Title
Hardware-Aware Efficient Deep Learning

Permalink
https://escholarship.org/uc/item/6w13g869

Author
Dong, Zhen

Publication Date
2022

Peer reviewed|Thesis/dissertation

Hardware-Aware Efficient Deep Learning

by

Zhen Dong

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt Keutzer, Chair
Professor Trevor Darrell
Professor Joseph Gonzalez
Doctor Bichen Wu

Fall 2022

Hardware-Aware Efficient Deep Learning

Abstract

Hardware-Aware Efficient Deep Learning

by

Zhen Dong

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt Keutzer, Chair

Significant improvements in the accuracy of Neural Networks (NNs) have been observed for a wide range of problems, often achieved by highly over-parameterized models. Despite the accuracy of these state-of-the-art models, their sheer size makes it impossible to deploy them for many resource-constrained applications, such as real-time intelligent healthcare monitoring, autonomous driving, audio analysis, and speech recognition. This creates a problem in realizing pervasive deep learning, which requires real-time inference with low energy consumption, high accuracy, and limited computational resources.

Achieving efficient NNs that can achieve real-time constraints with optimal accuracy requires the co-optimization of 1) NN architecture design, 2) model compression methods, and 3) the design of hardware engines. Previous work pursuing efficient deep learning focused more on optimizing proxy metrics such as memory size and the FLOPs, while the hardware specifications actually play an important role in determining the overall performance. Furthermore, due to the extremely large design space, the aforementioned three aspects are often optimized separately and empirically in previous literature, making the whole design process time-consuming and sub-optimal.

In this dissertation, we first systematically studied the quantization method, which is a widely used and standard model compression technique. Instead of using a heuristic design or costly searching, we tackled the mixed-precision quantization problem by leveraging the Hessian information, and our proposed Hessian-AWare Quantization (HAWQ) method achieved state-of-the-art performance on different networks and datasets. We further made the whole pipeline fully automatic (HAWQV2) and explored different aspects of quantization (ZeroQ) on different tasks (QBERT).

Based on our systematic quantization method, we then included hardware specifications and deployment into the design space (HAWQV3). The neural architecture was taken into the co-design (CoDeNet) and was searched automatically as well (HAO). Finally, we increased the efficiency of the whole automatic HW-SW co-design pipeline by introducing teacher-based block-wise distillation (ETA). Overall, our work in this dissertation demonstrates steps in the evolution from traditional NN design toward hardware-aware efficient deep learning. We believe this will further accelerate the deployment of advanced NNs on resource-limited devices and in real-world applications.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**
1.1 Hardware-Aware Efficient Deep Learning 1
1.2 Organization of the Thesis 2

2 Metrics of Model Efficiency **4**
2.1 Practical efficiency metrics 4
2.2 Theoretical efficiency metrics 4

3 Motivations to Hardware-Aware Efficient Deep Learning **6**
3.1 Quantization 6
3.2 HW-SW Co-Design 8

4 Introduction and Related Work of Quantization **11**
4.1 Linear & Non-linear Quantizers 11
4.2 QAT & PTQ 14
4.3 Quantization Granularity 17
4.4 Uniform & Mixed-Precision Quantization 18

5 Quantization: HAWQ **21**
5.1 Method 21
5.2 Experiments 24
5.3 Ablation Study 28

6 Quantization: HAWQV2 **30**
6.1 Method 30
6.2 Experiments 36

7 Quantization: Q-BERT 41
7.1 Introduction to NLP tasks and Compression 41
7.2 Method 42
7.3 Experiments 46

8 Quantization: ZeroQ 53
8.1 Method 53
8.2 Experiments 58

9 Conclusion on Quantization 62

10 Introduction and Related Work of HW-SW Co-Design 63
10.1 Hardware-Aware Neural Architecture Design 63
10.2 Hardware-Aware Model Compression 64
10.3 Hardware-Software Co-Optimization 65

11 HW-SW Co-Design: HAWQV3 66
11.1 Method 66
11.2 Experiments 72

12 HW-SW Co-Design: CoDeNet 77
12.1 Introduction 77
12.2 Method 79
12.3 Experiments 88

13 HW-SW Co-Design: HAO 92
13.1 Method 92
13.2 Experiments 101

14 HW-SW Co-Design: ETA 106
14.1 Method 106
14.2 Experiments 111
14.3 Ablation Study 113

15 Conclusion on HW-SW Co-Design 118

16 Conclusions 119
16.1 Impact of our work 119
16.2 Future work 120

Bibliography 122

List of Figures

1.1 Three components of hardware-aware efficient deep learning. 2

3.1 Comparison of peak operations, energy cost, and relative area cost for different bit-precision. 7
3.2 Throughput and power of different commercial edge processors for NN inference. 9

4.1 Comparison between uniform quantization and non-uniform quantization. 12
4.2 Illustration of symmetric quantization and asymmetric quantization. 13
4.3 Comparison between Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). 15
4.4 Illustration of different quantization granularities. 17
4.5 Illustration of mixed-precision quantization. 19

5.1 Top eigenvalue of each individual block of pre-trained ResNet20 on Cifar-10 and Inception-V3 on ImageNet. 23
5.2 Accuracy recovery from Hessian aware mixed-precision quantization versus HAWQ-Reverse-Precision quantization. 28
5.3 Effectiveness of Hessian aware block-wise fine-tuning. 29

6.1 Average Hessian trace of different blocks in InceptionV3 and ResNet50 on ImageNet, along with the loss landscape. 31
6.2 Illustration of the structure of Hessian w.r.t to activations. 34
6.3 Average Hessian trace of different blocks in SqueezeNext and RetinaNet, along with the loss landscapes. 34
6.4 Pareto Frontier: The trade-off between model size and the sum of Ω metric. . . 35
6.5 Relationship between the convergence of Hutchinson and the number of data points as well as the number of steps. 36
6.6 Average Hessian trace w.r.t. activations in RetinaNet and the relationship between the convergence of Hutchinson and the number of data points. 37

7.1 From (a) to (d): Top eigenvalue distributions for different encoder layers for SST-2, MNLI, CoNNL-03, SQuAD, respectively. 43
7.2 The loss landscape for different layers in MNLI and CoNNL-03. 43
7.3 The loss landscape for different layers of BERT on SQuAD. 45

7.4 The overview of Group-wise Quantization Method. 46
7.5 KL divergence over attention distribution between Q-BERT/DirectQ and Baseline. 52

8.1 Visualization of Gaussian data and Distilled Data. 54
8.2 Sensitivity of each layer in ResNet50 when quantized to different weight precision, measured with different kinds of data. 55
8.3 The Pareto frontier of ZeroQ using ResNet50 on ImageNet. 56

11.1 Illustration of fake vs true quantization for convolution and batch normalization. 67
11.2 Illustration of HAWQV3 for a residual block with and without transition layer. . 69
11.3 Illustration of the final model specification that the ILP solver finds for ResNet18 with latency constraint. 76

12.1 Example for the input-adaptive deformable convolution sampling locations and offset range distribution. 78
12.2 Deformable convolution with input-adaptive offsets generation. 78
12.3 Major algorithm modifications for deformable convolution operational co-design. 81
12.4 Hardware engine for deformable convolution. 82
12.5 The architecture diagrams of our building blocks and model architecture. 84
12.6 The output heads of CenterNet for object detection. 84
12.7 Architectural diagram of the FPGA accelerator. 86
12.8 Latency-accuracy trade-off of CoDeNet on VOC. 91

13.1 Hardware design space of HAO. 93
13.2 LUT usage of multipliers with different input precisions. 95
13.3 Example mapping of two low-precision MACs onto a DSP. 95
13.4 Illustration of HAO pipeline. 98
13.5 The performance of the latency predictor and the accuracy predictor 101
13.6 Pareto frontier of HAO for accuracy and latency. 103
13.7 Illustration of neural architecture and quantization setting searched by HAO. . . 103

14.1 Quantization-aware ETA results. 113
14.2 Comparison between accuracy predictor and the additive objective used in ILP [174] on 5% of training data. 116

List of Tables

5.1 Quantization results of ResNet20 on Cifar-10. 25
5.2 HAWQ quantization results of Inception-V3 on ImageNet. 26
5.3 HAWQ quantization results of ResNet50 on ImageNet. 27
5.4 HAWQ Quantization results of SqueezeNext on ImageNet. 27

6.1 HAWQV2 quantization results on ImageNet. 38
6.2 HAWQV2 quantization results of RetinaNet-ResNet50 on Microsoft COCO 2017. 39
6.3 The effectiveness of metrics proposed in HAWQV2. 40
6.4 The effectiveness of average Hessian trace. The experiments are for InceptionV3 on ImageNet. 40

7.1 Quantization results for BERTBASE on Natural Language Understanding tasks. . 47
7.2 Quantization results for BERTBASE on SQuAD. 48
7.3 Effects of group-wise quantization for Q-BERT on three tasks. 49
7.4 Quantization effect to different modules. 50

8.1 ZeroQ quantization results of ResNet50, MobileNetV2, and ShuffleNet on ImageNet. 58
8.2 Uniform post-quantization results on ImageNet with ResNet18. 60
8.3 Object detection with ZeroQ on Microsoft COCO using RetinaNet. 61

11.1 HAWQV3 quantization results for ResNet18/50 and InceptionV3. 73
11.2 HAWQV3 mixed-precision quantization results for ResNet18 and ResNet50 with different constraints. 74

12.1 Ablation study of operation choices for object detection on VOC and COCO. . . 80
12.2 Co-designed hardware performance comparison. 82
12.3 Quantized CoDeNet on VOC object detection. 88
12.4 Quantized CoDeNet on COCO object detection. 88
12.5 CoDeNet performance compared with prior works. 89
12.6 FPGA resource utilization. 90

13.1 Notations for hardware design. 94
13.2 Performance comparison on ImageNet with prior works. 104
13.3 Hardware resources utilization and power 104

14.1 ETA results on ImageNet. 110
14.2 Transformer-based ETA results on ImageNet-1K. 112
14.3 Ablation study on different optimization methods. 114
14.4 Effect of different optimizers and the amount of data on the performance of the accuracy predictor. 114
14.5 Ablation Study on Elastic Resolution. 115
14.6 Speedup achieved by applying elastic width during layer-wise pretraining. 117

Acknowledgments

I'm lucky to have received so much help and support along the way of my PhD journey. First, I want to sincerely thank my advisor, Professor Kurt Keutzer, for all his advice, guidance, and support. In addition to specific skills, Kurt has taught me the methodology of conducting research and also his thinking about life. Whenever I faced issues, Kurt would always be there to help me, from addressing the cultural difference to advising my career path. I feel grateful and fortunate to work with Kurt for the past four years.

I would like to express my gratitude to my qual and dissertation committee, Professor Joseph Gonzalez, Professor Trevor Darrel, and Doctor Bichen Wu, who have advised me on my research directions as well as career development and choices in the future. For many of my previous works, I received valuable guidance and help from Professor Michael Mahoney and Professor David Patterson, which I greatly appreciate. I would also like to thank Professor James Demmel, Professor Aydin Buluc, Professor Katherine Yelick, Professor Ilan Adler, and Professor Borivoje Nikolic for their valuable suggestions and instructions.

I have worked with and learned from my senior mentors and collaborators. For the quantization part of my works, I received great help and guidance from Amir Gholami and Zhewei Yao. For the HW-SW co-design part of my works, I want to thank Qijing (Jenny) Huang and Dequan Wang. And I want to acknowledge the help received from Shanghang Zhang on another line of our works on cross-domain text classification [132, 133, 134], which are not presented in this thesis.

Moreover, I'm thankful to my mentors and collaborators from the industry. I feel honored to work with Hongxu Yin, Pavlo Molchanov, Arash Vahdat from NVIDIA, Yida Wang, Leyuan Wang from Amazon, Xiaoyong Liu from Alibaba, Peter Vajda, Peizhao Zhang from Meta, Ellick Chan and Kittur Ganesh from Intel, Jiayu Ye from Google, Jiashi Feng from ByteDance, Forrest Iandola from Tesla and Kees Vissers from AMD. I would always appreciate the opportunities I had to collaborate with so many people from the industry.

My research has been dependent on my fellow collaborators who shared their creative minds and technical skills with me. I want to thank Sheng Shen, Sehoon Kim, Huanrui Yang, Linjian Ma, Weijiang Yu, Guohao Li, Woosuk Kwon and Daquan Zhou.

I would always cherish the opportunity to mentor or interact with so many extraordinary minds from institutions around the world. I want to thank Yaohui Cai, Daiyaan Arfeen, Aniruddha Nrusimha, Zhangcheng Zheng, Eric Tan, Tianmu Lei, Hanbing Zhan, Lu Yu, Yizhao Gao, Shixing Yu, Tian Li, Xiang Chen, Sophia Yan, Kaicheng Zhou, Qiang Zhou, Mingfei Guo, Lingran Zhao, Lutfi Eren Erdogan, Yang Zhou, Brian Yu, and others.

Finally, I want to thank my parents who always dedicatedly and faithfully support me and mentor me in every aspect of life. Their unconditional love and encouragement have armed me with the strength to pursue my academic career and beyond.

Chapter 1

Introduction

1.1 Hardware-Aware Efficient Deep Learning

As the parameter size and computation of state-of-the-art deep learning models grow dramatically, the efficient deployment of these models on different hardware platforms has become increasingly crucial. Given specific hardware resources and constraints, 1) model compression, 2) neural architecture design/search, and 3) hardware optimization are the mainstream methods to obtain feasible solutions. Including quantization, pruning, knowledge distillation, and factorization methods, model compression aims to compress a pretrained model concerning both model size and computation. With the current hardware support for low-precision computations, quantization has become a popular procedure to address these challenges. From a different perspective, NAS algorithms try to search for an efficient neural architecture and then train it from scratch. In contrast, hardware optimization is always performed after the neural architecture and model compression methods are fixed. Despite the merits, in order to fully utilize the system and achieve hardware-aware efficient deep learning, there are actually three issues in previous works which we tried to address in this thesis.

First, we want to note that the three components to achieve efficient deep learning are not orthogonal to each other. As shown in Figure 1.1, the performance of a specific neural architecture is actually highly relevant to the model compression method and the hardware specifications. For example, a ResNet50 with 4-bit quantization can run much faster than its counterpart with 8-bit quantization on an FPGA board with appropriate configurations. However, on a GPU that only supports 8-bit integer, the 8-bit quantized ResNet50 can have the same speed as the 4-bit ResNet50, while being able to achieve higher accuracy. In this thesis, we aim to achieve hardware-aware efficient deep learning, where we jointly consider the three aspects and try to obtain the sweet point in the trade-offs between them.

Secondly, previous works try to optimize proxy metrics such as the model size and the FLOPs of neural network models, assuming a high correlation of these theoretical metrics to practical efficient metrics such as latency, throughput, as well as energy consumption. However, it has been pointed out that proxy metrics can be misleading in specific cases. In

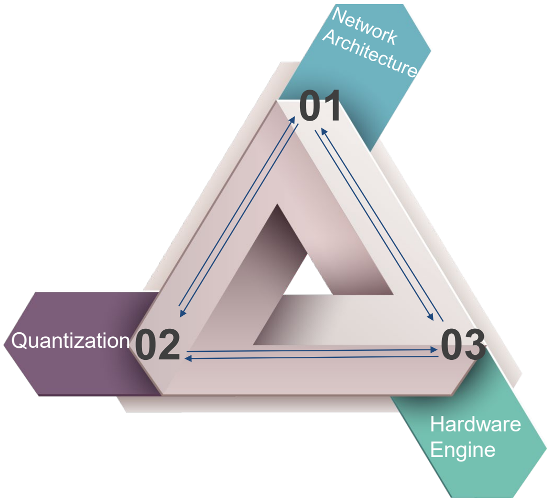


Figure 1.1: Three components of hardware-aware efficient deep learning.

order to avoid sub-optimal solutions, in this thesis, we want to directly optimize the practical metrics by leveraging the developed hardware engines or simulators.

Finally, jointly considering neural network architecture, model compression and hardware design can form an extremely large search space. Take a small part of the joint space as an example, mixed-precision quantization allows each layer of a neural network to select a specific quantization bitwidth, which leads to an exponentially large search space of the bitwidth configurations. Given the magnitude of the search space, previous methods are generally heuristic, which requires domain knowledge and manual efforts, or are time-consuming, which requires formidable computational resources to perform a searching process. In this thesis, we made our methods automatic and efficient by applying optimization methods such as Hessian analysis and integer optimization, as well as learning-based techniques such as latency and accuracy simulators, as well as block-wise knowledge distillation.

Our proposed methods in this thesis were able to achieve great performance while addressing the aforementioned issues. For example, with our mixed-precision quantization, we achieved a 10 × compression ratio on various models with only around 1% accuracy drop (please refer to Chapter 6 and Chapter 7). Furthermore, as an example of our use of HW-SW co-design, our 4-bit/8-bit mixed-precision model gets 67.1 AP50 on Pascal VOC with only 2.9 MB size, which is 21 × smaller but 10% more accurate than Tiny-YOLO (as shown in Chapter 12).

1.2 Organization of the Thesis

In this thesis, we first introduce the metrics we used to define hardware-aware efficient deep learning in Chapter 2. In Chapter 3, we show the motivations for applying hardware-aware efficient deep learning. Then we categorize our works into two lines, where the first line presents our progress in systematic quantization, and the second line describes our techniques to automatically and jointly explore the three design spaces in Figure 1.1. Specifically, Chapter 4 introduces concepts and related works of quantization, followed by Chapter 5 to Chapter 8 describing our works HAWQ, HAWQV2, Q-BERT, ZeroQ, and we make a conclusion in Chapter 9. In Chapter 10, we introduce the research direction of hardware-software co-design and the previous arts. We show our works HAWQV3, CoDeNet, HAO and ETA in Chapter 11 to Chapter 14, with Chapter 15 as a conclusion. Finally, we review the importance of our works and discuss the potential future directions in Chapter 16.

Chapter 2

Metrics of Model Efficiency

This dissertation is devoted to the topic of efficient deep learning. In this chapter, we begin by establishing the metrics by which we evaluate efficiency.

2.1 Practical efficiency metrics

Latency refers to the time taken to process one unit of data provided only one unit of data is processed at a time on the computing system. The unit of latency is seconds.

Throughput refers to the rate of successful neural network inference on the computing system. Traditionally, throughput is usually measured in bits per second (bit/s or bps). In deep learning, throughput often refers to frame per second (fps), where an input image can serve as one frame.

Power Consumption refers to the electrical energy per unit time, supplied to operate the neural network on the computing system. Power consumption is usually measured in units of watts (W) or kilowatts (kW).

2.2 Theoretical efficiency metrics

Model Size Assume that we have a model with L layers, $M_i^{(b_i)}$ denotes the size of the i-th layer with b_i bit quantization, then we have:

$$\text{Model Size} = \sum\nolimits_{i=1}^{L} M_i^{(b_i)}, \tag{2.1}$$

Note that $M_i^{(b_i)}$ depends on the parameter size P_i and the bitwidth b_i, where $M_i^{(b_i)} = P_i b_i/8$. The unit of model size is Byte, and 1 Byte equals 8 bits. The value of P_i depends on the type of the layer. For a convolutional layer:

$$P_i = C_{\text{in,i}} \times K_i^2 \times C_{\text{out,i}}, \tag{2.2}$$

where K_i is the kernel size of the i-th layer, $C_{\text{in,i}}$ and $C_{\text{out,i}}$ are number of the input and output channels of the i-th layer, respectively.

FLOPs & MAC FLOPs represents **FL**oating point **OP**eration**s**, which is a standard and handy proxy metric used to measure the computation of a specific neural network. For a model with L layers, F_i denotes the FLOPs of the i-th layer, then we have:

$$\text{FLOPs} = \sum\nolimits_{i=1}^{L} F_i, \tag{2.3}$$

Note that FLOPs or F_i are independent of the bitwidth b_i, since an operation with lower precision still counts as one operation when calculating FLOPs. The value of F_i depends on the type of the layer. For a convolutional layer:

$$F_i = (2C_{\text{in,i}} \times K_i^2 - 1) \times H_{out,i} W_{out,i} C_{\text{out,i}}, \tag{2.4}$$

where K_i is the kernel size of the i-th layer, $C_{\text{in,i}}$ and $C_{\text{out,i}}$ are number of the input and output channels of the i-th layer, respectively. $H_{\text{out,i}}$ and $W_{\text{out,i}}$ are the height and width of the output feature map of the i-th layer. Note that one multiply-accumulate operation counts as two floating point operations, and therefore the related term MAC_i (**M**ultiply-**AC**cumulate operation) is around half of the corresponding F_i.

BOPS BOPS measures the total **B**it **Op**eration**s** for calculating a layer [7]. It is a common metric for evaluating the computation of quantized neural networks. For a model with L layers, $G_i^{(b_i)}$ denotes the BOPS of the i-th layer with b_i bit quantization, and we have:

$$\text{BOPS} = \sum\nolimits_{i=1}^{L} G_i^{(b_i)}, \tag{2.5}$$

Defining $b_{w_i},\ b_{a_i}$ to be the bit precision used for weight and activation of the i-th layer, then:

$$G_i^{(b_i)} = b_{w_i} b_{a_i} \text{MAC}_i, \tag{2.6}$$

where MAC_i is the total Multiply-Accumulate operations for computing the i-th layer.

Chapter 3

Motivations to Hardware-Aware Efficient Deep Learning

As soon as abstract mathematical computations were adapted to computation on digital computers, the problem of efficient representation, manipulation, and communication of the numerical values in those computations arose. Strongly related to the problem of numerical representation is the problem of quantization: in what manner should a set of continuous real-valued numbers be distributed over a fixed discrete set of numbers to minimize the number of bits required and also to maximize the accuracy of the attendant computations? In this chapter, we introduce the importance of quantization and its impact on hardware-aware efficient deep learning. We also consider more broadly other facets of HW-SW co-design.

