model	Memory usage	Memory saving	Speedup
1-bit ASDA-FRCNN Res-18	35.5 Mbit	10.2×	58×
Faster R-CNN Res-18	361.3 Mbit	–	–
1-bit ASDA-FRCNN Res-34	46.5 Mbit	14.4×	58×
TBN Res-34 [57]	46.5 Mbit	14.4×	40×
Faster R-CNN Res-34	669.9 Mbit	–	–
1-bit ASDA Res-18	33.7 Mbit	11.1×	58×
XNOR-Net [47]	33.7 Mbit	11.1×	58×
ResNet-18	374.1 Mbit	–	–

ResNet-34, respectively. Our proposed ASDA ResNet-18 realizes 11.1× memory saving and 58× acceleration compared to the full-precision one. The reason is that the projection parameters W_j^l are only used when training for enriching the diversity in ASDA-FRCNN, whereas they are not used during inference. For efficiency analysis, if all of the operands of the convolutions are binary, then the convolutions can be estimated by XNOR and bitcounting operations, which gains 58× speedup in CPUs [47].

6.4.3 *Q-YOLO: Efficient Inference for Real-Time Object Detection*

6.4.3.1 Preliminaries

Network Quantization Process

We first review the main steps of the post-training quantization (PTQ) process and supply the details. Firstly, the network is trained or provided as a pre-trained model using full-precision and floating-point arithmetic for weights and activations. Subsequently, numerical representations of weights and activations are suitably transformed for quantization. Finally, the fully quantized network is either deployed on integer arithmetic hardware or simulated on GPUs, enabling efficient inference with reduced memory storage and computational requirements while maintaining reasonable accuracy levels.

6.4.3.2 Uniform Quantization

Assuming the quantization bit width is b, the quantizer $\mathrm{Q}(\mathbf{x}|b)$ can be formulated as a function that maps a floating-point number $\mathbf{x} \in \mathbb{R}$ to the nearest quantization bin:

$$\mathrm{Q}(\mathbf{x}|b) : \mathbb{R} \rightarrow \hat{\mathbf{x}}, \tag{6.40}$$

$$\hat{\mathbf{x}} = \begin{cases} \{-2^{b-1}, \cdots, 2^{b-1} - 1\} & \text{Signed,} \\ \{0 \cdots, 2^b - 1\} & \text{Unsigned.} \end{cases} \tag{6.41}$$

There are various quantizers $\mathrm{Q}(\mathbf{x}|b)$, where uniform [24] are typically used. Uniform quantization is well supported on most hardware platforms. Its unsigned quantizer $\mathrm{Q}(\mathbf{x}|b)$ can be defined as:

$$\mathrm{Q}(\mathbf{x}|b) = \mathrm{clip}(\lfloor \frac{\mathbf{x}}{s_{\mathbf{x}}} \rceil + zp_{\mathbf{x}}, 0, 2^b - 1), \tag{6.42}$$

where $s_{\mathbf{x}}$ (scale) and $zp_{\mathbf{x}}$ (zero-point) are quantization parameters. In Eq. 6.43, u (upper) and l (lower) define the quantization grid limits:

$$s_{\mathbf{x}} = \frac{u - l}{2^b - 1}, zp_{\mathbf{x}} = \mathrm{clip}(\lfloor -\frac{l}{s} \rceil, 0, 2^b - 1). \tag{6.43}$$

The dequantization process can be formulated as follows:

$$\tilde{\mathbf{x}} = (\hat{\mathbf{x}} - zp_{\mathbf{x}}) \times s_{\mathbf{x}}. \tag{6.44}$$