3.1 Quantization

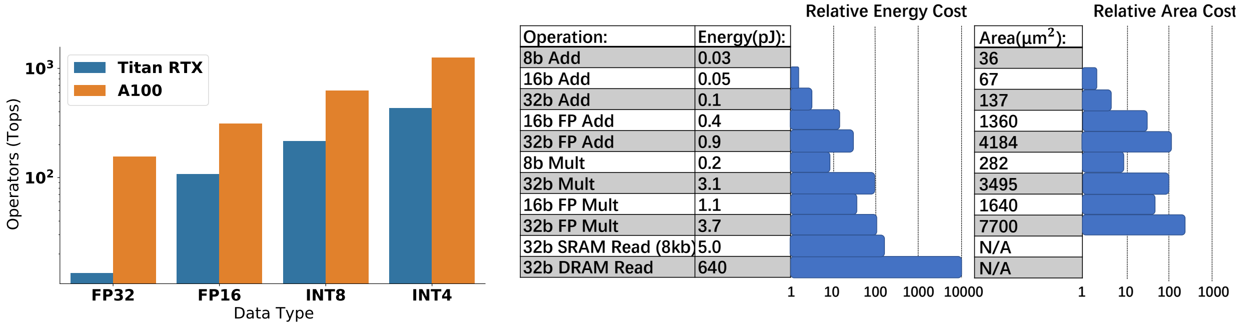
Quantization and Memory

This perennial problem of quantization becomes particularly important when memory resources are restricted, and it has come to the forefront in recent years due to the remarkable performance and the formidable parameter size (for example, 175B parameters in GPT-3 [18]) of advanced Neural Network models in computer vision, natural language processing, and related areas. Moving from floating-point representations to low-precision fixed integer values represented in 4 bits or less holds the potential to reduce the memory footprint by a factor of 16×, and in fact, reductions of 4× to 8× are often realized in practice in these applications. Therefore, it is not surprising that quantization has emerged recently as an important and very active sub-area of model compression. With the aid of more advanced quantization methods (such as mixed-precision quantization), a 16× reduction in memory consumption can become feasible, which significantly alleviates the memory bottleneck commonly found in computing systems nowadays.

Quantization and Latency

By quantizing the floating point values of weights and activations in a NN to integers, the model size can be shrunk significantly, without any modification to the architecture. This also allows one to use reduced-precision Arithmetic Logic Units (ALUs) which are faster and more power-efficient, as compared to floating point ALUs. As shown in Figure 3.1 (left), many hardware processors, including NVIDIA A100 and Titan RTX, support fast processing of low-precision arithmetic that can boost the inference throughput and latency.

[146] shows that INT8 inference of popular computer vision models, including ResNet50 [87], VGG-19 [214], and inceptionV3 [221] using TVM [27] quantization library, can achieve 3.89×, 3.32×, and 5.02× speedup on NVIDIA GTX 1080, respectively. [204] further shows that INT4 inference of ResNet50 could bring an additional 50-60% speedup on NVIDIA T4 and RTX, compared to its INT8 counterpart, emphasizing the importance of using lower bitwidth to maximize efficiency. Recently, [268] leverages mix-precision quantization to achieve 23% speedup for ResNet50, as compared to INT8 inference without accuracy degradation, and [119] extends INT8-only inference to the BERT model to enable up to 4.0× faster inference than FP32. While the aforementioned works focus on acceleration on GPUs, [108] also obtained 2.35× and 1.40× latency speedup on Intel Cascade Lake CPU and Raspberry Pi4 (which are both non-GPU architectures), respectively, through INT8 quantization of various vision models.



Operation:	Energy(pJ):	Area(μm^2):
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8kb)	5.0	N/A
32b DRAM Read	640	N/A

Figure 3.1: (Left) Comparison between peak operations for different bit-precision logic on Titan RTX and A100 GPU. (Right) Comparison of the corresponding energy cost and relative area cost for different precision for 45nm technology [93]. As one can see, lower precision provides better energy efficiency and higher throughput.

Quantization and Power Consumption

Quantization methods can reduce memory traffic volume, which is a significant source of energy consumption [93]. Moreover, as illustrated in Figure 3.1 (right), for a 45nm technology [93], low-precision logic is significantly more efficient in terms of energy consumption and area. For example, performing INT8 addition is 30× more energy efficient and 116× more area efficient as compared to FP32 addition [93].

Quantization and Accuracy

Theoretically, most current NN models are heavily over-parameterized. As a result, there is ample opportunity for reducing the bitwidth of parameters without impacting accuracy. In practice, 8-bit quantization of weights generally results in trivial accuracy degradation compared to the floating point counterpart. There are also cases where the 8-bit quantized models actually outperform the original pre-trained models, due to the fact that an appropriate level of quantization serves as a regularizer of the neural network. An 8-bit quantization on the activations can potentially cause accuracy degradation, due to the instability and large quantization range of activations in specific NN models. It has been observed that this accuracy degradation tends to be non-trivial for the recent transformer-based or mlp-based neural networks.

Post-training quantization (PTQ) with bitwidth between 4 and 8 generally leads to moderate accuracy degradation, while this gap can be well alleviated by applying quantization-aware training (QAT). With the assistance of advanced quantization methods, 4-bit quantization often achieves the sweet point, with a tolerable accuracy degradation (for example, 1% accuracy on ImageNet) and a much faster inference. Finally, directly applying quantization with ultra-low bitwidth (lower than 4) may lead to a severe accuracy drop, while the accuracy of the quantized models can become comparable to the baseline after utilizing cutting-edge quantization techniques, as we will show in the latter sections.

3.2 HW-SW Co-Design

Diversity of Hardware Platforms

Figure 3.2 plots the throughput of different commercial edge processors that are widely used for NN inference at the edge. In the past few years, there has been a significant improvement in the computing power of the edge processors, and this allows deployment and inference of costly NN models that were previously available only on servers. We should note that the overhead of a NN component (in terms of latency, power consumption, etc.) is hardware-dependent. For example, hardware with a dedicated cache hierarchy can execute bandwidth-bound operations much more efficiently than hardware without such a cache hierarchy. Given the diversity of different hardware platforms, there exists no universal operations or neural architectures that can be efficient on all hardware devices. As a result, solely trying to optimize the software part of the pipeline (neural architecture and model compression) can be sub-optimal. It is important to adapt the NN architectures and model compression for a particular target hardware platform. Taking the efficiency metrics in Chapter 2 into consideration can make the algorithms become hardware-aware. Moreover, incorporating the

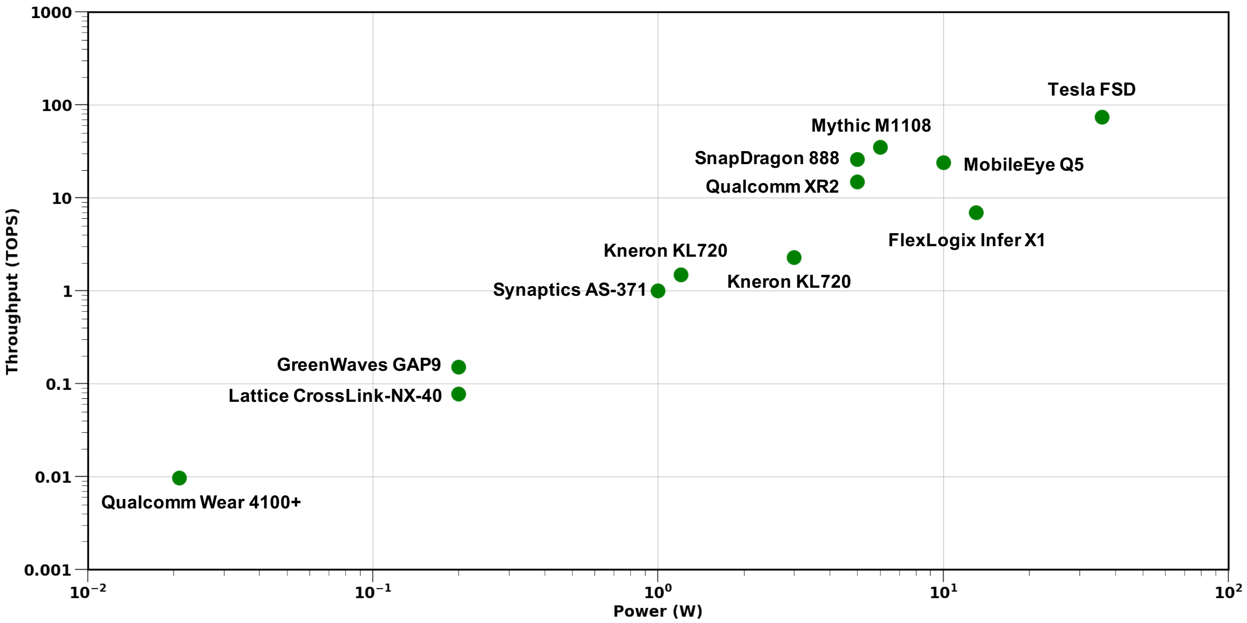


Figure 3.2: Throughput and power of different commercial edge processors for NN inference.

hardware specifications into the design space can further improve the utilization of resources and boost accuracy.

HW-SW Co-Design and Latency

HW-SW co-design is crucial in optimizing the latency. Firstly, for model compression methods, no accuracy gain can be obtained without a proper hardware design. Specifically, the latency of a 4-bit quantized network is the same as the 8-bit quantized counterpart on most GPUs, because only 8-bit integers are supported on the hardware. However, on the NVIDIA T4 GPU with 4-bit tensor cores, which are specialized execution units designed for efficient 4-bit matrix multiplications, the latency of a 4-bit quantized network can be significantly decreased. Secondly, for different neural architectures or operations, an HW-SW co-design is generally able to maintain accuracy while achieving non-trivial speed-up. As an example, our work CoDeNet [102] has shown that limiting and rounding the offset range in deformable convolution will not hurt its expressibility, while these modifications make the operation hardware-friendly and can therefore run efficiently on our specially designed engine on FPGA.

HW-SW Co-Design and Accuracy

Since the HW-SW Co-Design search space contains the software search space (neural architecture and model compression), theoretically, the optimal setting in the co-design space should always achieve a better trade-off between the accuracy and the latency than the setting found in the original space. For example, as shown in CoDeNet [102], a detector with co-designed deformable convolution can achieve comparable results to the baseline (36.8 AP50 on Microsoft COCO versus 38.4). Meanwhile, the co-designed depthwise 3×3 deformable convolution only has a 2.1 ms latency running on the FPGA, while the counterpart used in the baseline has a 10× larger latency of 20.5 ms. Although the HW-SW co-design can be advantageous, including the hardware specifications makes the search space too large to be fully optimized. There is a chance that good settings are not found by manual efforts or searching methods, leading to sub-optimal accuracy or latency. As such, good HW-SW co-design methods are necessary for the whole design pipeline and for real-world applications.

Chapter 4

Introduction and Related Work of Quantization

Having motivated quantization in the previous chapter, we now proceed to discuss more formally the process of efficiently arriving at quantized values of a NN. Assume that the NN has L layers with learnable parameters, denoted as $\{W_1, W_2, ..., W_L\}$, with θ denoting the combination of all such parameters. Without loss of generality, we focus on the supervised learning problem, where the nominal goal is to optimize the following empirical risk minimization function:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta), \tag{4.1}$$

where (x, y) is the input data and the corresponding label, $l(x, y; \theta)$ is the loss function (e.g., Mean Squared Error or Cross Entropy loss), and N is the total number of data points. Let us also denote the input hidden activations of the i^{th} layer as h_i, and the corresponding output hidden activation as a_i. We assume that we have the trained model parameters θ, stored in floating point precision. In quantization, the goal is to reduce the precision of both the parameters (θ), as well as the intermediate activation maps (i.e., $h_i,\ a_i$) to low-precision, with minimal impact on the generalization power/accuracy of the model. To do this, we need to define a quantization operator that maps a floating point value to a quantized one, which is described next.

4.1 Linear & Non-linear Quantizers

We need first to define a function that can quantize NN weights and activations to a finite set of values. This function takes real values in floating points, and it maps them to a lower precision range, as illustrated in Figure 4.1. A popular choice for a quantization function is as follows:

$$Q(r) = \text{Int}\big(r/S\big) - Z, \tag{4.2}$$

where Q is the quantization operator, r is a real-valued input (activation or weight), S is a real-valued scaling factor, and Z is an integer zero point.

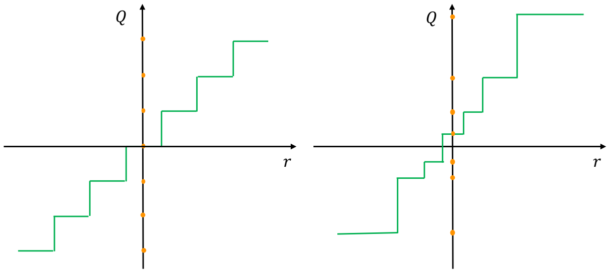


Figure 4.1: Comparison between uniform quantization (left) and non-uniform quantization (right). Real values in the continuous domain r are mapped into discrete, lower precision values in the quantized domain Q, which are marked with orange bullets. Note that the distances between the quantized values (quantization levels) are the same in uniform quantization, whereas they can vary in non-uniform quantization.

Furthermore, the Int function maps a real value to an integer value through a rounding operation (e.g., round to nearest and truncation). In essence, this function is a mapping from real values r to some integer values. This method of quantization is also known as *uniform quantization*, as the resulting quantized values (aka quantization levels) are uniformly spaced (Figure 4.1, left). There are also *non-uniform quantization* methods whose quantized values are not necessarily uniformly spaced (Figure 4.1, right). It is possible to recover real values r from the quantized values $Q(r)$ through an operation that is often referred to as *dequantization*:

$$\tilde{r} = S(Q(r) + Z). \tag{4.3}$$

Note that the recovered real values $\tilde{r}$ will not exactly match r due to the rounding operation.

One important factor in uniform quantization is the choice of the scaling factor S in Eq. 4.2. This scaling factor essentially divides a given range of real values r into a number of partitions (as discussed in [123, 107, 69]):

$$S = \frac{\beta - \alpha}{2^b - 1}, \tag{4.4}$$

where $[\alpha, \beta]$ denotes the clipping range, a bounded range that we are clipping the real values with, and b is the quantization bit width. Therefore, in order for the scaling factor to be defined, the clipping range $[\alpha, \beta]$ should first be determined. The process of choosing the clipping range is often referred to as *calibration*. A straightforward choice is to use the min/max of the signal for the clipping range, i.e., $\alpha = r_{min}$, and $\beta = r_{max}$.

This approach is an *asymmetric quantization* scheme, since the clipping range is not necessarily symmetric with respect to the origin, i.e., $-\alpha \neq \beta$, as illustrated in Figure 4.2 (Right). It is also possible to use a *symmetric quantization* scheme by choosing a symmetric clipping range of $\alpha = -\beta$. A popular choice is to choose these based on the min/max values of

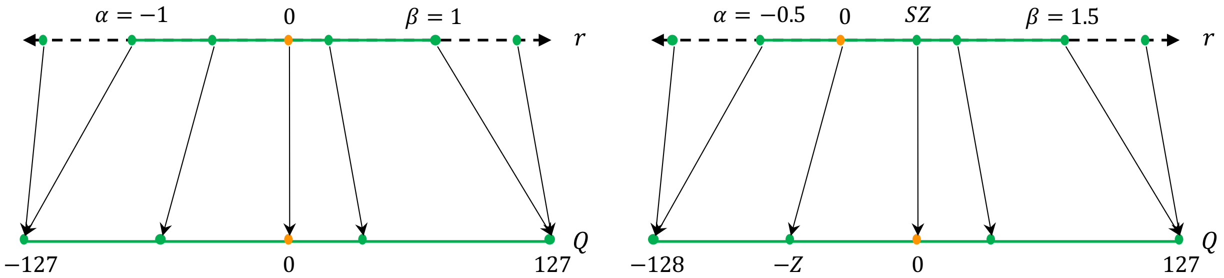


Figure 4.2: Illustration of symmetric quantization and asymmetric quantization. Symmetric quantization with restricted range maps real values to [-127, 127], and full range maps to [-128, 127] for 8-bit quantization.

the signal: $-\alpha = \beta = \max(|r_{max}|, |r_{min}|)$. Asymmetric quantization often results in a tighter clipping range as compared to symmetric quantization. This is especially important when the target weights or activations are imbalanced, e.g., the activation after ReLU that always has non-negative values. Using symmetric quantization, however, simplifies the quantization function in Eq. 4.2 by replacing the zero point with $Z = 0$:

$$Q(r) = \text{Int}\left(\frac{r}{S}\right). \tag{4.5}$$

Here, there are two choices for the scaling factor. In "full range" symmetric quantization S is chosen as $\frac{2max(|r|)}{2^n - 1}$ (with floor rounding mode), to use the full INT8 range of [-128,127]. However, in "restricted range" S is chosen as $\frac{max(|r|)}{2^{n-1}-1}$, which only uses the range of [-127,127]. As expected, the full range approach is more accurate. Symmetric quantization is widely adopted in practice for quantizing weights because zeroing out the zero point can lead to a reduction in computational cost during inference [253], and also makes the implementation more straightforward. However, note that for activations the cross terms occupying due to the offset in the asymmetric activations are a static data independent term and can be absorbed in the bias (or used to initialize the accumulator) [15].

Using the min/max of the signal for both symmetric and asymmetric quantization is a popular method. However, this approach is susceptible to outlier data in the activations. These could unnecessarily increase the range and, as a result, reduce the resolution of quantization. One approach to address this is to use percentile instead of min/max of the signal [167]. That is to say, instead of the largest/smallest value, the i-th largest/smallest values are used as β/α. Another approach is to select α and β to minimize KL divergence (i.e., information loss) between the real values and the quantized values [172]. We refer the interested readers to [253] where the different calibration methods are evaluated on various models.

Some work in the literature has also explored non-uniform quantization [73, 254, 77, 94, 148, 173, 35, 23, 187, 277, 240, 111, 117, 260, 61, 233, 286, 186, 265, 141], where quantization steps as well as quantization levels are allowed to be non-uniformly spaced.

The formal definition of non-uniform quantization is shown in Eq. 4.6, where X_i represents the discrete quantization levels and Δ_i the quantization steps (thresholds):

$$Q(r) = X_i, \text{ if } r \in [\Delta_i, \Delta_{i+1}). \tag{4.6}$$

Specifically, when the value of a real number r falls in between the quantization step Δ_i and Δ_{i+1}, quantizer Q projects it to the corresponding quantization level X_i. Note that neither X_i's nor Δ_i's are uniformly spaced.

Non-uniform quantization may achieve higher accuracy for a fixed bit-width, because one could better capture the distributions by focusing more on important value regions or finding appropriate dynamic ranges. For instance, many non-uniform quantization methods have been designed for bell-shaped distributions of the weights and activations that often involve long tails [12, 23, 137, 173, 109, 60]. A typical rule-based non-uniform quantization is to use a logarithmic distribution [173, 287], where the quantization steps and levels increase exponentially instead of linearly. Another popular branch is *binary-code-based* quantization [111, 255, 277, 75, 103] where a real-number vector $\mathbf{r} \in \mathbb{R}^n$ is quantized into m binary vectors by representing $\mathbf{r} \approx \sum_{i=1}^{m} \alpha_i \mathbf{b}_i$, with the scaling factors $\alpha_i \in \mathbb{R}$ and the binary vectors $\mathbf{b}_i \in \{-1, +1\}^n$. Since there is no closed-form solution for minimizing the error between $\mathbf{r}$ and $\sum_{i=1}^{m} \alpha_i \mathbf{b}_i$, previous research relies on heuristic solutions. To further improve the quantizer, more recent work [227, 75, 255] formulates non-uniform quantization as an optimization problem. As shown in Eq. 4.7, the quantization steps/levels in the quantizer Q are adjusted to minimize the difference between the original tensor and the quantized counterpart.

$$\min_{Q} \|Q(r) - r\|^2 \tag{4.7}$$

Furthermore, the quantizer itself can also be jointly trained with the model parameters. These methods are referred to as learnable quantizers, and the quantization steps/levels are generally trained with iterative optimization [277, 255] or gradient descent [147, 117, 260].

In addition to rule-based and optimization-based non-uniform quantization, clustering can also be beneficial to alleviate the information loss due to quantization. Some works [73, 254] use k-means on different tensors to determine the quantization steps and levels, while another work [35] applies a Hessian-weighted k-means clustering on weights to minimize the performance loss.

4.2 QAT & PTQ

It is often necessary to adjust the parameters in the NN after quantization. This can either be performed by re-training the model, a process that is called Quantization-Aware Training (QAT), or done without re-training, a process that is often referred to as Post-Training Quantization (PTQ). Schematic comparison between these two approaches is illustrated in Figure 4.3, and further discussed below (we refer interested readers to [177] for a more detailed discussion on this topic).

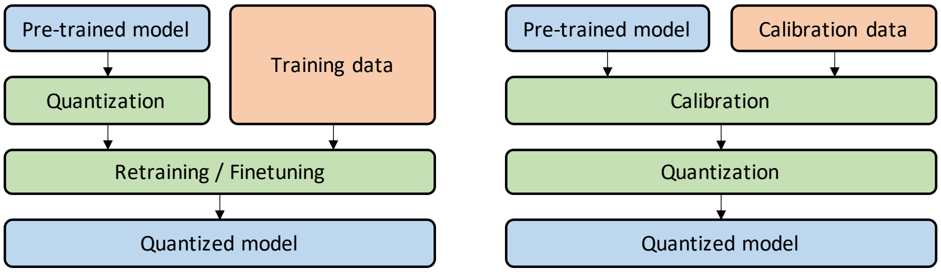


Figure 4.3: Comparison between Quantization-Aware Training (QAT, Left) and Post-Training Quantization (PTQ, Right). In QAT, a pre-trained model is quantized and then finetuned using training data to adjust parameters and recover accuracy degradation. In PTQ, a pre-trained model is calibrated using calibration data (e.g., a small subset of training data) to compute the clipping ranges and the scaling factors. Then, the model is quantized based on the calibration result. Note that the calibration process is often conducted in parallel with the finetuning process for QAT.

Quantization-Aware Training

Given a trained model, quantization may introduce a perturbation to the trained model parameters, and this can push the model away from the point to which it converged when it was trained with floating point precision. It is possible to address this by re-training the NN model with quantized parameters so that the model can converge to a point with better loss. One popular approach is to use Quantization-Aware Training (QAT), in which the usual forward and backward pass are performed on the quantized model in floating point, but the model parameters are quantized after each gradient update (similar to projected gradient descent). In particular, it is important to do this projection after the weight update is performed in floating point precision. Performing the backward pass with floating point is important, as accumulating the gradients in quantized precision can result in zero-gradient or gradients that have a high error, especially in low-precision [38, 148, 103, 196, 78, 79, 222, 182].

An important subtlety in backpropagation is how the non-differentiable quantization operator (Equation 4.2) is treated. Without any approximation, the gradient of this operator is zero almost everywhere, since the rounding operation in Equation 4.2 is a piece-wise flat operator. A popular approach to address this is to approximate the gradient of this operator by the so-called Straight Through Estimator (STE) [14]. STE essentially ignores the rounding operation and approximates it with an identity function.

Despite the coarse approximation of STE, it often works well in practice, except for ultra low-precision quantization such as binary quantization [8]. The work of [271] provides a theoretical justification for this phenomenon, and it finds that the coarse gradient approximation of STE can in expectation correlate with population gradient (for a proper choice of STE). From a historical perspective, we should note that the original idea of STE can be traced back to the seminal work of [200, 199], where an identity operator was used to approximate gradient from the binary neurons.

While STE is the mainstream approach [294, 216], other approaches have also been explored in the literature [26, 58, 23, 156, 128, 3]. We should first mention that [14] also proposes a stochastic neuron approach as an alternative to STE. Other approaches using combinatorial optimization [64], target propagation [126], or Gumbel-softmax [110] have also been proposed. Another different class of alternative methods tries to use regularization operators to enforce the weight to be quantized. This removes the need to use the non-differentiable quantization operator in Equation 4.2. These are often referred to as *Non-STE* methods [34, 8, 181, 128, 94, 287, 4]. Recent research in this area includes ProxQuant [8] which removes the rounding operation in the quantization formula Equation 4.2, and instead uses the so-called *W-shape*, non-smooth regularization function to enforce the weights to quantized values. Other notable research includes using pulse training to approximate the derivative of discontinuous points [43], or replacing the quantized weights with an affine combination of floating point and quantized parameters [157]. The recent work of [179] also suggests AdaRound, which is an adaptive rounding method as an alternative to the round-to-nearest method. Despite interesting works in this area, these methods often require a lot of tuning and so far STE approach is the most commonly used method.

In addition to adjusting model parameters, some prior work found it effective to learn quantization parameters during QAT as well. PACT [33] learns the clipping ranges of activations under uniform quantization, while QIT [117] also learns quantization steps and levels as an extension to a non-uniform quantization setting. LSQ [57] introduces a new gradient estimate to learn scaling factors for non-negative activations (e.g., ReLU) during QAT, and LSQ+ [15] further extends this idea to general activation functions such as swish [195] and h-swish [95] that produce negative values.

Post-Training Quantization

An alternative to the expensive QAT method is Post-Training Quantization (PTQ) which performs the quantization and the adjustments of the weights, without any fine-tuning [10, 169, 37, 284, 60, 59, 127, 178, 21, 138, 89, 67, 68, 104, 213]. As such, the overhead of PTQ is very low and often negligible. Unlike QAT, which requires a sufficient amount of training data for retraining, PTQ has an additional advantage in that it can be applied in situations where data is limited or unlabeled. However, this often comes at the cost of lower accuracy as compared to QAT, especially for low-precision quantization.

For this reason, multiple approaches have been proposed to mitigate the accuracy degradation of PTQ. For example, [10, 63] observe inherent bias in the mean and variance of the weight values following their quantization and propose bias correction methods; and [169, 178] show that equalizing the weight ranges (and implicitly activation ranges) between different layers or channels can reduce quantization errors. ACIQ [10] analytically computes the optimal clipping range and the channel-wise bitwidth setting for PTQ. Although ACIQ can achieve low accuracy degradation, the channel-wise activation quantization used in ACIQ is hard to efficiently deploy on hardware. In order to address this, the OMSE method [37] re-

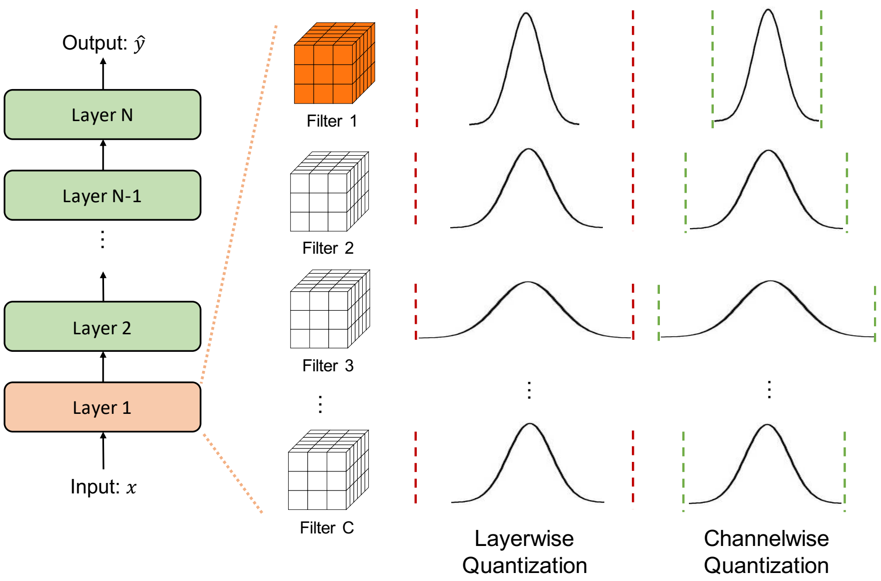


Figure 4.4: Illustration of different quantization granularities. In layerwise quantization, the same clipping range is applied to all the filters that belong to the same layer. This can result in bad quantization resolution for the channels that have narrow distributions (e.g., Filter 1 in the figure). One can achieve better quantization resolution using channelwise quantization that dedicates different clipping ranges to different channels.

moves channel-wise quantization on activation and proposes to conduct PTQ by optimizing the L2 distance between the quantized tensor and the corresponding floating point tensor. Furthermore, to better alleviate the adverse impact of outliers on PTQ, an outlier channel splitting (OCS) method is proposed in [284] which duplicates and halves the channels containing outlier values. Another notable work is AdaRound [179] which shows that the naive round-to-nearest method for quantization can counter-intuitively results in sub-optimal solutions, and it proposes an adaptive rounding method that better reduces the loss. While AdaRound restricts the changes of the quantized weights to be within ±1 from their full-precision counterparts, AdaQuant [104] proposes a more general method that allows the quantized weights to change as needed. PTQ schemes can be taken to the extreme, where neither training nor testing data are utilized during quantization (aka zero-shot scenarios), as discussed in [21].

4.3 Quantization Granularity

In most computer vision tasks, the activation input to a layer is convolved with many different convolutional filters, as illustrated in Figure 4.4. Each of these convolutional filters can have a different range of values. As such, one differentiator for quantization methods is the granularity of how the clipping range $[\alpha, \beta]$ is calculated for the weights. We categorized them as follows.

Layerwise Quantization In this approach, the clipping range is determined by considering all of the weights in convolutional filters of a layer [123, 70], as shown in the third column of Figure 4.4. Here one examines the statistics of the entire parameters in that layer (e.g., min, max, percentile, etc.), and then uses the same clipping range for all the convolutional filters. While this approach is very simple to implement, it often results in sub-optimal accuracy, as the range of each convolutional filter can vary a lot. For example, a convolutional kernel that has a relatively narrower range of parameters may lose its quantization resolution due to another kernel in the same layer with a wider range.

Groupwise Quantization One could group multiple different channels inside a layer to calculate the clipping range (of either activations or convolution kernels). This could be helpful for cases where the distribution of the parameters across a single convolution/activation varies a lot. For instance, this approach was found useful in Q-BERT [211] for quantizing Transformer [234] models that consist of fully-connected attention layers. However, this approach inevitably comes with the extra cost of accounting for different scaling factors.

Channelwise Quantization A popular choice of the clipping range is to use a fixed value for each convolutional filter, independent of other channels [288, 277, 107, 123, 102, 212], as shown in the last column of Figure 4.4. That is to say, each channel is assigned a dedicated scaling factor. This ensures a better quantization resolution and often results in higher accuracy. Channelwise quantization is currently the standard method used for quantizing convolutional kernels. It generally comes with negligible overhead.

Sub-channelwise Quantization The previous approach could be taken to the extreme, where the clipping range is determined with respect to any groups of parameters in convolution or fully-connected layer. However, this approach could add considerable overhead, since the different scaling factors need to be taken into account when processing a single convolution or full-connected layer. Therefore, groupwise quantization could establish a good compromise between the quantization resolution and the computation overhead.

4.4 Uniform & Mixed-Precision Quantization

It is easy to see that the hardware performance improves as we use lower precision quantization. However, uniformly quantizing a model to ultra low-precision can cause significant accuracy degradation. It is possible to address this with mixed-precision quantization [290, 236, 53, 7, 259, 192, 241, 97, 183, 80, 202, 151, 54, 285]. In this approach, each layer is quantized with different bit precision, as illustrated in Figure 4.5. One challenge with this approach is that the search space for choosing this bit setting is exponential in the number of layers. Different approaches have been proposed to address this huge search space.

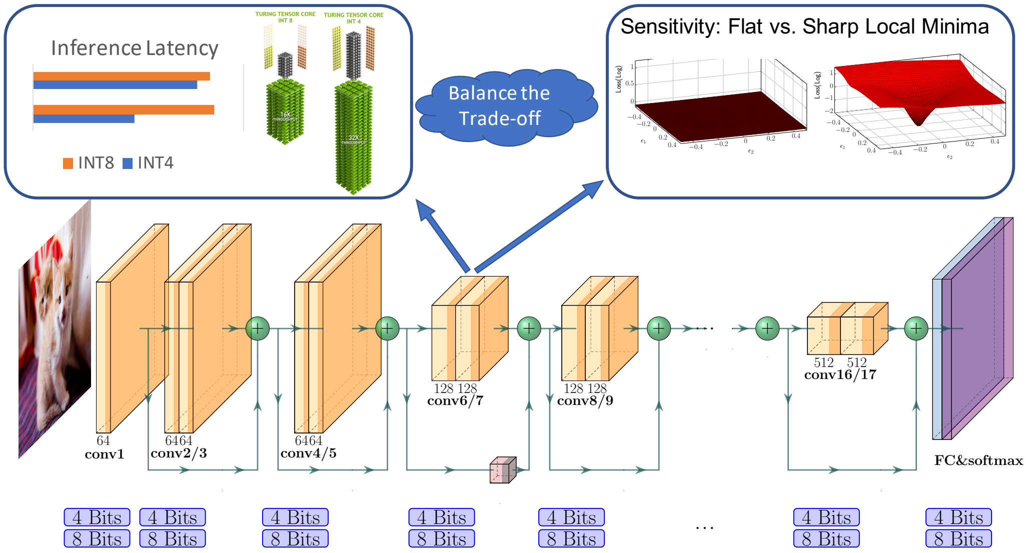


Figure 4.5: Illustration of mixed-precision quantization. In mixed-precision quantization, the goal is to keep sensitive and efficient layers in higher precision, and only apply low-precision quantization to insensitive and inefficient layers. The efficiency metric is hardware dependent, and it could be latency or energy consumption.

Selecting this mixed-precision for each layer is essentially a searching problem, and many different methods have been proposed for it. The recent work of [236] proposed a reinforcement learning (RL) based method to determine automatically the quantization policy, and the authors used a hardware simulator to take the hardware accelerator's feedback in the RL agent feedback. The paper [251] formulated the mixed-precision configuration searching problem as a Neural Architecture Search (NAS) problem and used the Differentiable NAS (DNAS) method to efficiently explore the search space. One disadvantage of these exploration-based methods [236, 251] is that they often require large computational resources, and their performance is typically sensitive to hyperparameters and even initialization.

Another class of mixed-precision methods uses periodic function regularization to train mixed-precision models by automatically distinguishing different layers and their varying importance with respect to accuracy while learning their respective bitwidths [181].

Different than these exploration and regularization-based approaches, HAWQ [53] introduces an automatic way to find the mixed-precision settings based on the second-order sensitivity of the model. It was theoretically shown that the trace of the second-order operator (i.e., the Hessian) can be used to measure the sensitivity of a layer to quantization [52]. In HAWQv2, this method was extended to mixed-precision activation quantization [52], and was shown to be much faster than RL-based mixed-precision methods [236]. In HAWQv3, an integer-only, hardware-aware quantization was introduced [268] that proposed a fast Integer Linear Programming method to find the optimal bit precision for a given application-specific constraint (e.g., model size or latency). This work also addressed the common question about the hardware efficiency of mixed-precision quantization by directly deploying them on T4 GPUs, showing up to 50% speed up with mixed-precision (INT4/INT8) quantization as compared to INT8 quantization.

Chapter 5

Quantization: HAWQ

In this Chapter, we discuss how to perform mixed-precision quantization focusing on the following *key problem*:

> Can we efficiently find feasible solutions without applying time-consuming and costly searching algorithms on the exponentially large search space of mixed-precision quantization?

We propose Hessian-AWare Quantization (HAWQ) to address this problem by taking advantage of the Hessian information.

5.1 Method

One possible approach that can be used to measure quantization sensitivity is to use first-order information, based on the gradient vector. However, the gradient can be very misleading. This can be easily illustrated by considering a simple 1-d parabolic function of the form $y = \frac{1}{2}ax^2$ at origin (*i.e.*, $x = 0$). The gradient signal at the origin is zero, irrespective of the value of a. However, this does not mean that the function is not sensitive to perturbation in x. We can get better metrics for sensitivity by using second-order information, based on the Hessian matrix. This clearly shows that higher values of a result in more sensitivity to input perturbations.

For the case of high dimensions, the second order information is stored in the Hessian matrix, of size $n_i \times n_i$ for each block. For this case, we can compute the eigenvalues of the Hessian to measure sensitivity, as described next.

Second-Order Information

We compute the eigenvalues of the Hessian (*i.e.*, the second-order operator) of each block in the network. Note that it is not possible to explicitly form the Hessian since the size of a block (denoted by n_i for ith block) can be quite large. However, it is possible to compute the Hessian eigenvalues without explicitly forming the Hessian, using a matrix-free power iteration algorithm [269, 166, 270]. This method requires the computation of the so-called Hessian *matvec*, which is the result of the multiplication of the Hessian matrix with a given (possibly random) vector v. To illustrate how this can be done for a deep network, let us first denote g_i as the gradient of loss L with respect to the i^{th} block parameters,

$$g_i = \frac{\partial L}{\partial W_i}. \tag{5.1}$$

For a random vector v (which has the same dimension as g_i), we have:

$$\frac{\partial (g_i^T v)}{\partial W_i} = \frac{\partial g_i^T}{\partial W_i} v + g_i^T \frac{\partial v}{\partial W_i} = \frac{\partial g_i^T}{\partial W_i} v = H_i v, \tag{5.2}$$

where H_i is the Hessian matrix of L with respect to W_i. We can then use the power-iteration method to compute the top eigenvalue of H_i, as shown in Algorithm 1. Intuitively the algorithm requires multiple evaluations of the Hessian matvec, which can be computed using Equation 5.2.

Algorithm 1: Power Iteration for Hessian Eigenvalue Computation

```
Block Parameter: $W_i$.
Compute the gradient of $W_i$ by backpropagation, $i.e.$, $g_i = \frac{dL}{dW_i}$.
Draw a random vector $v$ (same dimension as $W_i$).
Normalize $v$, $v = \frac{v}{\|v\|}$
for $i = 1, 2, \ldots, n$ do                                  // Power Iteration
    Compute $gv = g_i^T v$                                     // Inner product
    Compute $Hv$ by backpropagation, $Hv = \frac{d(gv)}{dW_i}$   // Get Hessian vector
     product
    Normalize and reset $v$, $v = \frac{Hv}{\|Hv\|}$
end
```

It is well known that, based on the theory of Minimum Description Length (MDL), fewer bits are required to specify a flat region up to a given threshold, and vice versa for a region with sharp curvature [198, 91]. The intuition for this is that the noise created by the imprecise location of a flat region is not magnified for a flat region, making it more amenable to aggressive quantization. The opposite is true for sharp regions, in that even small round-off errors may be amplified. Therefore, it is expected that layers with a higher Hessian spectrum (*i.e.*, larger eigenvalues) are more sensitive to quantization. The distribution of these eigenvalues for different blocks are shown in Figure 6.1 for ResNet20 on CIFAR-10 and Inception-V3 on ImageNet. As one can see, different blocks exhibit orders of magnitude differences in the Hessian spectrum. For instance, ResNet20 is an order of magnitude more sensitive to perturbations to its 9^{th} block, than its last block.

To further illustrate this, we provide 1D visualizations of the loss landscape as well. To this end, we first compute the Hessian eigenvector of each block, and we perturb each block individually along the eigenvector and compute how the loss changes. It can be clearly seen that blocks with larger Hessian eigenvalue (*i.e.*, sharper curvature) exhibit larger fluctuations in the loss, as compared to those with smaller Hessian eigenvalue (*i.e.*, flatter curvature). A corresponding 3D plot is also shown in Figure 6.1, where instead of just considering the top eigenvector, we also compute the second top eigenvector and visualize the loss by perturbing the weights along these two directions. These surface plots are computed for the 9^{th} and the last blocks of ResNet20, as well as the 2nd and last blocks of Inception-V3.

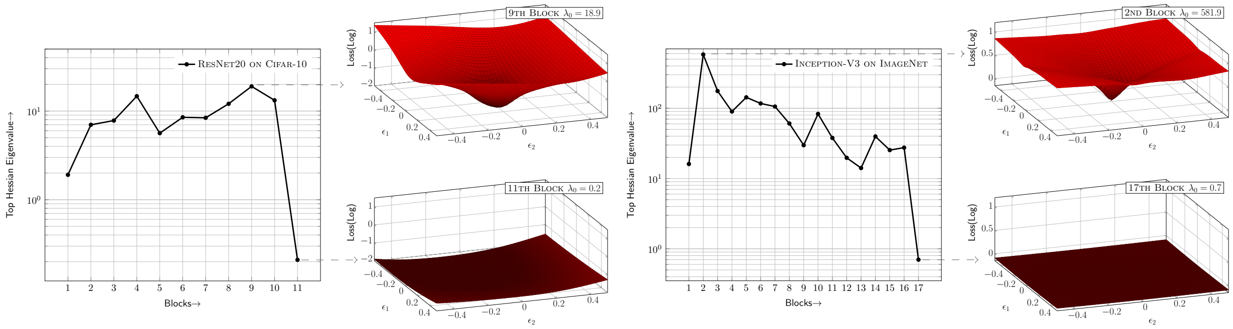


Figure 5.1: Top eigenvalue of each individual block of pre-trained ResNet20 on Cifar-10 (Left), and Inception-V3 on ImageNet (Right). Note that the magnitudes of eigenvalues of different blocks vary by orders of magnitude.

Algorithm

We approximate the Hessian as a block diagonal matrix, scaled by its top eigenvalue λ as $\{H_i \approx \lambda_i I\}_{i=1}^m$, where m is the number of blocks in the network. Based on the MDL theory, layers with large λ cannot be quantized to ultra-low precision without significant perturbation to the model. Thus we can use the Hessian spectrum of each block to sort the different blocks and perform less aggressive quantization to layers with a large Hessian spectrum. However, some of these blocks may contain a very large number of parameters, and using higher bits here would lead to a large memory footprint of the quantized network. Therefore, as a compromise, we weight the Hessian spectrum with the block's memory footprint and use the following metric for sorting the blocks:

$$S_i = \lambda_i / n_i, \tag{5.3}$$

where λ_i is the top eigenvalue of H_i. Based on this sorting, layers that have a large number of parameters and have small eigenvalue would be quantized to lower bits, and vice versa. That is, after S_i is computed, we sort S_i in descending order and use it as a metric to determine the quantization precision.[1]

[1] Note that, as mentioned in the limitations section, S_i does not give us the exact bit precision but a

Algorithm 2: Hessian AWare Quantization

```
Block-wise Hessian eigenvalues $\lambda_i$ (computed from Algorithm 1), and block
  parameter size $n_i$ for $i = 1, \cdots, m$.
for $i = 1, 2, \ldots, m$ do      // Quantization Precision
    $S_i = \lambda_i / n_i$                       // See Equation 5.3
end
Order $S_i$ in descending order and determine relative quantization precision for each
  block.
Compute $\Delta W_i$.
for $i = 1, 2, \ldots, m$ do      // Fine-Tuning Order
    $\Omega_i = \lambda_i \|\Delta W_i\|^2$                  // See Equation 6.9
end
Order $\Omega_i$ in descending order and perform block-wise fine-tuning
```

Quantization-aware re-training of the neural network is necessary to recover performance which can sharply drop due to ultra-low precision quantization. A straightforward way to do this is to re-train (hereafter referred to as fine-tune) the whole quantized network at once. However, as we will discuss in Section 5.2, this can lead to sub-optimal results. A better strategy is to perform multi-stage fine-tuning. However, the order in multi-stage tuning is important and different orders could lead to very different accuracies.

We sort different blocks for fine-tuning based on the following metric:

$$\Omega_i = \lambda_i \|Q(W_i) - W_i\|_2^2, \tag{5.4}$$

where i refers to i^{th} block, λ_i is the Hessian eigenvalue, and $\|Q(W_i) - W_i\|_2$ is the L_2 norm of quantization perturbation. The intuition here is to first fine-tune layers that have high curvature as well as a large number of parameters that cause more perturbations after quantization. Note that the latter metric depends on the bits used for quantization and thus is not a fixed metric. The motivation for choosing this order is that fine-tuning blocks with large Ω_i can significantly affect other blocks, thus making prior fine-tuning of layers with small Ω_i futile.