6.4.3.3 Quantization Range Setting

The quantization range setting establishes the quantization grid's upper and lower clipping thresholds, denoted as u and l, respectively. The crucial trade-off in range setting lies in the balance between two types of errors: clipping error and rounding error. Clipping error arises when data is truncated to fit within the predefined grid limits, as described in Eq. 6.43. Such truncation leads to information loss and decreased precision in the resulting quantized representation. On the other hand, rounding error occurs due to the imprecision introduced during the rounding operation, as described in Eq. 6.42. This error can accumulate over time and impact the overall accuracy of the quantized representation. The following methods provide different trade-offs between the two quantities. **MinMax** In the experiments, we use the MinMax method for weight quantization, where clipping thresholds $l_{\mathbf{x}}$ and $u_{\mathbf{x}}$ are formulated as:

$$l_{\mathbf{x}} = \min(\mathbf{x}), u_{\mathbf{x}} = \max(\mathbf{x}). \tag{6.45}$$

This leads to no clipping error. However, this approach is sensitive to outliers, as strong outliers may cause excessive rounding errors. **Mean Squared Error (MSE)** One way to mitigate the problem of large outliers is by employing an MSE-based range setting. In this method, we determine $l_{\mathbf{x}}$ and $u_{\mathbf{x}}$ that minimize the mean squared error (MSE) between the original and quantized tensor:

$$\underset{l_{\mathbf{x}}, u_{\mathbf{x}}}{\arg\min}\ \mathrm{MSE}(\mathbf{x}, \mathbf{Q}_{l_{\mathbf{x}}, u_{\mathbf{x}}}), \tag{6.46}$$

where $\mathbf{x}$ represents the original tensor and $\mathbf{Q}_{l_\mathbf{x},u_\mathbf{x}}$ denotes the quantized tensor produced using the determined clipping thresholds $l_\mathbf{x}$ and $u_\mathbf{x}$. The optimization problem is commonly solved using grid search, golden section method, or analytical approximations with closed-form solutions.

6.4.3.4 Unilateral Histogram (UH)-Based Activation Quantization

To address the issue of activation value imbalance, we propose a new approach called unilateral histogram (UH)-based activation quantization. We empirically study the activation values after forward propagation through the calibration dataset. We observe a concentrated distribution of values near the lower bound, accompanied by a noticeable decrease in occurrences above zero. Further analysis of the activation values reveals that the empirical value of −0.2785 is the lower bound. This phenomenon can be attributed to the frequent utilization of the Swish (SILU) activation function in the YOLO series.

Algorithm 14: Unilateral histogram (UH)-based activation quantization

```
1: **Input:** FP32 Histogram $H$ with 2048 bins
2: **for** $i$ **in** range(128, 2048) **do**
3:     Reference distribution $P \leftarrow H[0:i]$
4:     Outliers count $c \leftarrow \sum_{j=i}^{2047} H[j]$
5:     $P[i-1] \leftarrow P[i-1] + c$
6:     $P \leftarrow \frac{P}{\sum_j (P[j])}$
7:     Candidate distribution $C \leftarrow$ Quantize $H[0:i]$ into 128 levels
8:     Expand $C$ to have $i$ bins
9:     $Q \leftarrow \frac{C}{\sum_j (C[j])}$
10:    $MSE[i] \leftarrow$ Mean Squared Error($P$, $Q$)
11: **end for**
12: **Output:** Index $m$ for which $MSE[m]$ is minimal.
```

Based on the empirical evidence, we introduce an asymmetric quantization approach called unilateral histogram (UH)-based activation quantization. In UH, we iteratively determine the maximum truncation value that minimizes the quantization error while keeping the minimum truncation value fixed at −0.2785, as illustrated in the following:

$$u_\mathbf{x} = \underset{l_\mathbf{x},u_\mathbf{x}}{\arg\min}\ \mathrm{MSE}(\mathbf{x}, \mathbf{Q}_{l_\mathbf{x},u_\mathbf{x}}), l_\mathbf{x} = -0.2785. \tag{6.47}$$

To evaluate the quantization error during the search for the maximum truncation value, we utilize the fp32 floating-point numbers derived from the center values of the gathered 2048 bins, as introduced in Algorithm 14. These numbers are successively quantized, considering the current maximum truncation value. Through this iterative process, we identify the optimal truncation range. The UH activation quantization method offers two key advantages. Firstly, it significantly reduces calibration time. Secondly, it ensures stable activation quantization by allowing a more extensive set of integers to represent the frequently occurring activation values between 0 and -0.2785, thereby improving quantization accuracy.