5.2 Experiments

Cifar-10 After computing the eigenvalues of block Hessian (shown in Figure 6.1), we compute the weighted sensitivity metric of Eq. 5.3, along with Ω_i based on Eq. 6.9. We then perform the quantization based on the HAWQ algorithm. Results are shown in Table 5.1.

For comparison, we test the quantization performance without using the Hessian information, which we refer to as "Direct" method, as well as other methods in the literature including Dorefa [288], PACT [33], LQ-Net [277], and DNAS [251], as shown in Table 5.1.

relative ordering for the bits of different blocks.

Table 5.1: Quantization results of ResNet20 on Cifar-10. We abbreviate quantization bits used for weights as "w-bits," activations as "a-bits," testing accuracy as "Acc," and the compression ratio of weights/activations as "W-Comp/A-Comp." Furthermore, we show results without using Hessian information ("Direct"), as well as other state-of-the-art methods [288, 33, 277]. In particular, we compare with the recent proposed DNAS approach of [251]. Our method achieves similar testing performance with a significantly higher compression ratio (especially in activations). Here "MP" refers to mixed-precision quantization, and the lowest bits used for weights and activations are reported. Also note that [288, 33, 277] use 8-bit for the first and last layers. The exact per-layer configuration for mixed-precision quantized ResNet20 is presented in the Appendix of [53].

Quantization	w-bits	a-bits	Acc	W-Comp	A-Comp
Baseline	32	32	92.37	1.00×	1.00×
Dorefa [288]	2	2	88.20	16.00×	16.00×
Dorefa [288]	3	3	89.90	10.67×	10.67×
PACT [33]	2	2	89.70	16.00×	16.00×
PACT [33]	3	3	91.10	10.67×	10.67×
LQ-Nets [277]	2	2	90.20	16.00×	16.00×
LQ-Nets [277]	3	3	91.60	10.67×	10.67×
LQ-Nets [277]	2	32	91.80	16.00×	1.00×
LQ-Nets [277]	3	32	92.00	10.67×	1.00×
DNAS [251]	1 MP	32	92.00	16.60×	1.00×
DNAS [251]	1 MP	32	**92.72**	11.60×	1.00×
Direct	2 MP	4	90.34	16.00×	8.00×
HAWQ	2 MP	4	**92.22**	13.11×	8.00×

For methods that use Mixed-Precision (MP) quantization, the lowest bits used for weights ("w-bits"), and activations ("a-bits") are reported.

The Direct method achieves good compression, but it results in a 2.03% accuracy drop, as shown in Table 5.1. Furthermore, a comparison with other state-of-the-art shows a similar trend. There have been several methods proposed in the literature to address this reduction, with the latest method introduced in [277], where a learnable quantization method is used. As one can see, LQ-Nets results in 0.77% accuracy degradation with 10.67× compression ratio, whereas HAWQ has only 0.15% accuracy drop with 13.11× compression. Moreover, HAWQ achieves similar accuracy as compared to DNAS [251] but with 8× higher compression ratio for activations.

ImageNet Here, we test the HAWQ method for quantizing Inception-V3 [221] on ImageNet. Inception-V3 is appealing for efficient hardware implementation, as it does not use any residual connections. Such non-linear structures create dependencies that may be very

Table 5.2: Quantization results of Inception-V3 on ImageNet. We abbreviate quantization bits used for weights as "w-bits," activations as "a-bits," top-1 testing accuracy as "Top-1," and weight compression ratio as "W-Comp." Furthermore, we compare HAWQ with the direct quantization method without using Hessian ("Direct") and Integer-Only method [107]. Here "MP" refers to mixed-precision quantization. We report the exact per-layer configuration for mixed-precision quantization in the appendix. Compared to [107, 187], we achieve a higher compression ratio with higher testing accuracy.

Method	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	32	32	77.45	1.00×	91.2
Integer-Only [107]	8	8	75.40	4.00×	22.8
Integer-Only [107]	7	7	75.00	4.57×	20.0
RVQuant [187]	3 MP	3 MP	74.14	10.67×	8.55
Direct	2 MP	4 MP	69.76	15.88×	5.74
HAWQ	2 MP	4 MP	75.52	12.04×	**7.57**

difficult to optimize for fast inference [264]. As before, we first compute the block Hessian eigenvalues, which are reported in Figure 6.1, and then compute the corresponding weighted sensitivity metric.

We report the quantization results in Table 6.1, where as before we compare with a direct quantization, as well as recently proposed "Integer-Only" [107], and RVQuant methods [187]. Direct quantization of Inception-V3 (*i.e.*, without use of second-order information), results in 7.69% accuracy degradation. Using the approach proposed in [107] results in more than 2% accuracy drop, even though it uses higher bit precision. However, HAWQ results in an accuracy gap of 2% with a compression ratio of 12.04×, both of which are better than previous work [107, 187].[2]

We also compare Deep Compression [82] and the AutoML-based method of HAQ, which has been recently introduced [236]. We compare our HAWQ results with their ResNet50 quantization results, as shown in Table 6.1. HAWQ achieves higher top-1 accuracy of 75.48% with a model size of 7.96MB, whereas the AutoML-based HAQ method has a top-1 of 75.30% even with 16% larger model size of 9.22MB.

Furthermore, we apply HAWQ to quantize SqueezeNext [71] on ImageNet. We choose the wider SqueezeNext model which has a baseline accuracy of 69.38% with 2.5 million parameters (10.1MB in single precision). We are able to quantize this model to uniform 8-bit precision, with just 0.04% top-1 accuracy drop. Direct quantization of SqueezeNext (*i.e.*, without use of second-order information), results in 3.98% accuracy degradation. HAWQ results in an unprecedented 1MB model size, with only a 1.36% top-1 accuracy drop. The

[2] We should emphasize here that the work of [107] uses integer arithmetic, and it is not completely fair to compare their results with ours.

Table 5.3: Quantization results of ResNet50 on ImageNet. We show results of state-of-the-art methods [288, 33, 277, 82]. In particular, we also compare with the recent proposed AutoML approach of [236]. We achieve a higher compression ratio with higher testing accuracy compared to [236]. Also note that [288, 33, 277] use 8-bit for the first and last layers.

Method	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	32	32	77.39	1.00×	97.8
Dorefa [288]	2	2	67.10	16.00×	6.11
Dorefa [288]	3	3	69.90	10.67×	9.17
PACT [33]	2	2	72.20	16.00×	6.11
PACT [33]	3	3	75.30	10.67×	9.17
LQ-Nets [277]	3	3	74.20	10.67×	9.17
Deep Comp. [82]	3	MP	75.10	10.41×	9.36
HAQ [236]	MP	MP	75.30	10.57×	9.22
HAWQ	2 MP	4 MP	**75.48**	12.28×	**7.96**

significance of this result is that it allows deployment of the whole model on-chip or on hardware with very limited memory and power constraints.

Table 5.4: Quantization results of SqueezeNext on ImageNet. We show a case where HAWQ is used to achieve uniform quantization to 8 bits for both weights and activations, with an accuracy similar to ResNet18. We also show a case with mixed precision, where we compress SqueezeNext to a model with just 1MB size with only 1.36% accuracy degradation. Furthermore, we compare HAWQ with the direct quantization method without using Hessian ("Direct").

Method	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	32	32	69.38	1.00×	10.1
ResNet18 [188]	32	32	69.76	1.00×	44.7
HAWQ	8	8	**69.34**	4.00×	2.53
Direct	3 MP	8	65.39	9.04×	1.12
HAWQ	3 MP	8	**68.02**	9.25×	**1.09**

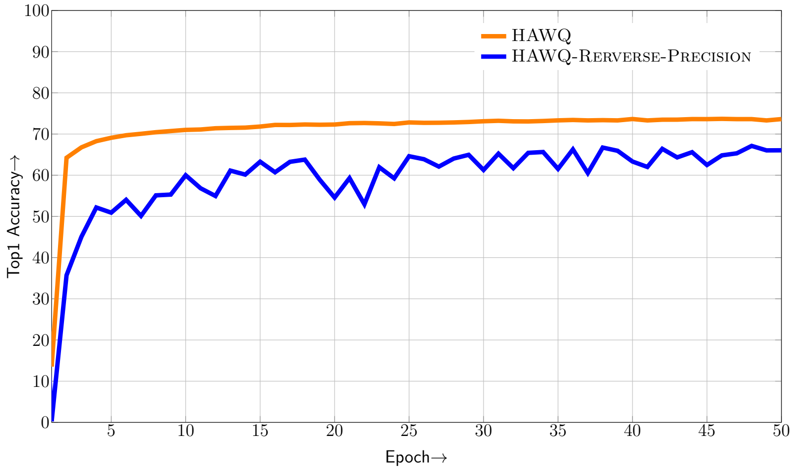


Figure 5.2: Accuracy recovery from Hessian aware mixed-precision quantization versus HAWQ-Reverse-Precision quantization. Here, we show top-1 accuracy of quantized Inception-V3 on ImageNet. HAWQ-Reverse-Precision achieves 66.72% (compression-ratio 7.2) top-1 accuracy, while our HAWQ method achieves 74.36% (compression-ratio 12.0) top-1 accuracy (7.64% better) with a higher convergence speed (30 epochs v.s. 50 epochs of HAWQ-Reverse-Precision).

5.3 Ablation Study

Hessian AWare Mixed Precision Quantization

We first discuss the ablation study for step (i), where the quantization precision is chosen based on Eq. 5.3. As discussed above, blocks with higher values of S_i are assigned higher quantization precision, and vice versa for layers with relatively lower values of S_i. For the ablation study, we reverse this order and avoid performing the block-wise fine-tuning of step (ii) so we can isolate step (i). Instead of the fine-tuning phase, we re-train the whole network at once after the quantization is performed. The results are shown in Figure 5.2, where we perform 50 epochs of fine-tuning using Inception-V3 on ImageNet. As one can see, HAWQ results in significantly better accuracy (74.26% as compared to 66.72%) than the reverse method. This is despite the fact that the latter approach only has a compression ratio of 7.2×, whereas HAWQ has a compression ratio of 12.0×.

Another interesting observation is that the convergence speed of the Hessian aware approach is significantly faster than the reverse method. Here, HAWQ converges in about 30 epochs, whereas the HAWQ-Reverse-Precision case takes 50 epochs before converging to a sub-optimal value (Figure 5.2).

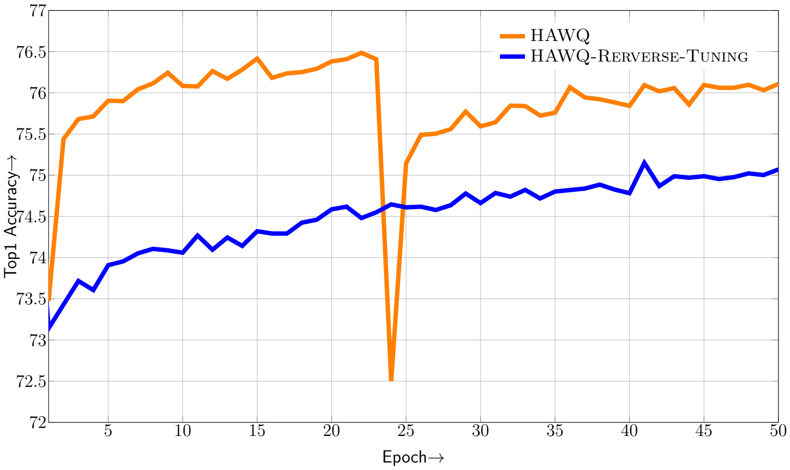


Figure 5.3: Effectiveness of Hessian aware block-wise fine-tuning. Here, HAWQ shows the quantization process based on the descending order of Ω_i for Inception-V3 with Hessian aware quantization order. HAWQ-Reverse-Tuning shows the quantization process of Inception-V3 with a reverse order. Note that HAWQ finishes the fine-tuning of this block in just 25 epochs and switches to fine-tuning another block, whereas HAWQ-Reverse-Tuning takes 50 epochs for this block, before converging to sub-optimal top-1.

Block-Wise Fine-Tuning

Here we perform the ablation study for the Hessian-based fine-tuning part of HAWQ. The block-wise fine-tuning is performed based on Ω_i (Equation 6.9) of each block. The blocks are fine-tuned based on the descending order of Ω_i. Similar to the above, we compare the quantization performance when a reverse ordering is used (*i.e.*, we use the ascending order of Ω_i and refer to this as "HAWQ-Reverse-Tuning").

We test this ablation study using Inception-V3 on ImageNet, as shown in Figure 5.3. As one can see, the fine-tuning for the HAWQ method quickly converges in just 25 epochs, allowing it to switch to fine-tuning the next block. However, "HAWQ-Reverse-Tuning" takes more than 50 epochs to converge for this block.

Chapter 6

Quantization: HAWQV2

We have introduced HAWQ which can efficiently conduct mixed-precision quantization with high accuracy. In this Chapter, we discussed two *key problems* of HAWQ:

> Is top-1 eigenvalue in HAWQ the best metric to measure second-order sensitivity?
>
> The Hessian information in HAWQ can only provide relative sensitivity, while a specific mixed-precision setting can only be manually generated.

Here we propose HAWQV2 which mathematically shows that the average Hessian trace is a better metric than the Top-1 eigenvalue. And HAWQV2 uses a Pareto frontier method that can automatically generate the best mixed-precision settings to minimize the second-order quantization perturbation.

6.1 Method

Sensitivity Metric

HAWQ uses the top Hessian eigenvalue to determine the relative sensitivity order of different layers [53]. However, a NN model contains millions of parameters, and thus millions of Hessian eigenvalues. Therefore, just measuring the top eigenvalue can be sub-optimal. As a simple example, consider two functions $F_1(x, y) = 100x^2 + y^2$ and $F_2(x, y) = 100x^2 + 99y^2$. The top Hessian eigenvalues of F_1 and F_2 are the same (i.e., 200). However, it is clear that F_2 is more sensitive than F_1 since F_2 has a much larger function value change along the y-axis. Below, we perform a theoretical analysis and show that a better metric is to compute the average Hessian trace (i.e., average of all Hessian eigenvalues) instead of just the top eigenvalue, and later in Section 6.2 we perform an empirical ablation study which supports this finding. Note that in practice the trace and top eigenvalues can be significantly different.

Assumption 1 *Assume that:*

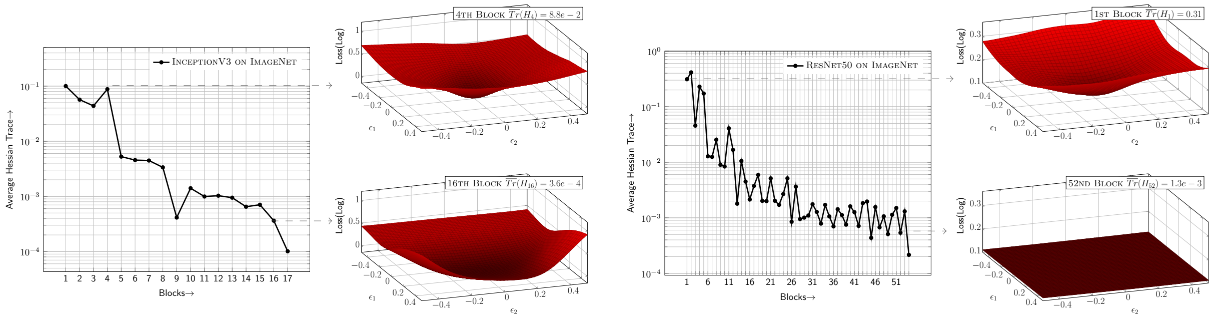


Figure 6.1: Average Hessian trace of different blocks in InceptionV3 and ResNet50 on ImageNet, along with the loss landscape of the block 4 and 16 in InceptionV3 (block 1 and 52 in ResNet50). As one can see, the average Hessian trace is significantly different for different blocks.

- *The model is twice differentiable and has converged to a local minimum such that the first and second-order optimality conditions are satisfied, i.e., the gradient is zero and the Hessian is positive semi-definite.*
- *If we denote the Hessian of the i^{th} layer as H_i, and its corresponding orthonormal eigenvectors as $v_1^i,\ v_2^i, ..., v_{n_i}^i$, then the quantization-aware fine-tuning perturbation, $\Delta W_i^* = \arg\min_{W_i^*+\Delta W_i^* \in Q(\cdot)} L(W_i^* + \Delta W_i^*)$, satisfies*

$$\Delta W_i^* = \alpha_{bit} v_1^i + \alpha_{bit} v_2^i + ... + \alpha_{bit} v_{n_i}^i. \tag{6.1}$$

  Here, n_i is the dimension of W_i, W_i^* is the converging point of i^{th} layer, and $Q(\cdot)$ is the quantization function that maps floating point values to reduced precision values. Note that α_{bit} is a constant number based on the precision setting and quantization range.[1]

- *The third-order term, $\|\frac{\nabla^3 \mathcal{L}}{\nabla W_i^3}\|\ \|\Delta W_i^*\|^3/6$ in the Taylor expansion series is small.*

Given this assumption, we establish the following lemma.

Lemma 1 *Under Assumption 1, when we quantize two layers (denoted by B_1 and B_2) with same amount of perturbation, namely $\|\Delta W_1^*\|_2^2 = \|\Delta W_2^*\|_2^2$, we will have:*

$$\mathcal{L}(W_1^* + \Delta W_1^*, W_2^*, \cdots, W_L^*) \leq \mathcal{L}(W_1^*, W_2^* + \Delta W_2^*, W_3^*, \cdots, W_L^*), \tag{6.2}$$

if

$$\frac{1}{n_1} Tr(\nabla_{W_1}^2 \mathcal{L}(W_1^*)) \leq \frac{1}{n_2} Tr(\nabla_{W_2}^2 \mathcal{L}(W_2^*)). \tag{6.3}$$

[1] We assume α_{bit} a constant for simplicity. It can be relaxed to random coefficients with the same second moment, i.e., α_{bit} can be random variables for different directions $(v_1^i,\ v_2^i, ..., v_{n_i}^i)$ but with same $\mathbf{E}[\alpha_{bit}^2]$.

Proof Denote the gradient and Hessian of the first layer as g_1 and H_1, correspondingly. By Taylor's expansion we have:

$$\mathcal{L}(W_1^* + \Delta W_1^*) = \mathcal{L}(W_1^*) + g_1^T \Delta W_1^* + \frac{1}{2}\Delta W_1^{*T} H_1 \Delta W_1^* = \mathcal{L}(W_1^*) + \frac{1}{2}\Delta W_1^{*T} H_1 \Delta W_1^*.$$

Here, we have used the fact that the gradient at the optimum point is zero and that the loss function is locally convex. Also note that $\mathcal{L}(W_1^*) = \mathcal{L}(W_2^*)$ since the model has the same loss before we quantize any layer. Based on the assumption, ΔW_1^* can be decomposed by the eigenvectors of the Hessian. As a result, we have:

$$\Delta W_1^{*T} H_1 \Delta W_1^* = \sum_{i=1}^{n_1} \alpha_{bit,1}^2 v_i^{1T} H_1 v_i^1 = \alpha_{bit,1}^2 \sum_{i=1}^{n_1} \lambda_i^1,$$

where (λ_i^1, v_i^1) is the corresponding eigenvalue and eigenvector of Hessian. Similarly, for the second layer we will have: $\Delta W_2^{*T} H_2 \Delta W_2^* = \alpha_{bit,2}^2 \sum_{i=1}^{n_2} \lambda_i^2$, where λ_i^2 is the i^{th} eigenvalue of H_2. Since $\|\Delta W_1^*\|_2 = \|\Delta W_2^*\|_2$, we have $\sqrt{n_1}\alpha_{bit,1} = \sqrt{n_2}\alpha_{bit,2}$. Therefore, we have:

$$\mathcal{L}(W_2^* + \Delta W_2^*) - \mathcal{L}(W_1^* + \Delta W_1^*) = \alpha_{bit,2}^2 n_2 (\frac{1}{n_2}\sum_{i=1}^{n_2} \lambda_i^2 - \frac{1}{n_1}\sum_{i=1}^{n_1} \lambda_i^1) \geq 0.$$

It is easy to see that the lemma holds since the sum of eigenvalues equals to the trace of the matrix. □

It should be noted that the proof still holds for cases where $||\Delta W_1^*||_2^2 \neq ||\Delta W_2^*||_2^2$. In such cases, Eq. 6.3 becomes:

$$\frac{||\Delta W_1^*||_2^2}{n_1} Tr(\nabla_{W_1}^2 \mathcal{L}(W_1^*)) \leq \frac{||\Delta W_2^*||_2^2}{n_2} Tr(\nabla_{W_2}^2 \mathcal{L}(W_2^*)), \tag{6.4}$$

indicating that $\overline{Tr}(H_i)||\Delta W_i^*||_2^2$ can be used as a measure of sensitivity.

At first, computing the Hessian trace may seem a prohibitive task, as we do not have direct access to the elements of the Hessian matrix. Furthermore, forming the Hessian matrix explicitly is not computationally feasible. However, it is possible to leverage the extensive literature in Randomized Numerical Linear Algebra (RandNLA) [164, 159] which addresses this type of problem. In particular, the seminar works of [6, 9] have proposed randomized algorithms for fast trace estimation, using so-called matrix-free methods which do not require the explicit formation of the Hessian operator. Here, we are interested in the trace of a symmetric matrix $H \in R^{d \times d}$. Then, given a random vector $z \in R^d$ whose component is i.i.d. sampled Gaussian distribution ($N(0,1)$) (or Rademacher distribution), we have:

$$Tr(H) = Tr(HI) = Tr(H\,\mathbb{E}[zz^T]) = \mathbb{E}[Tr(Hzz^T)] = \mathbb{E}[z^T H z], \tag{6.5}$$

where I is the identity matrix. Based on this, the Hutchinson algorithm [6] can be used to estimate the Hessian trace:

$$Tr(H) \approx \frac{1}{m}\sum_{i=1}^{m} z_i^T H z_i = Tr_{Est}(H). \tag{6.6}$$

We show empirically that this algorithm has good convergence properties, resulting in trace computation being orders of magnitude faster than training the network itself.