6.4.3.5 Experiments

To assess the performance of the proposed Q-YOLO [59] detectors, we conducted a comprehensive series of experiments on the widely recognized COCO `2017` [32] detection benchmark. As one of the most popular object detection datasets, COCO `2017` [32] has become instrumental in benchmarking state-of-the-art object detectors, thanks to its rich annotations and challenging scenarios. Throughout our experimental analysis, we employed standard COCO metrics on the bounding box detection task to evaluate the efficacy of our approach.

Implementation Details
We randomly selected 1500 training images from the COCO `train2017` dataset [32] as the calibration data, which served as the foundation for optimizing the model parameters. The performance evaluation occurred on the COCO `val2017` dataset [32], comprising 5000 images. The image size is set to 640×640.

Unless otherwise noted, our experiments employed symmetric channel-wise quantization for weights and asymmetric layer-wise quantization for activations. We consistently applied the MinMax approach for quantizing weights to ensure a fair and unbiased comparison. The input and output layers of the model are more sensitive to the loss of accuracy. To maintain the model's overall performance, the original accuracy of these layers is usually retained. We also follow this practice.

Main Results
We apply our proposed Q-YOLO to quantize YOLOv5s [55], YOLOv5m [55], YOLOv7 [58], and YOLOv7x [58], which have an increasing number of parameters. The results of the full-precision model and the 8-bit and 4-bit quantized models using MinMax, percentile, and Q-YOLO methods are all presented in Table 6.14.

Table 6.14 compares several quantization approaches and detection methods in computing complexity and storage cost. Our Q-YOLO significantly accelerates computation and reduces storage requirements for various YOLO detectors. Similarly, in terms of detection accuracy, when using Q-YOLO to quantize the YOLOv5 series models to 8 bits, there is virtually no decline in the average precision (AP) value compared to the full-precision model. As the number of model parameters increases dramatically, quantizing the YOLOv7 series models to 8 bits results in a slight decrease in accuracy. When quantizing models to 4 bits, the accuracy experiences a significant loss due to the reduced expressiveness of 4-bit integer representation. Particularly, when using the MinMax quantization method, the model loses all its accuracy, whereas the percentile method, which roughly truncates 99.99% of the extreme values, fails to bring notable improvement. Differently, Q-

Table 6.14 A comparison of various quantization methods applied to YOLOv5s [55], YOLOv5m [55], YOLOv7 [58], and YOLOv7x [58], which have an increasing number of parameters, on the COCO `val2017` dataset [32]. The term bits (W-A) represents the bit width of weights and activations. The best results are displayed in bold