We have incorporated the above approach and computed the average Hessian trace for different layers of InceptionV3 and ResNet50, as shown in Figure 6.1. As one can see, there is a significant difference between the average Hessian traces for different layers. To better illustrate this, we have also plotted the loss landscape of InceptionV3 and ResNet50 by perturbing the pre-trained model along the first and second eigenvectors of the Hessian for each layer. It is clear that different layers have significantly different "sharpness."

Mixed Precision Activation

The above analysis is not restricted to weights, and in fact, it can be extended to mixed-precision activation quantization. In Section 6.2, we will show that this is particularly useful for tasks such as object detection. The theoretical results remain the same, except that the Hessian here is with respect to activations instead of model parameters. In the matrix-free Hutchinson algorithm, we need the result of the following Hessian-vector product to compute the Hessian trace:

$$z^T H_{a_j} z = z^T \left(\nabla^2_{a_j} \frac{1}{N} \sum_{i=1}^{N} f(x_i, y_i, \theta) \right) z, \tag{6.7}$$

where a_j is the activations of the j^{th} layer. Here, $H_{a_j} \in \mathbb{R}^{(\sum_{i=1}^{N} |a_j(x_i)|) \times (\sum_{i=1}^{N} |a_j(x_i)|)}$, where $|a_j(x_i)|$ is the size of the activation of the j^{th} layer for i^{th} input. This is because a_j is a concatenation of $a_j(x_i), \forall i$. See Figure 6.2 for illustration of the matrix H_{a_j} and its shape. Not only is it prohibitive to compute the Hessian matrix, but the Hessian-vector product is also infeasible since even generating the random vectors $z \in \mathbb{R}^{\sum_{i=1}^{N} |a_j(x_i)|}$ is prohibitive, let alone computing its product with H_{a_j}. Furthermore, note that a_j depends on x_i, and for many tasks such as object detection on Microsoft COCO, x_i does not have a fixed size. As a result, the activation size of each layer depends on the input data and is not fixed, which further complicates computing Hessian trace w.r.t. activations.

However, H_{a_j}, has a very interesting structure. As illustrated in Figure 6.2, it is block diagonal, with $H_{a_j(x_i)}$ being the blocks, where $H_{a_j(x_i)} = \nabla^2_{a_j(x_i)} \frac{1}{N} f(x_i, y_i, \theta)$. This is due to the fact that different inputs are independent of each other. As a result, we can compute the Hessian trace for the layer's activations *for one input* at a time, and then average the resulting Hessian traces of each block diagonal part, i.e.,

$$z^T H_{a_j} z = \frac{1}{N} \sum_{i=1}^{N} z_i^T H_{a_j(x_i)} z_i, \tag{6.8}$$

where z_i is the corresponding components of z w.r.t. the i^{th} input, i.e., x_i. We note that usually this trace computation converges very fast, and it is not necessary to average over the entire dataset. See Figure 6.6 for more details.

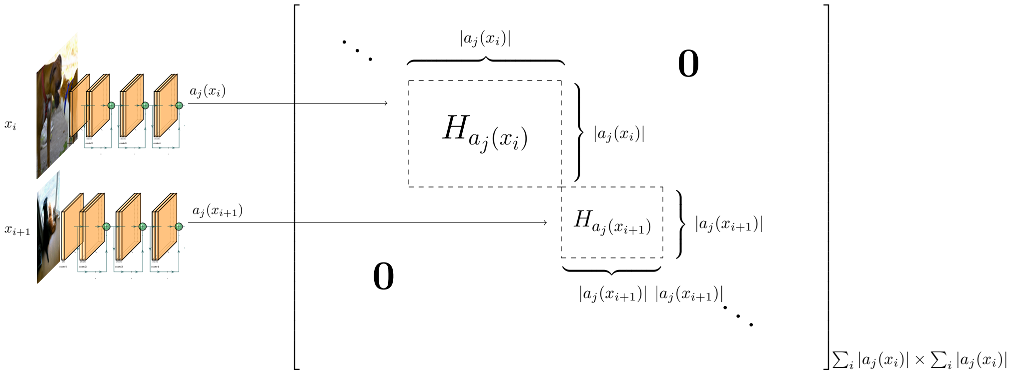


Figure 6.2: Illustration of the structure of Hessian w.r.t to activations (H_{a_j}). It is evident that different sized inputs x_i will produce different sized blocks $H_{a_j(x_i)}$ which appear on the diagonal of H_{a_j}.

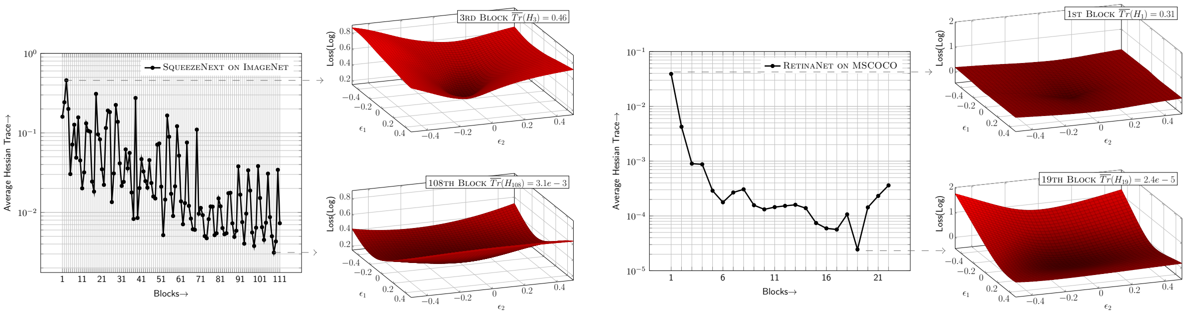


Figure 6.3: Average Hessian trace of different blocks in SqueezeNext and RetinaNet, along with the loss landscape of block 3 and 108 in SqueezeNext, and block 1 and 19 in RetinaNet. It should be noted that block 1 to block 17 in RetinaNet are the ResNet50 backbone, block 18 to block 20 are FPN, and block 21 and block 22 are the classification and regression heads, respectively. As one can see, the average Hessian trace is significantly different for different blocks. We assign higher bits for blocks with larger average Hessian trace, and fewer bits for blocks with smaller average Hessian trace. For reference, in Figure 6.1 we showed a similar plot but for InceptionV3 and ResNet50.

Automatic Bit Selection

An important limitation of relative sensitivity analysis is that it does not provide the specific bit precision setting for different layers. This is true even if we use the average Hessian trace, instead of the top Hessian eigenvalue. For example, we show the average Hessian trace for different blocks of InceptionV3 in Figure 6.1. We can clearly see that block 1 to block 4 have the largest average Hessian trace, and block 9 or block 16 have orders of magnitude smaller average Hessian trace. However, although we know the first four blocks are more sensitive,

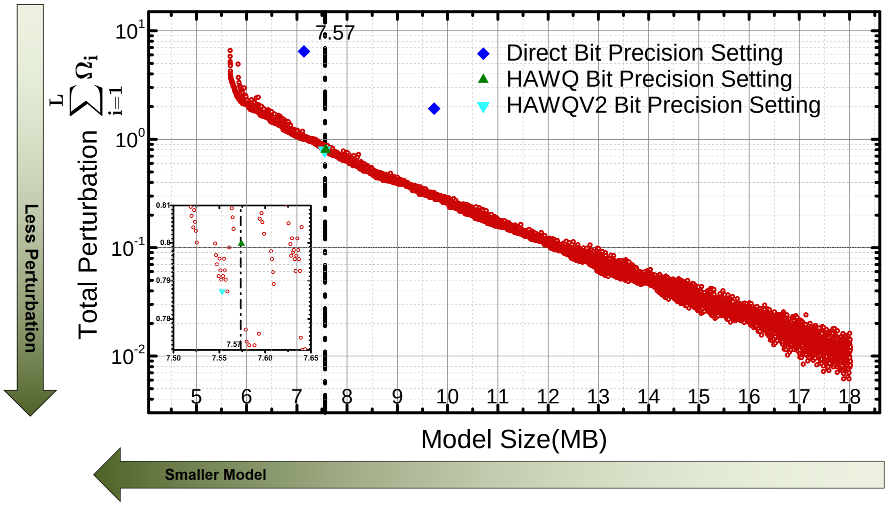


Figure 6.4: Pareto Frontier: The trade-off between model size and the sum of Ω metric (of Eqn. (6.9)) in InceptionV3. Here, L is the number of blocks in the model, and each point in the figure stands for a specific bit precision setting. We show the precision setting used in Direct quantization and HAWQ. To achieve fair comparison, we set a constraint on HAWQV2 to have the same model size as HAWQ.

we still cannot determine whether to assign 8-bit or 4-bit for these layers.

Denote by $\mathcal{B}$ the set of all admissible bit precision settings that satisfy the relative sensitivity analysis based on the average Hessian trace discussed above. Compared to the original exponential search space, applying the sensitivity constraint makes the cardinality (size) of $\mathcal{B}$ significantly smaller. As an example, the original mixed-precision search space for ResNet50 is $4^{50} \approx 1.3 \times 10^{30}$ if bit-precisions are chosen among $\{1, 2, 4, 8\}$. Using the Hessian-trace sensitivity constraint significantly reduces this search space to $|\mathcal{B}| = 2.3 \times 10^4$. However, this search space is still prohibitively large, especially for deeper models such as ResNet152. In the HAWQ paper [53], we manually chose the bit precision among this reduced search space, but this manual selection is undesirable.

We found that this problem can be efficiently addressed using a Pareto frontier approach. The main idea is to sort each candidate bit-precision setting in $\mathcal{B}$ based on the total second-order perturbation that they cause, according to the following metric:

$$\Omega = \sum_{i=1}^{L} \Omega_i = \sum_{i=1}^{L} \overline{Tr}(H_i) \cdot \|Q(W_i) - W_i\|_2^2, \tag{6.9}$$

where i refers to the i^{th} layer, L is the number of layers in the model, $\overline{Tr}(H_i)$ is the average Hessian trace, and $\|Q(W_i) - W_i\|_2$ is the L_2 norm of quantization perturbation. The intuition is that a bit precision setting with minimal second-order perturbation to the model should lead to good generalization after quantization-aware fine-tuning. Given a target model size, we sort the elements of $\mathcal{B}$ based on their Ω value, and we choose the bit precision setting with minimal Ω. While this approach cannot theoretically guarantee the best possible performance, we have found that in practice it can generate bit precision settings that exceed current state-of-the-art results with a small time cost (as shown in Section 6.2). An important benefit of this approach is that it removes the manual precision selection process used in our previous work on HAWQ [53].

We show the process for choosing the exact bit precision setting of InceptionV3 in Figure 6.4. Each red dot denotes a specific bit precision setting for different blocks of InceptionV3. For each target model size, HAWQV2 chooses the bit precision setting with minimal Ω value. With green triangles, we have also denoted the bit precision setting that was manually selected in the HAWQ paper [53]. The automatic bit precision setting of HAWQV2 exceeds the accuracy of HAWQ, as will be discussed in the next section.

6.2 Experiments

Hutchinson's Method for Trace Estimation

In Figure 6.5, we show the convergence plot for Hutchinson's algorithm as we increase the number of iterations used for the Hessian trace estimation. It can be clearly seen that the trace converges rapidly as we increase the number of data points over 512, over which the sub-sampled Hessian is computed. We can see that 50 Hutchinson iterations are sufficient to achieve an accurate approximation with low variance. Based on the convergence analysis, we are able to calculate all the average Hessian traces, shown in Figure 6.1, corresponding to 54 blocks in a ResNet50 model, within 30 minutes (33s per block on average) using 4 GPUs. The Hutchinson algorithm, in addition to the automatic bit precision selection, makes HAWQV2 a significantly faster algorithm than previous searching-based algorithms [236].

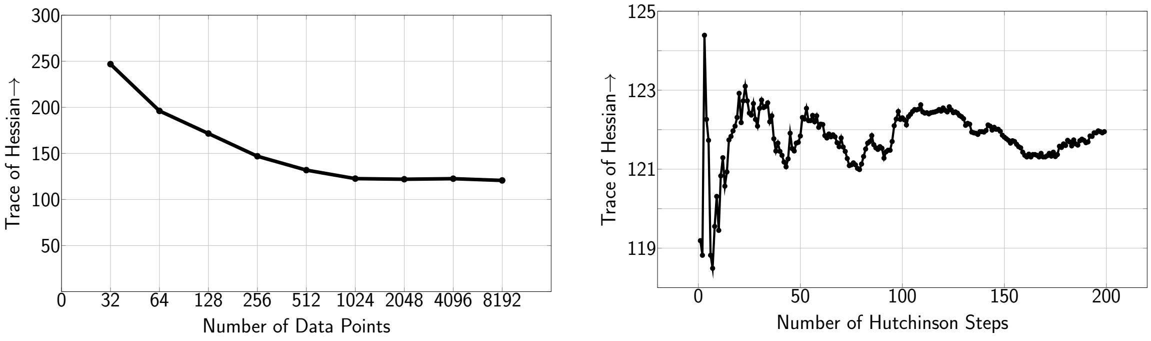


Figure 6.5: Relationship between the convergence of Hutchinson and the number of data points (Left) as well as the number of steps (Right) used for trace estimation on block 21 in ResNet50.

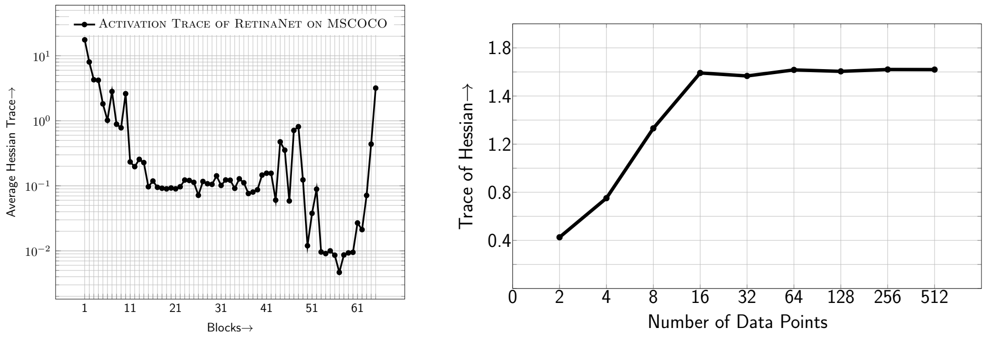


Figure 6.6: (Left) Average Hessian trace w.r.t. activations in RetinaNet. As we can see, the average Hessian trace varies significantly across activations of different blocks. We use this information to perform mixed-precision activation quantization as discussed in Section 6.1. (Right) we show the relationship between the convergence of Hutchinson and the number of data points used for trace estimation on block 5 in RetinaNet. We used 128 data points with 50 Hutchinson steps to plot the left figure.

ImageNet

As shown in Table 6.1, we first apply HAWQV2 on ResNet50 [87], and compare HAWQV2 with other popular quantization methods [288, 33, 277, 82, 236, 53]. It should be noted that [288, 33, 277, 82] followed traditional quantization rules which set the precision of the first and last layers to 8-bit, and quantized other layers to an identical precision. Both [236, 53] are mixed-precision quantization methods. Also, [236] uses reinforcement learning methods to search for a good precision setting, while HAWQ uses second-order information to guide the precision selection as well as the block-wise fine-tuning. HAWQ achieves 75.48% with a 7.96MB model size. Keeping model size the same, HAWQV2 can achieve 75.92% accuracy without any heuristic knowledge or manual efforts.

We then show results on InceptionV3 [221]. Direct quantization of InceptionV3 (i.e., without use of second-order information), results in 7.69% accuracy degradation. Using the approach proposed in [107] results in more than 2% accuracy drop, even though it uses higher bit precision. HAWQ [53] results in a 2% accuracy gap with a compression ratio of 12.04×. HAWQV2 **automatically** generates the exact precision setting for the whole network, and still achieves better accuracy than the manual method of HAWQ.

We also apply HAWQV2 to quantize deep and highly compact models such as SqueezeNext. We choose the wider SqueezeNext model which has a baseline accuracy of 69.38% with 2.5 million parameters (10.1MB in single precision). We can see from Table 6.1 that direct quantization of SqueezeNext (i.e., without use of second-order information), results in 3.98% accuracy degradation. HAWQ results in a 1MB model size, with a 1.36% top-1 accuracy drop. By applying HAWQV2 on SqueezeNext, we can achieve a 68.68% accuracy with an unprecedented model size of 1.07MB.

Table 6.1: Quantization results on ImageNet. We abbreviate quantization bits used for weights as "w-bits," quantization bits used for activations as "a-bits," top-1 testing accuracy as "Top-1," and weight compression ratio as "W-Comp." Furthermore, we compare HAWQV2 with the direct quantization method of [53] ("Direct") and other state-of-the-art quantization methods. Here "MP" refers to mixed-precision quantization, and we show the lowest bit-precision used in a mixed-precision setting. Compared to [107, 187], we achieve a higher compression ratio with higher testing accuracy.

(a) ResNet50 on ImageNet.

Method	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	32	32	77.39	1.00×	97.8
Dorefa [288]	2	2	67.10	16.00×	6.11
Dorefa [288]	3	3	69.90	10.67×	9.17
PACT [33]	2	2	72.20	16.00×	6.11
PACT [33]	3	3	75.30	10.67×	9.17
LQ-Nets [277]	3	3	74.20	10.67×	9.17
Deep Comp. [82]	3	MP	75.10	10.41×	9.36
HAQ [236]	MP	MP	75.30	10.57×	9.22
HAWQ [53]	2 MP	4 MP	75.48	12.28×	7.96
HAWQV2	2 MP	4 MP	**75.92**	12.24×	7.99

(b) InceptionV3 on ImageNet

Method	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	32	32	77.45	1.00×	91.2
IntOnly [107]	8	8	75.40	4.00×	22.8
RVQ [187]	3 MP	3 MP	74.14	10.67×	8.55
Direct [53]	2 MP	4 MP	69.76	15.88×	5.74
HAWQ [53]	2 MP	4 MP	75.52	12.04×	7.57
HAWQV2	2 MP	4 MP	**75.98**	12.04×	7.57

(c) SqueezeNext on ImageNet

Method	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	32	32	69.38	1.00×	10.1
Direct [53]	3 MP	8	65.39	9.04×	1.12
HAWQ [53]	3 MP	8	68.02	9.26×	1.09
HAWQV2	3 MP	8	**68.68**	9.40×	1.07

Microsoft COCO

In order to show the generalization capability of HAWQV2, we also test object detection task Microsoft COCO 2017 [145]. RetinaNet [144] is a single stage detector that can achieve state-of-the-art mAP with a very simple network architecture. As shown in Table 6.2, we use the pretrained RetinaNet with ResNet50 backbone as our baseline model, which can achieve 35.6 mAP with 145MB model size. We first show the result of direct quantization where no Hessian information is used. Even with quantization-aware fine-tuning and channel-wise quantization of weights, directly quantizing weights and activations in RetinaNet to 4-bit causes a significant 4.1 mAP degradation. FQN [130] is a recently proposed quantization method that reduces this accuracy gap to 3.1 mAP with the same compression ratio as the Direct method. We implement HAWQ to perform mixed-precision quantization, which results in 33.5 mAP. As a comparison, using HAWQV2 achieves a state-of-the-art performance of 34.1 mAP, which is 0.6 mAP higher than [53] and 1.6 mAP higher than [130] with even smaller model size.

It should also be noted that we found the activation quantization to be sensitive for object detection models. For instance, increasing activation quantization bit precision to 6-bit can result in a 34.8 mAP. One might argue that using 6-bit for activation leads to higher activation memory, which can be a problem for extreme cases such as on micro-controllers where

Table 6.2: Quantization results of RetinaNet-ResNet50 on Microsoft COCO 2017. We show results of direct quantization, mixed-precision quantization [53], as well as a state-of-the-art quantization method for object detection [130]. HAWQV2 can outperform previous results by a large margin. We also show that HAWQV2 with mixed-precision activations can achieve even better mAP, with a slightly lower activation compression ratio.

Method	w-bits	a-bits	mAP	W-Comp	A-Comp	Size(MB)
Baseline	32	32	35.6	1.00×	1.00×	145
Direct	4	4	31.5	8.00×	8.00×	18.13
FQN [130]	4	4	32.5	8.00×	8.00×	18.13
HAWQ	3 MP	4	33.5	8.10×	8.00×	17.90
HAWQV2	3 MP	4	**34.1**	8.10×	8.00×	17.90
HAWQV2	3 MP	4 MP	**34.4**	8.10×	7.62×	17.90
HAWQV2	3 MP	6	**34.8**	8.10×	5.33×	17.90

every bit counts. For these situations, we can use mixed-precision activation as discussed in Section 6.1, with the same automatic bit-precision selection method using Pareto optimal curve. As can be seen in Table 6.2, mixed-precision activation quantization can achieve a very good trade-off between accuracy and compression. With only a marginal change to activation compression ratio, it can achieve 34.4 mAP, which significantly outperforms uniform 4-bit activation, and is even close to a uniform 6-bit activation quantization.

Ablation Study

Here, we perform three ablation studies. First, we show why it is important to choose the bit-precision setting that results in the smallest model perturbation as done in Figure 6.4. The results are shown in Table 6.3(a), where the ablation row uses a bit precision setting with large model perturbation. As one can see, the HAWQV2 approach achieves more than 1% higher accuracy with a smaller model size.

Second, we measure the importance of using the Hessian trace to weigh the sensitivity $\Omega_i = \overline{Tr}(H_i)\|\Delta W_i\|_2^2$ in Eq. 6.9. The results are shown in Table 6.3(b), where we compare with using only parameter perturbation as the sensitivity metric $\Omega_i = \|\Delta W_i\|_2^2$. As we can see, HAWQV2 with the average Hessian trace is 0.85% better than L2-Sensitivity, while achieving a smaller model size.

Finally, we also compare HAWQV2 with a sensitivity that is weighted by Top-1 Hessian eigenvalue. The results are shown in Table 6.4. HAWQ uses a heuristic metric $S_i = \lambda_i / n_i$ to select mixed-precision bitwidths, where λ_i is the top eigenvalue of the i^{th} layer and n_i is the parameter size. However, as we can see, only using λ_i as a sensitivity metric in HAWQ (represented as HAWQ-Ablation) can lead to significant accuracy degradation. In contrast, the

Table 6.3: The effectiveness of metric in Eq. 6.9. Experiments are for SqueezeNext on ImageNet.

(a) Accuracy v.s. Total Perturbation

Method	w-bits	a-bits	Top-1	Size(MB)	Perturb.
Baseline	32	32	69.38	10.1	0
Large Perturbation (Ablation)	3 MP	8	67.46	1.09	3.2
Min Perturbation (HAWQV2)	3 MP	8	**68.68**	**1.07**	1.1

(b) $\overline{Tr}(H_i)\|\Delta W_i\|_2^2$ v.s. $\|\Delta W_i\|_2^2$

Method	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	32	32	69.38	1.00×	10.1
L2-Sensitivity (Ablation)	3 MP	8	67.83	9.18×	1.10
Trace-Sensitivity (HAWQV2)	3 MP	8	**68.68**	**9.40×**	**1.07**

theoretically derived metric, average Hessian trace, can achieve better results than HAWQ with the same compression ratio. To achieve a fair comparison, we constrain HAWQV2 to assign the same quantization precision for layers in the same block of InceptionV3, as what HAWQ did, and we make their activation bit settings to be the same (referred to as HAWQV2-blockwise).

Table 6.4: The effectiveness of average Hessian trace. The experiments are for InceptionV3 on ImageNet. We abbreviate quantization bits used for weights as "w-bits," quantization bits used for activations as "a-bits," top-1 testing accuracy as "Top-1," and weight compression ratio as "W-Comp." Here "MP" refers to mixed-precision quantization, and we show the lowest bit-precision used in a mixed-precision setting. Blockwise means to assign the same bitwidth for layers within the same block. Compared to HAWQ and HAWQ-Ablation, HAWQV2 can achieve a higher compression ratio with higher accuracy.

Method	Metric	w-bits	a-bits	Top-1	W-Comp	Size(MB)
Baseline	NA	32	32	77.45	1.00×	91.2
HAWQ	λ_i/n_i	2 MP	4 MP	75.52	12.04×	**7.57**
HAWQ-Ablation	λ_i	2 MP	4 MP	73.35	10.56×	8.65
HAWQV2-blockwise	$\overline{Tr}(H_i)$	2 MP	4 MP	**75.73**	12.04×	**7.57**

Chapter 7

Quantization: Q-BERT

In HAWQ and HAWQV2 we explored mixed-precision quantization for computer vision tasks. In this Chapter, we try to answer the *key questions*:

> How will the accuracy be affected if we directly apply standard quantization to NLP models? What necessary adaptation should be made to alleviate the accuracy degradation?

Here we show our method Q-BERT which achieves significant compression ratios while maintaining accuracy.

7.1 Introduction to NLP tasks and Compression

We apply QBERT to Sentiment Classification, Natural Language Inference, Named Entity Recognition and Machine Reading Comprehension tasks. For Sentiment Classification, we evaluate on Stanford Sentiment Treebank (SST-2) [215]. For Named Entity Recognition, we use CoNLL-2003 English benchmark dataset for NER (CoNLL-03) [206]. For Natural Language Inference, we test on Multi-Genre Natural Language Inference (MNLI) [248]. For Machine Reading Comprehension, we evaluate the Stanford Question Answering Dataset (SQuAD) [194]. More specifically, SST-2 is a movie review dataset with binary annotations, where the binary label indicates positive and negative reviews. MNLI is a multi-genre NLI task for predicting whether a given premise-hypothesis pair is entailment, contradiction, or neural. Its test and development datasets are further divided into in-domain (MNLI-m) and cross-domain (MNLI-mm) splits to evaluate the generality of tested models. CoNLL-03 is a newswire article dataset for predicting the exact span of the annotated four entity types: person, location, organization, and miscellaneous. SQuAD is a task to answer the question by extracting the relevant span from the context, where a paragraph of context and a question is provided for each sample.

Notable examples of NLP compression work are LSTM and GRU-based models for machine translation and language model [255, 239]. Transformer-based architectures have become de-facto models used for a range of Natural Language Processing tasks. In particular, the BERT-based models achieved significant accuracy gain for GLUE tasks, CoNLL-03 and SQuAD. However, BERT-based models have a prohibitive memory footprint and latency, which is due to the incorporation of very large fully connected layers and attention matrices in Transformers [234, 44, 266, 154, 193]. As a result, deploying BERT-based models in resource-constrained environments has become a challenging task. Pilot work addressing this are [170, 16]. From a different angle, [229, 161] have probed the architectural change of the self-attention layer to make the Transformer lightweight. There have also been attempts to use distillation to reduce large pre-trained Transformer models in [226, 218]. However, significant accuracy loss is observed even for a relatively small compression ratio of 4×. Here we show that this compression ratio could be increased up to 13×, including 4× reduction of embedding layer, with much smaller performance degradation.

7.2 Method

In this section, we introduce our proposed BERT quantization methods, including the mixed precision quantization based on Hessian information, as well as techniques used for the group-wise quantizing scheme.

As in [44], a fine-tuned BERTBASE model consists of three parts: embedding; Transformer based encoder layers; and output layer. Specifically, assuming $x \in X$ is the input word (sentence) and $y \in Y$ is the corresponding label, we have the loss function L defined as:

$$L(\theta) = \sum_{(x_i, y_i)} \text{CE}(\text{softmax}(W_c(W_n(...W_1(W_e(x_i))))), y_i),$$

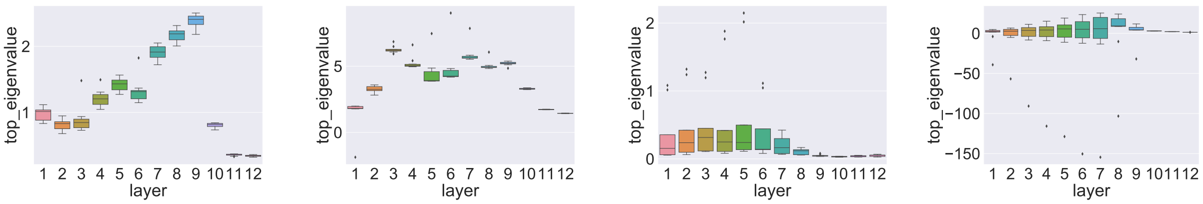
where CE is the cross entropy function (or other appropriate loss functions), θ is a combination of W_e, W_1, $W_2, ..., W_n$ and W_c. Here, W_e is the embedding table, W_1, $W_2, ...,$ W_n are the encoder layers, and W_c is the output/classifier layer[1].

The size of parameters in BERTBASE model is 91MB for embedding, 325MB for encoder and 0.01MB for output. We do not quantize the output layer due to its negligible size, and focus on quantizing both the embedding and encoder layers. As will be discussed in Section 7.3, we find that the embedding layer is much more sensitive to quantization than the encoder layers. As a result, we quantize embedding and encoder parameters in different ways. The quantization schemes we used are explained in detail in the following sections.

The above Hessian-based approach was used in [53], where top eigenvalues are computed and averaged for different training data. More aggressive quantization is performed for layers that have smaller top eigenvalue, which corresponds to a flatter loss landscape as in Figure 7.2. However, we find that assigning bits based only on the average top eigenvalues is infeasible for many NLP tasks.

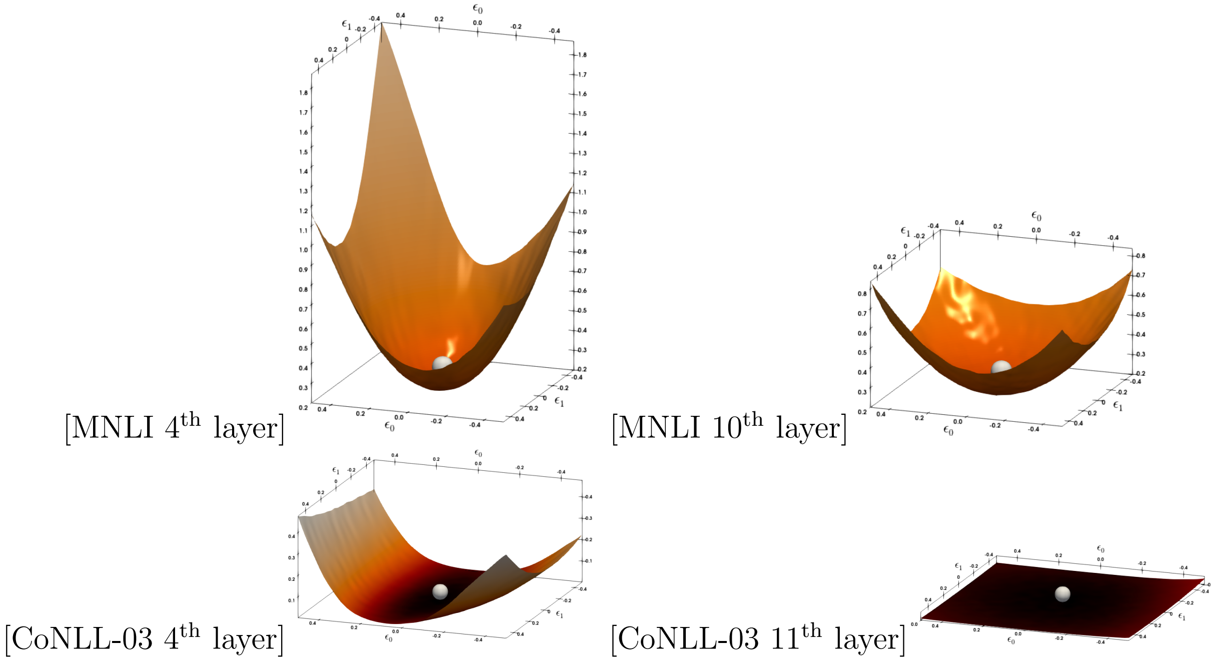
As shown in Figure 7.1, top eigenvalues of Hessian for some layers exhibits very high variance with respect to a different portion of the input dataset. As an example, the variance

[1] Here, we use W_* for both function and its corresponding parameters without confusion.



(a) SST-2 (b) MNLI (c) CoNLL-03 (d) SQuAD

Figure 7.1: From (a) to (d): Top eigenvalue distributions for different encoder layers for SST-2, MNLI, CoNNL-03, SQuAD, respectively. For each task, 10% of the data is used to compute the top eigenvalue, and we perform 10 individual runs to plot the top eigenvalue distribution. It can be seen that layers in the middle have higher mean values, and they also tend to have a larger variance than the others. The last three layers have the smallest variance as well as mean values among all layers.



[MNLI 4th layer] [MNLI 10th layer]

[CoNLL-03 4th layer] [CoNLL-03 11th layer]

Figure 7.2: The loss landscape for different layers in MNLI and CoNNL-03 is illustrated by perturbing the parameters along the first two dominant eigenvectors of the Hessian. The silver sphere shows the point in the parameter space to which the BERT model has converged. Layers that exhibit flatter curvature can be quantized to lower bit precision.

of the 7th layer for SQuAD stays larger than 61.6 while the mean of that layer is around 1.0, even though each data point corresponds to 10% of the entire dataset (which is 9K samples). To address this, we use the following metric instead of just using mean value,

$$\Omega_i \triangleq |\text{mean}(\lambda_i)| + \text{std}(\lambda_i), \tag{7.1}$$

where λ_i is the distribution of the top eigenvalues of H_i, calculated with 10% of the training dataset. [2]

After Ω_i is computed, we sort them in descending order, and we use it as a metric to relatively determine the quantization precision. We then perform quantization-aware fine-tuning based on the selected precision setting.

An important technical point that we need to emphasize is that our method expects that before performing quantization the trained model has converged to a local minimum. That is, the practitioners who trained BERT and performed its fine-tuning for downstream tasks should have chosen the hyper-parameters and number of iterations such that a local minimum has been reached. The necessary optimality conditions are zero gradients, and positive curvature (i.e., positive Hessian eigenvalue). In our analysis, we observed that for the three tasks of MNLI, CoNLL-03, and SST-2 the top Hessian eigenvalue is indeed positive. However, we find that the BERT model fine-tuned for SQuAD has actually *not* converged to a local minimum, as evident in the Hessian eigenvalues shown in Figure 7.1(d), where we observe very large negative eigenvalues. Directly visualizing the loss landscape also shows this very clearly as in Figure 7.3. Because of this, our expectation is that performing quantization on SQuAD would lead to higher performance degradation as compared to other tasks, and this is indeed the case as will be discussed next.

Group-wise Quantization

Assume that the input sequence has n words and each word has a d-dim embedding vector ($d = 768$ for BERTBASE), i.e., $x = (x(1), \ldots, x(n))^T \in \mathbb{R}^{n\times d}$. In the Transformer encoder, each self-attention head has 4 dense matrices, i.e., $W_k, W_q, W_v, W_o \in \mathbb{R}^{\frac{d}{N_h}\times d}$, where N_h is the number of attention heads. Here W_k, W_q, W_v, and W_o stand for key, query, value, and output weight matrix. Each self-attention head computes the weighted sum as

$$\text{Att}(x, x) = W_o \sum_{i=1}^{n} \text{softmax}\left(\frac{x(i)^T W_q^T W_k x(i)}{\sqrt{d}}\right) W_v x(i).$$

Then, multi-head self-attention (MHSA) will concatenate all of them as the final output, i.e., $(\text{Att}_1(x, x), \text{Att}_2(x, x), ..., \text{Att}_{N_h}(x, x))$.

Directly quantizing every 4 matrices in MHSA as an entirety with the same quantization range can significantly degrade the accuracy, since there are more than 2M parameters in

[2] Without confusion, we use λ_i for both single top eigenvalue and its distribution with respect to 10% of the data.

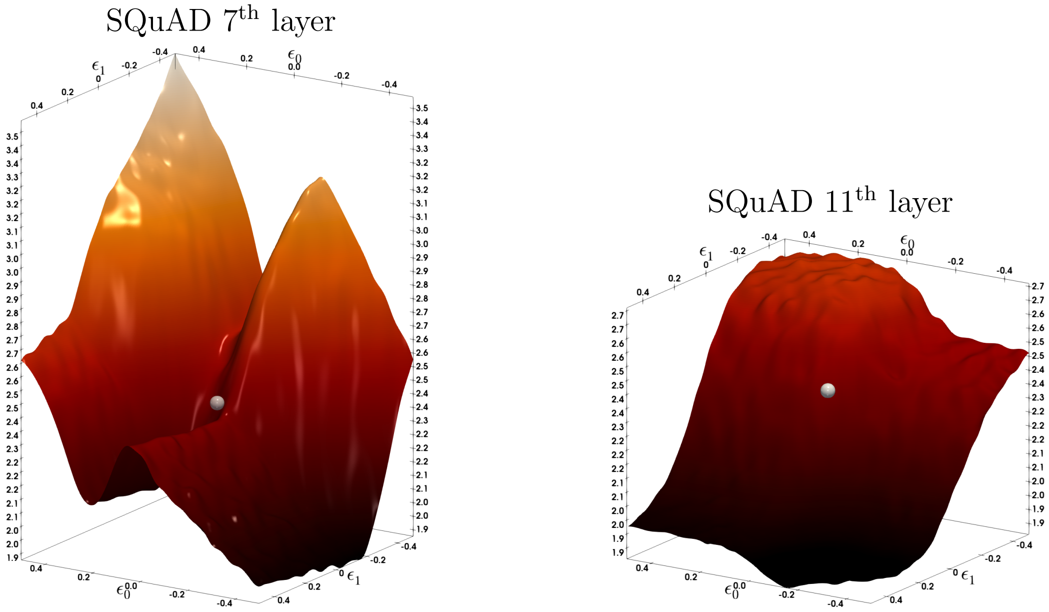


Figure 7.3: The loss landscape for different layers in SQuAD is illustrated by perturbing the parameters along the first two dominant eigenvectors of the Hessian. The silver sphere shows the point in the parameter space to which the BERT model has converged. Note that the stopping point of SQuAD has negative eigenvalues for both layers. This could be the reason we observed a relatively larger performance drop in SQuAD after quantization; see Table 7.2.

total, which corresponds to $4 \times 12 \times 64 = 3072$ output neurons, and the weights corresponding to each neuron may lie in different ranges of real numbers. Channel-wise quantization can be used to alleviate this problem in convolutional neural networks, where each convolutional kernel can be treated as a single output channel and have its own quantization range. However, this cannot be directly applied for dense matrices, since each dense matrix itself is a single kernel. Therefore, we propose group-wise quantization for attention-based models. We treat the individual matrix W with respect to each head in one dense matrix of MHSA as a group so there will be 12 groups. Furthermore, in each group, we bucket sequential output neurons together as sub-groups, e.g., each 6 output neurons as one sub-group so there are $12 \times \frac{64}{6} = 128$ sub-group in total (the hidden dim in each head of BERTBASE is $\frac{768}{12} = 64$). Each sub-group can have its own quantization range. An illustration is shown in Fig. 7.4 for W_v, where we concatenate N_h value matrix W_v to be a 3-d tensor. For layer-wise quantization, the entire 3-d tensor will be quantized into the same range of discrete numbers. A special case of group-wise quantization is that we treat each dense matrix as a group, and every matrix can have its own quantization range. And a more general case is that we partition each dense matrix with respect to the output neuron, and we bucket every continuous $\frac{d}{2N_h}$ output neuron as a group. The effect of finer group-wise quantization is further investigated in Section 7.3.

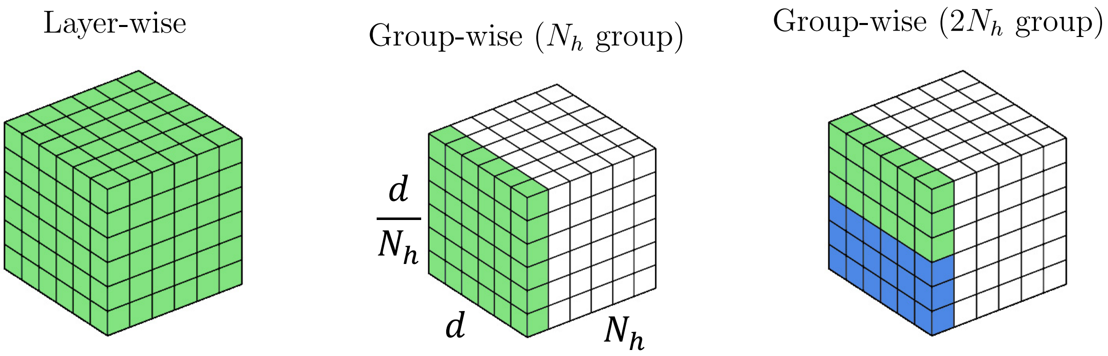


Figure 7.4: The overview of Group-wise Quantization Method. We illustrate this with value matrices of a multi-head self-attention layer. Here N_h(number of heads) value matrices W_v are concatenated together, which results in a 3-d tensor. The same color denotes the same group with a shared quantization range. As shown in (a), for layer-wise quantization, the entire 3-d tensor will be quantized from a universal quantization range into discrete unsigned integers. A special case of group-wise quantization in (b) is that we treat each dense matrix as a group, and every matrix can have its own quantization range. We show a more general case in (c), where we partition each dense matrix w.r.t output neuron and bucket every continuous $\frac{d}{2N_h}$ output neurons as a group.

7.3 Experiments

We present results of Q-BERT on the development set of the four tasks of SST-2, MNLI, CoNLL-03, and SQuAD, as summarized in Table 7.1 and 7.2. Direct quantization (DirectQ), i.e., quantization without mixed-precision and group-wise quantization is used as a baseline. As one can see, Q-BERT performs significantly better compared to the DirectQ method across all the four tasks in each bit setting. The gap becomes more obvious for ultra-low bit settings. As an example, in a 4-bit setting, Direct quantization (DirectQ) of SQuAD results in 11.5% performance degradation as compared to BERTBASE. However, for the same 4-bits setting, Q-BERT only exhibits 0.5% performance degradation. Moreover, under the 3-bit setting, the gap between Q-BERT and DirectQ increases even further to 9.68-27.83% for various tasks.

In order to push further the precision setting to lower bits, we investigate the mixed-precision Q-BERT (Q-BERT MP). As can be seen, Q-BERT with the uniform 2-bit setting has very poor performance across all four tasks, though the memory is reduced by 20% against the 3-bit setting. The reason behind this is the discrepancy that not all the layers have the same sensitivity to quantization as evident from loss landscape visualizations. Intuitively, for more sensitive layers, higher bit precision needs to be set, while for layers that are less sensitive, a 2-bit setting is already sufficient. To set mixed precision to each encoder layer of BERTBASE, we measure the sensitivity based on Eq. 7.1, which captures both mean and

Table 7.1: Quantization results for BERTBASE on Natural Language Understanding tasks. Results are obtained with 128 groups in each layer. We abbreviate quantization bits used for weights as "w-bits", embedding as "e-bits", model size in MB as "Size", and model size without embedding layer in MB as "Size-w/o-e". For simplicity and efficacy, all the models except for Baseline are using 8-bits activation. Furthermore, we compare Q-BERT with the direct quantization method ("DirectQ") without using mixed precision or group-wise quantization. Here "MP" refers to mixed-precision quantization.