Models	Method	Bits	$\text{Size}_{(\text{MB})}$	$\text{OPs}_{(\text{G})}$	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
YOLOv5s [55]	Real-valued	32–32	57.6	16.5	37.4	57.1	40.1	21.6	42.3	48.9
	MinMax	8–8	14.4	4.23	37.2	56.9	39.8	21.4	42.2	48.5
	Percentile [28]				36.9	56.4	39.6	21.3	42.4	48.1
	Q-YOLO				**37.4**	**56.9**	**39.8**	**21.4**	**42.4**	**48.8**
	Percentile [28]	4–4	7.7	2.16	7.0	14.2	6.3	4.1	10.7	7.9
	Q-YOLO				**14.0**	**26.2**	**13.5**	**7.9**	**17.6**	**19.0**
YOLOv5m [55]	Real-valued	32–32	169.6	49.0	45.1	64.1	49	28.1	50.6	57.8
	MinMax	8–8	42.4	12.4	44.9	64	48.9	27.8	50.5	57.4
	Percentile [28]				44.6	63.5	48.4	28.4	50.4	57.8
	Q-YOLO				**45.1**	**64.1**	**48.9**	**28**	**50.6**	**57.7**
	Percentile [28]	4–4	21.2	6.33	19.4	35.6	19.1	14.6	28.3	17.2
	Q-YOLO				**28.8**	**46**	**30.5**	**15.4**	**33.8**	**38.7**
YOLOv7 [58]	Real-valued	32–32	295.2	104.7	50.8	69.6	54.9	34.9	55.6	66.3
	MinMax	8–8	73.8	27.2	50.6	69.5	54.8	34.1	55.5	65.9
	Percentile [28]				50.5	69.3	54.6	34.5	55.4	66.2
	Q-YOLO				**50.7**	**69.5**	**54.8**	**34.8**	**55.5**	**66.2**
	Percentile [28]	4–4	36.9	14.1	16.7	26.9	17.8	10.3	20.1	20.2
	Q-YOLO				**37.3**	**55.0**	**40.9**	**21.5**	**41.4**	**53.0**
YOLOv7x [58]	Real-valued	32–32	25.5	189.9	52.5	71.0	56.6	36.6	57.3	68.0
	MinMax	8–8	142.6	49.5	52.3	70.9	56.7	36.6	57.1	67.7
	Percentile [28]				52.0	70.5	56.1	36.0	56.8	67.9
	Q-YOLO				**52.4**	**70.9**	**56.5**	**36.2**	**57.2**	**67.8**
	Percentile [28]	4–4	71.3	25.6	36.8	55.3	40.5	21.2	41.7	49.3
	Q-YOLO				**37.6**	**57.8**	**42.1**	**23.7**	**43.8**	**49.1**

YOLO successfully identifies a more appropriate scale for quantization, resulting in a considerable enhancement compared to conventional post-training quantization (PTQ) methods.

6.4.3.6 Ablation Study

Symmetry in Activation Quantization
Nowadays, quantization schemes are often subject to hardware limitations; for instance, NVIDIA [43] only supports symmetric quantization, as it is more inference-speed friendly. Therefore, discussing the symmetry in activation value quantization is meaningful. Table 6.15 compares results using Q-YOLO for symmetric and asymmetric quantization, with the latter exhibiting higher accuracy. The range of negative activation values lies between 0 and -0.2785, while the range of positive activation values exceeds that of the negative ones. The accuracy will naturally decrease if we force equal integer expression bit numbers on both positive and negative sides. Moreover, this decline becomes more pronounced as the quantization bit number decreases.

6.4.3.7 Quantization Type

In Table 6.16, we analyze the impact of different quantization types on the performance of the YOLOv5s and YOLOv5m models, considering three cases: quantizing only the weights (*only weights*), quantizing only the activation values (*only activation*), and quantizing both weights and activation values (*weights+activation*). The results demonstrate that, compared to quantizing the activation values, quantizing the weights consistently induces more considerable performance degradation. Additionally, the lower the number of bits, the greater the loss incurred by quantization.

Table 6.15 A comparison of symmetrical analysis of activation value quantization. *Asymmetric* indicates the use of an asymmetric activation value quantization scheme, while *symmetric* refers to the symmetric quantization of activation values

models	Bits	Symmetry	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
YOLOv5s [55]	Real-valued	–	37.4	57.1	40.1	21.6	42.3	48.9
	6–6	*Asymmetric*	35.9	55.7	38.3	20.4	41.0	47.6
		Symmetric	34.4	53.9	37.0	19.3	39.8	45.0
	4–4	*Asymmetric*	14.0	26.2	13.5	7.9	17.6	19.0
		Symmetric	2.7	5.9	2.2	1.3	4.2	4.6
YOLOv5m [55]	Real-valued	–	45.1	64.1	49.0	28.1	50.6	57.8
	6–6	*Asymmetric*	44.0	63.1	47.7	28	49.9	56.8
		Symmetric	42.4	61.1	46.0	25.3	48.3	55.9
	4–4	*Asymmetric*	28.8	46.0	30.5	15.4	33.8	38.7
		Symmetric	11.3	24.8	8.6	7.5	15.2	14.5