(a) SST-2

Method	w-bits	e-bits	Acc	Size	Size-w/o-e
Baseline	32	32	93.00	415.4	324.5
Q-BERT	8	8	92.88	103.9	81.2
DirectQ	4	8	85.67	63.4	40.6
Q-BERT	4	8	**92.66**	63.4	40.6
DirectQ	3	8	82.86	53.2	30.5
Q-BERT	3	8	**92.54**	53.2	30.5
Q-BERT MP	2/4 MP	8	**92.55**	53.2	30.5
DirectQ	2	8	80.62	43.1	20.4
Q-BERT	2	8	**84.63**	43.1	20.4
Q-BERT MP	2/3 MP	8	**92.08**	**48.1**	**25.4**

(b) MNLI

Method	w-bits	e-bits	Acc-m	Acc-mm	Size	Size w/o-e
Baseline	32	32	84.00	84.40	415.4	324.5
Q-BERT	8	8	83.91	83.83	103.9	81.2
DirectQ	4	8	76.69	77.00	63.4	40.6
Q-BERT	4	8	**83.89**	**84.17**	63.4	40.6
DirectQ	3	8	70.27	70.89	53.2	30.5
Q-BERT	3	8	**83.41**	**83.83**	53.2	30.5
Q-BERT MP	2/4 MP	8	**83.51**	**83.55**	53.2	30.5
DirectQ	2	8	53.29	53.32	43.1	20.4
Q-BERT	2	8	**76.56**	**77.02**	43.1	20.4
Q-BERT MP	2/3 MP	8	**81.75**	**82.29**	**46.1**	**23.4**

(c) CoNLL03

Method	w-bits	e-bits	F_1	Size	Size-w/o-e
Baseline	32	32	95.00	410.9	324.5
Q-BERT	8	8	94.79	102.8	81.2
DirectQ	4	8	89.86	62.2	40.6
Q-BERT	4	8	**94.90**	62.2	40.6
DirectQ	3	8	84.92	52.1	30.5
Q-BERT	3	8	**94.78**	52.1	30.5
Q-BERT MP	2/4 MP	8	**94.55**	52.1	30.5
DirectQ	2	8	54.50	42.0	20.4
Q-BERT	2	8	**91.06**	42.0	20.4
Q-BERT MP	2/3 MP	8	**94.37**	**45.0**	**23.4**

Table 7.2: Quantization results for BERTBASE on SQuAD.

Method	w-bits	e-bits	EM	F_1	Size	Size-w/o-e
Baseline	32	32	81.54	88.69	415.4	324.5
Q-BERT	8	8	81.07	88.47	103.9	81.2
DirectQ	4	8	66.05	77.10	63.4	40.6
Q-BERT	4	8	**80.95**	**88.36**	63.4	40.6
DirectQ	3	8	46.77	59.83	53.2	30.5
Q-BERT	3	8	**79.96**	**87.66**	53.2	30.5
Q-BERT MP	2/4 MP	8	**79.85**	**87.49**	53.2	30.5
DirectQ	2	8	4.77	10.32	43.1	20.4
Q-BERT	2	8	**69.68**	**79.60**	43.1	20.4
Q-BERT MP	2/3 MP	8	**79.25**	**86.95**	**48.1**	**25.4**

variance of the top eigenvalue of the Hessian shown in Figure 7.1. Note that all experiments in Figure 7.1 are based on 10 runs and each run uses 10% of the entire training dataset. We can obverse that for most of the lower encoder layers (layer 1-8), the variance is pretty large compared to the last three layers. We generally observe that the middle part (layer 4-8) has the largest mean(λ_i). Beyond the relatively smaller mean, the last three layers also have a much smaller variance, which indicates the insensitivity of these layers. Therefore, higher bits will only be assigned for middle layers according to Eq. 7.1 for Q-BERT 2/3 MP. In this way, with only additional 5MB memory storage, 2/3-bits Q-BERT MP is able to retain the performance drop within 2.3% for MNLI, SQuAD and 1.1% for SST-2, CoNLL-03, with up to 13× compression ratio in weights. Note that this is up to 6.8% better than Q-BERT with uniform 2 bits.

One consideration for quantization is that 3-bit quantized execution is typically not supported in hardware. It is however possible to load 3-bit quantized values and cast them to higher bit precision such as 4 or 8 bits in the execution units. This would still have the benefit of reduced memory volume to/from DRAM. It is also possible to avoid using 3 bits and instead use a mixture of 2 and 4 bits as shown in Table 7.1. For example, SST-2 Q-BERT MP with mixed 2/4-bit precision weights has the same model size as the 3-bit quantization in 53.2MB and achieves similar accuracy. We observe similar trends for other tasks as well.

One important observation is that we found SQuAD to be harder to quantize as compared to other tasks (as shown in Table 7.2). For example, 2-bits DirectQ results in more than 10% F_1 score degradation. Even Q-BERT has a larger performance drop as compared to other tasks in Table 7.1. We studied this phenomenon further through Hessian analysis. In Figure 7.1, among all the tasks, it can be clearly seen that SQuAD not only has a much larger eigenvalue variance, but it has very large negative eigenvalues. In fact, this shows that the existing BERT model for SQuAD has not reached a local minima. This is further illustrated in the 3-d loss landscape of all four tasks in Figure 7.2 and Figure 7.3. It can be clearly seen that for the other three tasks, the stopping point is at a quadratic bowl (at least in the first two dominant eigenvalue directions of the Hessian). However, compared to the others, SQuAD has a totally different structure to its loss landscape. As shown in Figure 7.3, the stopping points of different layers on SQuAD have negative curvature directions, which means they have not converged to a local minimum yet. This could well explain why the quantization of SQuAD results in more accuracy drop. Our initial attempts to address this by changing training hyper-parameters were not successful. We found that the BERT model quickly overfits the training data. However, we emphasize that fixing BERT model training itself is outside the scope and hard with academic computational resources.

Ablation Study

Effects of group-wise quantization

We measure the performance gains with different group numbers in Table 7.3. We can observe from the table that performing layer-wise quantization is sub-optimal for all four tasks (the performance drop is around 7% to 11.5%). However, the performance significantly increases as we increase the number of groups. For example, for 12 groups, the performance degradation is less than 2% for all the tasks. Further increasing the group number from 12 to 128 increases the accuracy further by at least 0.3% accuracy. However, increasing the group number further from 128 to 768 can only increase the performance by 0.1%. This shows that the performance gain almost saturates around 128 groups. It is also preferable not to have a very large value for the number of groups since it increases the number of Look-up Tables (LUTs) necessary for each matrix multiplication which can adversely affect hardware performance, and based on our results there are diminishing returns in terms of accuracy. In all our experiments in Section 7.3, we used 128 groups for both Q-BERT and Q-BERT MP.

Table 7.3: Effects of group-wise quantization for Q-BERT on three tasks. The quantization bits were set to be 4 for weights, 8 for embeddings and 8 for activations for all the tasks. From top to down, we increase the number of groups. In order to balance the accuracy and hardware efficiency, we set 128 groups for other experiments.

# Group	SST-2	MNLI-m/mm	CoNLL-03
Baseline	93.00	84.00/84.40	95.00
1	85.67	76.69/77.00	89.86
12	92.31	82.37/82.95	94.42
128	92.66	83.89/84.17	94.90
768 [3]	92.78	84.00/84.20	94.99

[3] Here we treat each output neuron as a single group.

Quantization effects on different modules of the BERT model.

Here we investigate the quantization effects with respect to different modules of the BERT model (multi-head self-attention versus feed-forward network, and different embedding layers, i.e., word embedding versus position embedding).

Generally speaking, we find that the embedding layer is more sensitive than weights for quantization. This is illustrated in Table 7.4, where we use 4-bits layerwise quantization for embedding, which results in an unacceptable performance drop up to 10% for SST-2, MNLI, CoNLL-03 and even more than 20% for SQuAD. This is despite the fact that we used 8/8-bits for weights/activations. On the contrary, encoder layers consume around 79% total parameters (4× embedding parameter size), while quantizing them to 4-bits in Table 7.1 and 7.2 leads to less performance loss.

Furthermore, we find that position embedding is very sensitive to quantization. For instance, quantizing position embedding to 4 bits results in generally 2% additional performance degradation than quantizing word embedding, even though the position embedding only accounts for less than 5% of the entire embedding. This indicates the importance of positional information in Natural Language Understanding tasks. Given position embedding only accounts for a small portion of model size, we can do mixed-precision quantization for embedding to further push down the model size boundary with a tolerable accuracy drop.

Table 7.4: Quantization effect to different modules. We abbreviate the quantization bits used for word embedding as "ew-bits", position embedding as "ep-bits", multi-head attention layer as "s-bits", and fully-connected layer as "f-bits". In (a), we set weight and activation bits as 8. In (b), we set embedding and activation bits as 8.

(a) quantization effect on embedding

Method	ew-bits	ep-bits	SST-2	MNLI-m/mm	CoNLL-03	SQuAD
Baseline	32	32	93.00	84.00/84.40	95.00	88.69
Q-BERT	8	8	92.88	83.83/83.91	94.79	88.47
Q-BERT	4	8	91.74	82.91/83.67	94.44	87.55
Q-BERT	8	4	89.11	82.84/82.25	93.86	72.38
Q-BERT	4	4	85.55	78.08/78.96	84.32	61.70

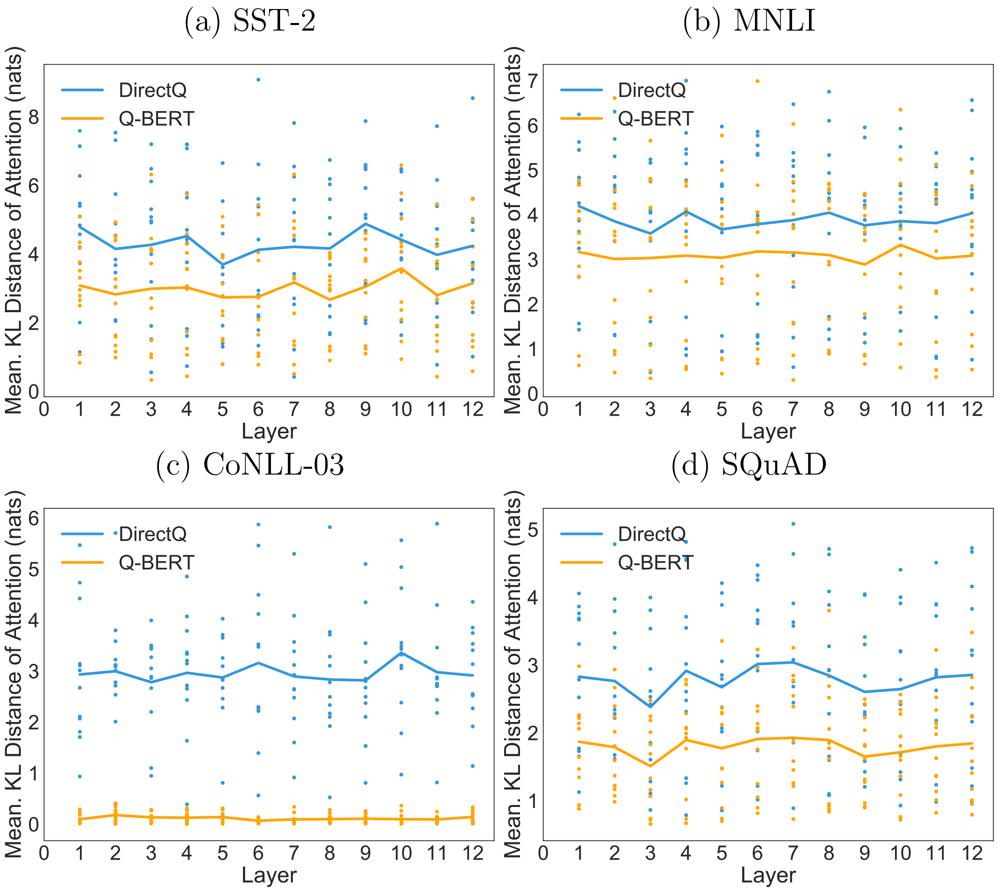
(b) quantization of multi-head attention versus fully-connected layer

Method	s-bits	f-bits	SST-2	MNLI-m/mm	CoNLL-03	SQuAD
Baseline	32	32	93.00	84.00/84.40	95.00	88.69
Q-BERT MP	1/2 MP	2/3 MP	89.56	73.66/74.52	91.74	75.81
Q-BERT MP	2/3 MP	1/2 MP	85.89	70.89/71.17	87.55	68.71
Q-BERT MP	2/3 MP	2/3 MP	92.08	81.75/82.29	93.91	86.95

To study the quantization effects on self-attention layers and fully-connected networks, we conducted extensive experiments under different bits settings for the encoder layers. The results are shown in Table 7.4. Specifically, we adopt the Q-BERT MP setting in Table 7.1, with a mixture of 2 and 3 bits for encoder weights. To test the robustness of the two modules inside each encoder layer, we further reduce one more bit in the corresponding modules and denote the resulting precision setting 1/2 MP. From Table 7.4, we can conclude that generally self-attention layer is more robust to quantization than the fully-connected network, since 1/2 MP self-attention results in about 5% performance drop while 1/2 MP fully-connected will worsen this to 11%.

Qualitative Analysis

We use attention information to conduct qualitative analysis to analyze the difference between Q-BERT and DirectQ. Specifically, we compute the Kullback–Leibler (KL) divergence between the attention distribution for the same input from the coordinated head of both quantized BERT and full-precision BERT. It should be noted that we compute the average distance out of 10% of the entire training dataset. The smaller KL divergence here means that the output of the multi-head attention of the two models is closer to each other. We illustrate this distance score for each individual head in Figure 7.5 for SST-2, MNLI, CoNLL-03, and SQuAD. We compared Q-BERT and DirectQ with 4-bits weights, 8-bits embedding, and 8-bits activation. Each scatter point in Figure 7.5 denotes the distance w.r.t one head, and the line chart shows the average results over the 12 heads in one layer. We can clearly see that Q-BERT always incurs a smaller distance to the original baseline model as compared to the DirectQ model, for all the different layers.


(a) SST-2

(b) MNLI

(c) CoNLL-03

(d) SQuAD

Figure 7.5: KL divergence over attention distribution between Q-BERT/DirectQ and Baseline. The distance between Q-BERT and Baseline is much smaller than that of DirectQ.

Chapter 8

Quantization: ZeroQ

The quantization methods in previous Chapters require quantization-aware training (QAT), while the data, as well as the time and resources needed for QAT, are not always accessible. In this Chapter, we focus on solving the *key question*:

> How to perform quantization without access to the data and fine-tuning?

Here we present ZeroQ, which can generate Distilled data based on the batchnorm statistics of the pretrained models. ZeroQ is able to achieve both uniform quantization and mixed-precision quantization in real time without data and QAT.

8.1 Method

The ZeroQ framework supports both fixed-precision and mixed-precision quantization. As we will show later, this mixed-precision quantization is key to achieving high accuracy for ultra-low precision settings such as 4-bit quantization. Typical choices for precision k are $\{2, 4, 8\}$ bit. Note that this mixed-precision quantization leads to exponentially large search space, as every layer could have one of these bit precision settings. It is possible to avoid this prohibitive search space if we could measure the sensitivity of the model to the quantization of each layer [53, 211, 52]. For the case of post-training quantization (i.e. without fine-tuning), a good sensitivity metric is to use Kullback–Leibler (KL) divergence between the original model and the quantized model, defined as:

$$\Omega_i(k) = \frac{1}{N} \sum_{j=1}^{N_{dist}} KL(\mathcal{M}(\theta; x_j), \mathcal{M}(\tilde{\theta}_i(k-bit); x_j)). \tag{8.1}$$

where $\Omega_i(k)$ measures how sensitive the i-th layer is when quantized to k-bit, and $\tilde{\theta}_i(k\text{-}bit)$ refers to quantized model parameters in the i-th layer with k-bit precision. If $\Omega_i(k)$ is small, the output of the quantized model will not significantly deviate from the output of the full precision model when quantizing the i-th layer to k-bits, and thus the i-th layer is relatively insensitive to *k-bit* quantization, and vice versa. However, an important problem is that for zero-shot quantization we do not have access to the original training dataset x_j in Eq. 8.1. We address this by "distilling" synthetic input data to match the statistics of the original training dataset, which we refer to as Distilled Data. We obtain the Distilled Data by solely analyzing the trained model itself, as described below.

Distilled Data

For zero-shot quantization, we do not have access to any of the training/validation data. This poses two challenges. First, we need to know the range of values for activations of each layer so that we can clip the range for quantization (the $[a, b]$ range mentioned above). However, we cannot determine this range without access to the training dataset. This is a problem for both uniform and mixed-precision quantization. Second, another challenge is that for mixed-precision quantization, we need to compute Ω_i in Eq. 8.1, but we do not have access to training data x_j. A very naïve method to address these challenges is to create random input data drawn from a Gaussian distribution with zero mean and unit variance and feed it into the model. However, this approach cannot capture the correct statistics of the activation data corresponding to the original training dataset. This is illustrated in Figure 8.2 (left), where we plot the sensitivity of each layer of ResNet50 on ImageNet measured with the original training dataset (shown in black) and Gaussian-based input data (shown in red). As one can see, the Gaussian data clearly does not capture the correct sensitivity of the model. For instance, for the first three layers, the sensitivity order of the red line is actually the opposite of the original training data.

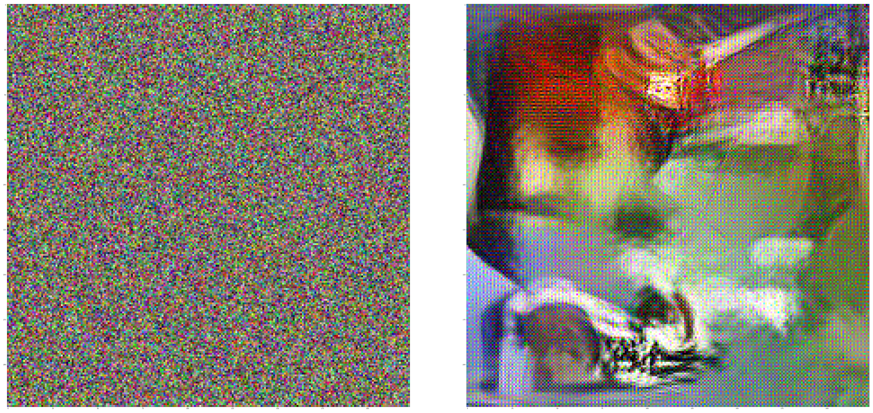


Figure 8.1: Visualization of Gaussian data (left) and Distilled Data (right). More local structures can be seen in our Distilled Data that is generated according to Algorithm 3.

To address this problem, we propose a novel method to "distill" input data from the NN model itself, i.e., to generate synthetic data carefully engineered based on the properties of the NN. In particular, we solve a distillation optimization problem, in order to learn an input

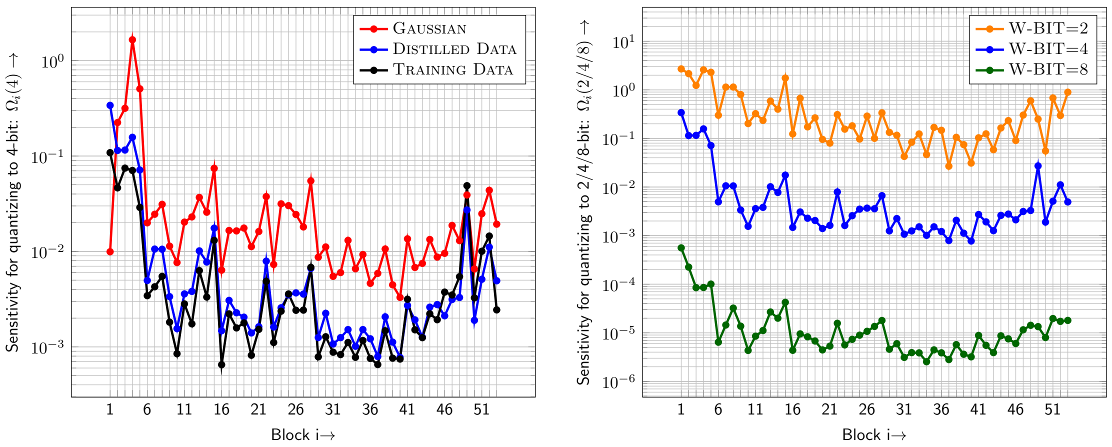


Figure 8.2: (Left) Sensitivity of each layer in ResNet50 when quantized to 4-bit weights, measured with different kinds of data (red for Gaussian, blue for Distilled Data, and black for training data). (Right) Sensitivity of ResNet50 when quantized to 2/4/8-bit weight precision (measured with Distilled Data).

data distribution that best matches the statistics encoded in the BN layer of the model. In more detail, we solve the following optimization problem:

$$\min_{x^r} \sum_{i=0}^{L} \|\tilde{\mu}_i^r - \mu_i\|_2^2 + \|\tilde{\sigma}_i^r - \sigma_i\|_2^2, \tag{8.2}$$

where x^r is the reconstructed (distilled) input data, and μ_i^r/σ_i^r are the mean/standard deviation of the Distilled Data's distribution at layer i, and μ_i/σ_i are the corresponding mean/standard deviation parameters stored in the BN layer at layer i. In other words, after solving this optimization problem, we can distill input data which, when fed into the network, can have a statistical distribution that closely matches the original model. This Distilled Data can then be used to address the two challenges described earlier. First, we can use the Distilled Data's activation range to determine quantization clipping parameters (the $[a, b]$ range mentioned above). Note that some prior work [10, 127, 284] address this by using limited (unlabeled) data to determine the activation range. However, this contradicts the assumptions of zero-shot quantization, and may not be applicable for certain applications. Second, we can use the Distilled Data and feed it in Eq. 8.1 to determine the quantization sensitivity (Ω_i). The latter is plotted for ResNet50 in Figure 8.2 (left) shown in solid blue color. As one can see, the Distilled Data closely matches the sensitivity of the model as compared to using Gaussian input data (shown in red). We show a visualization of the random Gaussian data as well as the Distilled Data for ResNet50 in Figure 8.1. We can see that the Distilled Data can capture fine-grained local structures.

```
Algorithm 3: Generation of Distilled Data
  Input: Model: $\mathcal{M}$ with $L$ Batch Normalization layers
  Output: A batch of distilled data: $x^r$
  Generate random data from Gaussian: $x^r$
  Get $\mu_i, \sigma_i$ from Batch Normalization layers of $\mathcal{M}$, $i \in 0,\ 1,\ \ldots,\ L$
                                                  // Note that $\mu_0 = 0,\ \ \sigma_0 = 1$
  for $j = 1, 2, \ldots$ do
      Forward propagate $\mathcal{M}(x^r)$ and gather intermediate activations
      Get $\tilde{\mu}_i$ and $\tilde{\sigma}_i$ from intermediate activations, $i \in 1,\ \ldots,\ n$
      Compute $\tilde{\mu}_0$ and $\tilde{\sigma}_0$ of $x^r$
      Compute the loss based on Eq. 8.2
      Backward propagate and update $x^r$
  end
```

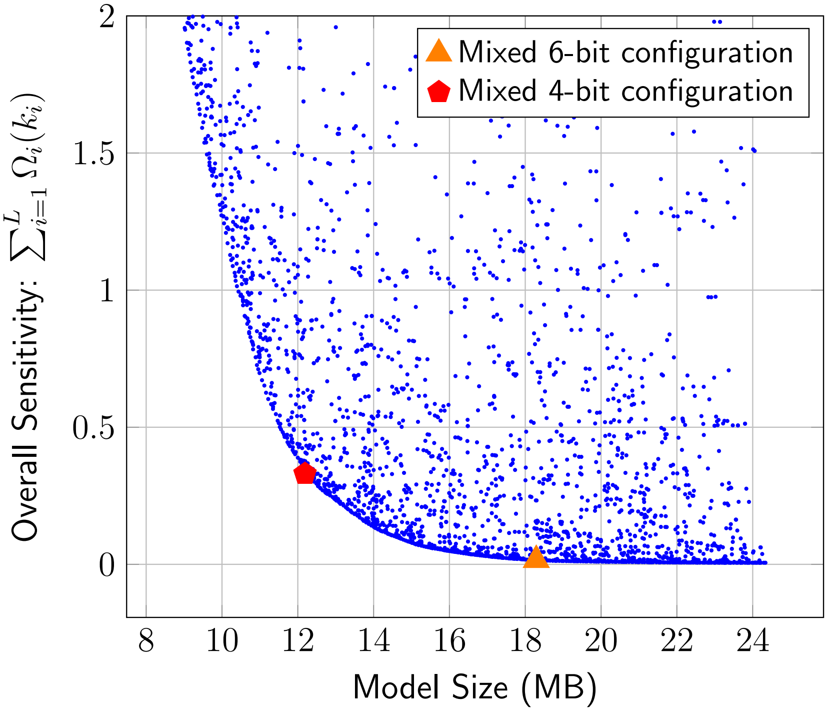


Figure 8.3: The Pareto frontier of ResNet50 on ImageNet. Each point shows a mixed-precision bit setting. The x-axis shows the resulting model size for each configuration, and the y-axis shows the resulting sensitivity. In practice, a constraint for model size is set. Then the Pareto frontier method chooses a bit-precision configuration that results in minimal perturbation. We show two examples of 4 and 6-bit mixed precision configurations shown in red and orange. The corresponding results are presented in Table 11.1.