Table 6.16 A comparison of quantization type. The term *only weights* signifies that only the weights are quantized, *only activation* indicates that only the activation values are quantized, and *weights+activation* represents the quantization of both activation values and weights

models	Bits	Quantization type	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
YOLOv5s [55]	Real-valued	–	37.4	57.1	40.1	21.6	42.3	48.9
	6–32	*Only weights*	36.7(−0.7)	56.6	39.3	20.9	41.4	48.4
	32–6	*Only activation*	36.6(−0.8)	56.2	39.3	21.0	42.0	47.9
	6–6	*Weights+activation*	35.9	55.7	38.3	20.4	41.0	47.6
	4–32	*Only weights*	19.6(−16.3)	35.6	19.3	11.3	22.5	25.7
	32–4	*Only activation*	30.6(−5.3)	49.1	32.6	17.0	36.7	40.7
	4–4	*Weights+activation*	14.0	26.2	13.5	7.9	17.6	19
YOLOv5m [55]	Real-valued	–	45.1	64.1	49.0	28.1	50.6	57.8
	6–32	*Only weights*	44.7(−0.4)	63.9	48.6	28.0	50.3	57.3
	32–6	*Only activation*	44.3(−0.8)	63.4	48.1	28.4	50.3	57.2
	6–6	*Weights+activation*	44	63.1	47.7	28.0	49.9	56.8
	4–32	*Only weights*	34.6(−9.4)	54.0	37.3	20.0	39.2	45.3
	32–4	*Only activation*	37.7(−6.3)	57.3	41.8	23.7	44.1	51.0
	4–4	*Weights+activation*	28.8	46.0	30.5	15.4	33.8	38.7

In YOLO, the weights learned by a neural network essentially represent the knowledge acquired by the network, making the precision of the weights crucial for model performance. In contrast, activation values serve as intermediate representations of input data propagating through the network and can tolerate some degree of quantization error to a certain extent.

6.4.3.8 Inference Speed

To practically verify the acceleration benefits brought about by our quantization scheme, we conducted inference speed tests on both GPU and CPU platforms. For the GPU, we selected the commonly used desktop GPU `NVIDIA RTX 4090` [43] and the `NVIDIA Tesla T4` [43], often used in computing centers for inference tasks. Due to our limited CPU resources, we only tested Intel products, the `i7-12700H` and `i9-10900`, both of which have $\times 86$ architecture. We chose TensorRT [1] and OpenVINO [2] for deployment tools. The entire process involved converting the weights from the torch framework into an ONNX model with QDQ nodes and deploying them onto specific inference frameworks. The inference mode was set to single-image serial inference, with an image size of 640×640. As most current inference frameworks only support symmetric quantization and 8-bit quantization, we had to choose a symmetric 8-bit quantization scheme, which resulted in a minimal decrease in accuracy compared to asymmetric schemes. As shown in Table 6.17, the acceleration is extremely significant, especially for the larger YOLOv7 model, wherein the speedup ratio when using a GPU even exceeded

Table 6.17 The inference speed of the quantized model is essential. The quantization scheme adopts uniform quantization, with single-image inference mode and an image size of 640 × 640. TensorRT [1]is selected as the GPU inference library, while OpenVINO [2] is chosen for the CPU inference library

Models	Bits	AP	GPU speed/ms		Intel CPU speed/ms	
			RTX 4090	Tesla T4	i7-12700H(×86)	i9-10900(×86)
YOLOv5s	32–32	37.4	4.9	7.1	48.7	38.7
	8–8	37.3	3.0	4.5	33.6	23.4
YOLOv7	32–32	50.8	16.8	22.4	269.8	307.8
	8–8	50.6	5.4	7.8	120.4	145.2

3× compared to the full-precision model. This demonstrates that quantization in real-time detectors can bring about a remarkable acceleration.

References

1. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. Accessed: 2022-09-03.
2. OpenVINO Toolkit. https://docs.openvinotoolkit.org/latest/index.html. Accessed: 2022-09-03.
3. Adrian Bulat and Georgios Tzimiropoulos. Xnor-net++: Improved binary neural networks. *arXiv preprint arXiv:1909.13863*, 2019.
4. Zhaowei Cai and Nuno Vasconcelos. Cascade r-cnn: Delving into high quality object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6154–6162, 2018.
5. Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.
6. Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.
7. Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.
8. Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.
9. Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 2010.
10. Pedro Felzenszwalb and Ramin Zabih. Discrete optimization algorithms in computer vision. *Tutorial at CVPR*, 2007.
11. Sicheng Gao, Runqi Wang, Liuyang Jiang, and Baochang Zhang. 1-bit wavenet: compressing a generative neural network in speech recognition with two binarized methods. In *2021 IEEE 16th conference on industrial electronics and applications (ICIEA)*, pages 2043–2047. IEEE, 2021.
12. Ross Girshick. Fast r-cnn. In *ICCV*, 2015.
13. Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

14. Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.
15. Jiaxin Gu, Ce Li, Baochang Zhang, Jungong Han, Xianbin Cao, Jianzhuang Liu, and David Doermann. Projection convolutional neural networks for 1-bit cnns via discrete back propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.
16. Jiaxin Gu, Ce Li, Baochang Zhang, Jungong Han, Xianbin Cao, Jianzhuang Liu, and David Doermann. Projection convolutional neural networks for 1-bit cnns via discrete back propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8344–8351, 2019.
17. Jiaxin Gu, Junhe Zhao, Xiaolong Jiang, Baochang Zhang, Jianzhuang Liu, Guodong Guo, and Rongrong Ji. Bayesian optimized 1-bit cnns. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4909–4917, 2019.
18. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
19. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
20. Felix Heide, Wolfgang Heidrich, and Gordon Wetzstein. Fast and flexible convolutional sparse coding. In *CVPR*, pages 5135–5143, 2015.
21. Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018.
22. Lianghua Huang, Xin Zhao, and Kaiqi Huang. Got-10k: A large high-diversity benchmark for generic object tracking in the wild. *IEEE transactions on pattern analysis and machine intelligence*, 43(5):1562–1577, 2019.
23. Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.
24. Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.
25. Hyungjun Kim, Kyungsu Kim, Jinseok Kim, and Jae-Joon Kim. Binaryduo: Reducing gradient mismatch in binary activation network by coupling binary activations. In *International Conference on Learning Representations*, 2019.
26. Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
27. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.
28. Rundong Li, Yan Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. Fully quantized network for object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2810–2819, 2019.
29. Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.
30. Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.
31. Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision (ECCV)*, pages 740–755, 2014.

32. Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *ECCV*, 2014.
33. Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.
34. Chunlei Liu, Wenrui Ding, Yuan Hu, Baochang Zhang, Jianzhuang Liu, Guodong Guo, and David Doermann. Rectified binary convolutional networks with generative adversarial learning. *International Journal of Computer Vision*, 129:998–1012, 2021.
35. Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Proc. of ECCV*, 2016.
36. Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. *arXiv preprint arXiv:2003.03488*, 2020.
37. Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018.
38. Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. In *International Conference on Learning Representations*, 2019.
39. Mark D McDonnell. Training wide residual networks for deployment using a single bit for each weight. *arXiv preprint arXiv:1802.08530*, 2018.
40. Matthias Mueller, Neil Smith, and Bernard Ghanem. A benchmark and simulator for uav tracking. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 445–461. Springer, 2016.
41. Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
42. Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.
43. NVIDIA. Nvidia corporation, 2022. Available at: https://www.nvidia.com/.
44. Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.
45. Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS Workshops*, 2017.
46. Rohit Prabhavalkar, Ouais Alsharif, Antoine Bruguier, and Lan McGraw. On the compression of recurrent neural networks with an application to lvcsr acoustic modeling for embedded speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5970–5974. IEEE, 2016.
47. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.
48. Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.
49. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NeurIPS*, 2015.
50. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016.

51. Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 658–666, 2019.
52. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.
53. Kunal Pratap Singh, Dahyun Kim, and Jonghyun Choi. Learning architectures for binary networks. *arXiv preprint arXiv:2002.06963*, 2020.
54. Siyang Sun, Yingjie Yin, Xingang Wang, De Xu, Wenqi Wu, and Qingyi Gu. Fast object detection based on binary deep convolution neural networks. *CAAI transactions on intelligence technology*, 3(4):191–197, 2018.
55. Ultralytics. YOLOv5: PyTorch implementation of YOLOv5 real-time object detection. https://github.com/ultralytics/yolov5, 2021.
56. Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. *Advances in neural information processing systems*, 29, 2016.
57. Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. Tbn: Convolutional neural network with ternary inputs and binary weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 315–332, 2018.
58. Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7464–7475, 2023.
59. Ming Wang, Hui Xian Sun, Jun Shi, Xuhui Liu, Baochang Zhang, and Xianbin Cao. Q-yolo: Efficient inference for real-time object detection. *ArXiv*, abs/2307.04816, 2023.
60. Qilong Wang, Banggu Wu, Pengfei Zhu, Peihua Li, Wangmeng Zuo, and Qinghua Hu. Eca-net: Efficient channel attention for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11534–11542, 2020.
61. Xiaodi Wang, Baochang Zhang, Ce Li, Rongrong Ji, Jungong Han, Xianbin Cao, and Jianzhuang Liu. Modulated convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 840–848, 2018.
62. Ziwei Wang, Jiwen Lu, Chenxin Tao, Jie Zhou, and Qi Tian. Learning channel-wise interactions for binary convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 568–577, 2019.
63. Ziwei Wang, Ziyi Wu, Jiwen Lu, and Jie Zhou. Bidet: An efficient binarized object detector. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2049–2058, 2020.
64. Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.
65. Yi Wu, Jongwoo Lim, and Ming-Hsuan Yang. Online object tracking: A benchmark. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2411–2418, 2013.
66. Yi Wu, Jongwoo Lim, and Ming-Hsuan Yang. Object tracking benchmark. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 37(09):1834–1848, 2015.
67. Xu Xiang, Yanmin Qian, and Kai Yu. Binary deep neural networks for speech recognition. In *INTERSPEECH*, pages 533–537, 2017.
68. Sheng Xu, Zhendong Liu, Xuan Gong, Chunlei Liu, Mingyuan Mao, and Baochang Zhang. Amplitude suppression and direction activation in networks for 1-bit faster r-cnn. *Proceedings of the 4th International Workshop on Embedded and Mobile Deep Learning*, 2020.
69. Li Yang, Zhezhi He, and Deliang Fan. Binarized depthwise separable neural network for object tracking in fpga. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 347–350, 2019.

70. Linlin Yang, Ce Li, Jungong Han, Chen Chen, Qixiang Ye, Baochang Zhang, Xianbin Cao, and Wanquan Liu. Image reconstruction via manifold constrained convolutional sparse coding for image sets. *JSTSP*, 11(7):1072–1081, 2017.
71. Shouyi Yin, Peng Ouyang, Shixuan Zheng, Dandan Song, Xiudong Li, Leibo Liu, and Shaojun Wei. A 141 uw, 2.46 pj/neuron binarized convolutional neural network based self-learning speech recognition processor in 28nm cmos. In *2018 IEEE Symposium on VLSI Circuits*, pages 139–140. IEEE, 2018.
72. Jiahui Yu, Yuning Jiang, Zhangyang Wang, Zhimin Cao, and Thomas Huang. Unitbox: An advanced object detection network. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 516–520, 2016.
73. Junhe Zhao, Sheng Xu, Runqi Wang, Baochang Zhang, Guodong Guo, David S. Doermann, and Dianmin Sun. Data-adaptive binary neural networks for efficient object detection and recognition. *Pattern Recognit. Lett.*, 153:239–245, 2021.
74. Shixuan Zheng, Peng Ouyang, Dandan Song, Xiudong Li, Leibo Liu, Shaojun Wei, and Shouyi Yin. An ultra-low power binarized convolutional neural network-based speech recognition processor with on-chip self-learning. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(12):4648–4661, 2019.
75. Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss: Faster and better learning for bounding box regression. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 12993–13000, 2020.
76. Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.
77. Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.