Pareto Frontier

As mentioned before, the main challenge for mixed-precision quantization is to determine the exact bit precision configuration for the entire NN. For an L-layer model with m possible precision options, the mixed-precision search space, denoted as $\mathcal{S}$, has an exponential size of m^L. For example for ResNet50 with just three bit precision of $\{2, 4, 8\}$ (i.e., $m = 3$), the search space contains 7.2×10^{23} configurations. However, we can use the sensitivity metric in Eq. 8.1 to reduce this search space. The main idea is to use higher bit precision for layers that are more sensitive, and lower bit precision for layers that are less sensitive. This gives us a relative ordering of the number of bits. To compute the precise bit precision setting, we propose a Pareto frontier approach similar to the method used in [52].

The Pareto frontier method works as follows. For a target quantized model size of S_{target}, we measure the overall sensitivity of the model for each bit precision configuration that results in the S_{target} model size. We choose the bit-precision setting that corresponds to the minimum overall sensitivity. In more detail, we solve the following optimization problem:

$$\min_{\{k_i\}_{i=1}^{L}} \Omega_{sum} = \sum_{i=1}^{L} \Omega_i(k_i) \quad s.t. \sum_{i=1}^{L} P_i * k_i \le S_{target}, \tag{8.3}$$

where k_i is the quantization precision of the i-th layer, and P_i is the parameter size for the i-th layer. Note that here we make the simplifying assumption that the sensitivity of different layers is independent of the choice of bits for other layers (hence Ω_i only depends on the bit precision for the i-th layer). Please refer to the Appendix of [21] for more details, where we describe how we relax this assumption without having to perform an exponentially large computation for the sensitivity for each bit precision setting. Using a dynamic programming method we can solve the best setting with different S_{target} together, and then we plot the Pareto frontier. An example is shown in Figure 8.3 for ResNet50 model, where the x-axis is the model size for each bit precision configuration, and the y-axis is the overall model perturbation/sensitivity. Each blue dot in the figure represents a mixed-precision configuration. In ZeroQ, we choose the bit precision setting that has the smallest perturbation with a specific model size constraint.

Importantly, note that the computational overhead of computing the Pareto frontier is $\mathcal{O}(mL)$. This is because we compute sensitivity Ω_i $(i = 1, 2, ..., L)$ with respect to all m different precision options, which leads to the $\mathcal{O}(mL)$ computational complexity. We should note that this Pareto Frontier approach (including the Dynamic Programming optimizer), is not theoretically guaranteed to result in the best possible configuration, out of all possibilities in the exponentially large search space. However, our results show that the final mixed-precision configuration achieves state-of-the-art accuracy with small performance loss, as compared to the original model in single precision.

Table 8.1: Quantization results of ResNet50, MobileNetV2, and ShuffleNet on ImageNet. We abbreviate quantization bits used for weights as "W-bit" (for activations as "A-bit"), top-1 test accuracy as "Top-1." Here, "MP" refers to mixed-precision quantization, "No D" means that none of the data is used to assist quantization, and "No FT" stands for no fine-tuning (re-training). Compared to post-quantization methods OCS [284], OMSE [122], and DFQ [178], ZeroQ achieves better accuracy. ZeroQ† means using percentile for quantization.

(a) ResNet50

Method	No D	No FT	W-bit	A-bit	Size (MB)	Top-1
Baseline	–	–	32	32	97.49	77.72
OMSE [122]	✓	✓	4	32	12.28	70.06
OMSE [122]	✗	✓	4	32	12.28	74.98
PACT [33]	✗	✗	4	4	12.19	76.50
ZeroQ	✓	✓	MP	8	**12.17**	**75.80**
ZeroQ†	✓	✓	MP	8	**12.17**	**76.08**
OCS [284]	✗	✓	6	8	18.46	74.80
ZeroQ	✓	✓	MP	6	**18.27**	**77.43**
ZeroQ	✓	✓	8	8	24.37	**77.67**

(b) MobileNetV2

Method	No D	No FT	W-bit	A-bit	Size (MB)	Top-1
Baseline	–	–	32	32	13.37	73.03
ZeroQ	✓	✓	MP	8	1.67	**68.83**
ZeroQ†	✓	✓	MP	8	1.67	**69.44**
Integer-Only [107]	✗	✗	6	6	2.50	70.90
ZeroQ	✓	✓	MP	6	2.50	**72.85**
RVQuant [187]	✗	✗	8	8	3.34	70.29
DFQ [178]	✓	✓	8	8	3.34	71.20
ZeroQ	✓	✓	8	8	3.34	**72.91**

(c) ShuffleNet

Method	No D	No FT	W-bit	A-bit	Size (MB)	Top-1
Baseline	–	–	32	32	5.94	65.07
ZeroQ	✓	✓	MP	8	0.74	**58.96**
ZeroQ	✓	✓	MP	6	1.11	**62.90**
ZeroQ	✓	✓	8	8	1.49	**64.94**

8.2 Experiments

In this section, we extensively test ZeroQ on a wide range of models and datasets. All of the results achieved by ZeroQ are 100% zero-shot without any need for fine-tuning. We also emphasize that we used exactly the same hyper-parameters (e.g., the number of iterations to generate Distilled Data) for all experiments, including the results on Microsoft COCO.

ImageNet

For each model, after generating Distilled Data based on Eq. 8.2, we compute the sensitivity of each layer using Eq. 8.1 for different bit precision. Next, we use Eq. 8.3 and the Pareto frontier introduced in Section 8.1 to get the best bit-precision configuration based on the overall sensitivity for a given model size constraint. We denote the quantized results as WwAh where w and h denote the bit precision used for weights and activations of the NN model.

We present zero-shot quantization results for ResNet50 in Table 11.1. As one can see, for W8A8 (i.e., 8-bit quantization for both weights and activations), ZeroQ results in only 0.05% accuracy degradation. Further quantizing the model to W6A6, ZeroQ achieves 77.43% accuracy, which is 2.63% higher than OCS [284], even though our model is slightly smaller (18.27MB as compared to 18.46MB for OCS).[1] We show that we can further quantize ResNet50 down to just 12.17MB with mixed precision quantization, and we obtain 75.80% accuracy. Note that this is 0.82% higher than OMSE [122] with access to training data and 5.74% higher than the zero-shot version of OMSE. Importantly, note that OMSE keeps activation bits at 32-bits, while for this comparison our results use 8-bits for the activation (i.e., 4× smaller activation memory footprint than OMSE). For comparison, we include results for PACT [33], a standard quantization method that requires access to training data and also requires fine-tuning.

An important feature of the ZeroQ framework is that it can perform quantization with very low computational overhead. For example, the end-to-end quantization of ResNet50 takes less than 30 seconds on 8 Tesla V100 GPUs (one epoch training time on this system takes 100 minutes). In terms of timing breakdown, it takes 3s to generate the Distilled Data, 12s to compute the sensitivity for all layers of ResNet50, and 14s to perform Pareto Frontier optimization.

We also show ZeroQ results on MobileNetV2 and compare it with both DFQ [178] and fine-tuning based methods [187, 107], as shown in Table 8.1. For W8A8, ZeroQ has less than 0.12% accuracy drop as compared to baseline, and it achieves a 1.71% higher accuracy as compared to the DFQ method.

Further compressing the model to W6A6 with mixed-precision quantization for weights, ZeroQ can still outperform Integer-Only [107] by 1.95% accuracy, even though ZeroQ does not use any data or fine-tuning. ZeroQ can achieve 68.83% accuracy even when the weight compression is 8×, which corresponds to using 4-bit quantization for weights on average.

We experimented with percentile-based clipping to determine the quantization range [130] (please see the Appendix of [21] for details). The results corresponding to percentile-based

[1] Importantly note that OCS requires access to the training data to do activation quantization while requires no data for weight quantization only. ZeroQ does not use any training/validation data for both weight and activation quantization.

Table 8.2: Uniform post-quantization on ImageNet with ResNet18. We use percentile clipping for W4A4 and W4A8 settings. ZeroQ† means using percentile for quantization.

Method	No D	No FT	W-bit	A-bit	Size (MB)	Top-1
Baseline	–	–	32	32	44.59	71.47
PACT [33]	✗	✗	4	4	5.57	69.20
DFC [85]	✓	✓	4	4	5.58	55.49
DFC [85]	✓	✗	4	4	5.58	68.06
ZeroQ†	✓	✓	MP	4	**5.57**	**69.05**
Integer-Only[107]	✗	✗	6	6	8.36	67.30
DFQ [178]	✓	✓	6	6	8.36	66.30
ZeroQ	✓	✓	MP	6	**8.35**	**71.30**
RVQuant [187]	✗	✗	8	8	11.15	70.01
DFQ [178]	✓	✓	8	8	11.15	69.70
DFC [85]	✓	✗	8	8	11.15	69.57
ZeroQ	✓	✓	8	8	11.15	**71.43**

clipping are denoted as *ZeroQ*† and reported in Table 8.1. We found that using percentile-based clipping is helpful for low precision quantization. Other choices for clipping methods have been proposed in the literature. Here we note that our approach is orthogonal to these improvements and that ZeroQ could be combined with these methods.

We applied ZeroQ to quantize efficient and highly compact models such as ShuffleNet, whose model size is only 5.94MB. ZeroQ achieves a small accuracy drop of 0.13% for W8A8. We can further quantize the model down to an average of 4-bits for weights, which achieves a model size of only 0.73MB, with an accuracy of 58.96%.

We also compare with the recent Data-Free Compression (DFC) [85] method. There are two main differences between ZeroQ and DFC. First, DFC proposes a fine-tuning method to recover accuracy for ultra-low precision cases. This can be time-consuming and as we show it is not necessary. In particular, we show that with mixed-precision quantization one can actually achieve higher accuracy without any need for fine-tuning. This is shown in Table 8.2 for ResNet18 quantization on ImageNet. In particular, note the results for W4A4, where the DFC method without fine-tuning results in more than 15% accuracy drop with a final accuracy of 55.49%. For this reason, the authors propose a method with post-quantization training, which can boost the accuracy to 68.05% using W4A4 for intermediate layers, and 8-bits for the first and last layers. In contrast, ZeroQ achieves a higher accuracy of 69.05% without any need for fine-tuning. Furthermore, the end-to-end zero-shot quantization of ResNet18 takes only 12s on an 8-V100 system (equivalent to 0.4% of the 45 minutes time for one epoch training of ResNet18 on ImageNet). Finally, the DFC method uses Inceptionism [176] to facilitate the generation of data with random labels, but it is hard to extend this for object detection and image segmentation tasks.

Microsoft COCO

Object detection is often much more complicated than ImageNet classification. To demonstrate the flexibility of our approach we also test ZeroQ on an object detection task on the Microsoft COCO dataset. RetinaNet [144] is a state-of-the-art single-stage detector, and we use the pretrained model with ResNet50 as the backbone, which can achieve 36.4 mAP.[2]

Table 8.3: Object detection on Microsoft COCO using RetinaNet. By keeping activations to be 8-bit, our 4-bit weight result is comparable with the recently proposed method FQN [130], which relies on fine-tuning. (Note that FQN uses 4-bit activations and the baseline used in [130] is 35.6 mAP).

Method	No D	No FT	W-bit	A-bit	Size (MB)	mAP
Baseline	✓	✓	32	32	145.10	36.4
FQN [130]	✗	✗	4	4	18.13	32.5
ZeroQ	✓	✓	MP	8	18.13	**33.7**
ZeroQ	✓	✓	MP	6	24.17	**35.9**
ZeroQ	✓	✓	8	8	36.25	**36.4**

One of the main differences between RetinaNet and previous NNs we tested on ImageNet is that some convolutional layers in RetinaNet are not followed by BN layers. This is because of the presence of a feature pyramid network (FPN) [143], and it means that the number of BN layers is slightly smaller than that of convolutional layers. However, this is not a limitation and the ZeroQ framework still works well. Specifically, we extract the backbone of RetinaNet and create Distilled Data. Afterward, we feed the Distilled Data into RetinaNet to measure the sensitivity as well as to determine the activation range for the entire NN. This is followed by optimizing for the Pareto Frontier.

The results are presented in Table 8.3. We can see that W8A8 ZeroQ has no performance degradation. For W6A6, ZeroQ achieves 35.9 mAP. Further quantizing the model to an average of 4-bits for the weights, ZeroQ achieves 33.7 mAP. Our results are comparable to the recent results of FQN [130], even though it is not a zero-shot quantization method (i.e., it uses the full training dataset and requires fine-tuning). However, it should be mentioned that ZeroQ keeps the activations to be 8-bits, while FQN uses 4-bit activations.

[2] Here we use the standard mAP 0.5:0.05:0.95 metric on COCO dataset.

Chapter 9

Conclusion on Quantization

In this thesis, we conduct research on mixed-precision quantization. The major challenge is finding the right precision setting for different layers, where a brute force approach is infeasible since the search space is exponentially large in the number of layers. Previous methods require large computational resources, and their performance is very sensitive to hyper-parameters and initialization.

To address these issues, we propose Hessian AWare Quantization (HAWQ) [53] where we use Hessian information (top-1 eigenvalue) to measure the sensitivity of each layer, and we assign higher bitwidth for layers that are more sensitive and lower bitwidth for layers that are less sensitive. The HAWQ method can generate relative sensitivity among layers, but cannot automatically determine the exact bitwidth for each layer. We further propose HAWQ-V2 [52] where we introduce an automatic way to find good mixed-precision settings. We first provide theoretical proof that the trace of the Hessian matrix is a better sensitivity measurement. Then we formulate the mixed-precision quantization problem to be an integer linear programming. We propose a Pareto Frontier solver to find mixed-precision settings automatically, and our results outperform human-tuned HAWQ results by a large margin.

In order to justify the generalization ability of our methods in different domains, we propose Q-BERT for natural language processing (NLP) tasks. We found NLP models are more in need of quantization given their formidable model size compared with computer vision models (BERT-large 700MB v.s. ResNet50 95MB). In Q-BERT, we propose group-wise quantization and we apply Hessian-based sensitivity analysis. We achieve state-of-the-art quantization results on MNLI/SST-2/Squad/NER tasks with model size down to 20MB.

Despite the advancement of the aforementioned methods, the standard QAT process is still needed before we can achieve accelerated inference. In many scenarios, this QAT can be infeasible because of privacy issues (no data for fine-tuning) or online learning requirements (no time for fine-tuning). Therefore, we proposed zero-shot quantization (ZeroQ), where we generate distilled data based on the batch normalization statistics stored in pretrained networks, and we perform mixed-precision quantization without fine-tuning the model. With no data or fine-tuning, ZeroQ can outperform previous post-training methods by a large margin, and can even match the quantization results that are fine-tuned with data.

Chapter 10

Introduction and Related Work of HW-SW Co-Design

10.1 Hardware-Aware Neural Architecture Design

Manual design of efficient models Works have been conducted to optimize the NN model architecture in terms of its micro-architecture [106, 96, 165, 160, 252, 203, 118, 283] (e.g., kernel types such as depth-wise convolution or low-rank factorization) as well as its macro-architecture [105, 96, 98, 205, 95, 223] (e.g., module types such as residual, or inception). In addition to improving accuracy, these previous works are hardware-aware since they consider proxy efficiency metrics such as the model size and FLOPs during the design flow. However, as mentioned in Chapter 2, practical metrics such as latency and throughput are actually more important in real-world applications. Moreover, the manual design requires domain knowledge and validation, which can be computationally intensive and not scalable.

Neural architecture search Previous works [295, 189, 150, 225, 19, 20, 250, 76, 55] first define a search space for potential neural architectures, and then apply algorithms such as reinforcement learning, evolutionary searching, differentiable searching, and random sampling etc. to search for good neural architectures. Early NAS algorithms [295, 189, 225] generally require a huge amount of computational resources to search for a decent result. Based on weight sharing, differentiable NAS algorithms (DNAS) [150, 19, 250] are introduced to reduce the searching cost by jointly optimizing the architectural factors and the weights in the network. Since DNAS trains a supernet containing all candidate operations, the total amount of operations becomes limited by the memory and computational constraint, which leads to smaller search spaces such as the DARTS [150] search space or the MobileNet [205, 95] search space. Although DNAS methods generally perform well, the search space itself is pre-defined and requires heuristic knowledge on specific tasks. In addition to solely optimizing accuracy, many previous works [19, 20, 225, 250, 235] also take efficiency metrics such as model size and FLOPs into consideration. Furthermore, some of these works retrieve latency or energy feedback from a given hardware platform, and search for optimal DNNs that can meet certain application constraints. Despite the merits, we should note that the hardware specifications are fixed in these methods, and thus are not included in the search space.

Neural architecture transformation (teacher-based NAS) [129, 174, 175] take a pre-trained model as a teacher network and then apply layer-wise or block-wise knowledge distillation to transform the teacher network into student networks with desired neural architectures. These algorithms provide a large search space compared with DNAS while being more computationally efficient than NAS based on reinforcement learning or evolutionary search. One shortcoming of teacher-based NAS methods is that they cannot change the resolution of student networks (since it has to be the same as the teacher network). Additionally, an evaluation metric is required to compare against different student networks, which is not guaranteed to be accurate in previous works (for example, the additive metric used in [174]). [129, 175] propose to use predictors for the ranking or the accuracy of student networks. Despite the effectiveness, the overhead of extra training and evaluation used to obtain the predictors makes [129, 175] inefficient. Like the NAS methods, neural architecture transformation works also consider metrics in Chapter 2 to balance efficiency and accuracy. Although these works are hardware-aware, they use a fixed hardware platform, the same as in the NAS methods.

10.2 Hardware-Aware Model Compression

Since inference speed is dependent on the characteristics of specific hardware platforms, simply applying model compression can be sub-optimal. To solve this problem, many hardware-aware compression methods [241, 92, 209, 272, 275, 244, 31] have been introduced to seek efficient inference of DNNs on targeted hardware platforms.

Specifically on quantization, not all hardware provides the same speedup after a certain layer/operation is quantized. The benefits from quantization are hardware-dependant, with many factors such as on-chip memory, bandwidth, and the cache hierarchy. It is important to consider these factors through hardware-aware quantization [254, 236, 90, 251, 262, 268, 243, 86]. In particular, previous work [236] uses a reinforcement learning agent to determine the hardware-aware mixed-precision setting for quantization, based on a look-up table of latency with respect to different layers with different bitwidth. However, this approach uses simulated hardware latency instead of real latency. To address this, the recent work of [268] directly deploys quantized operations on hardware and measures the actual deployment latency of each layer for different quantization bit precisions. It should be noted that, although some of the previous works jointly consider neural architecture and model compression, most of them are agnostic to both the neural architecture and the hardware specifications.

10.3 Hardware-Software Co-Optimization

To further improve the efficiency, in recent years, a few works have extended the previous NAS framework by integrating hardware design into the search space [281, 158, 83, 84, 139, 114, 113, 46, 261, 2, 1, 207, 112, 263]. Generally, these software/hardware co-search algorithms adopt pre-defined hardware design templates and incorporate several high-level design hyperparameters in the search framework. In addition to neural architectures, some works also incorporated quantization in their search space. [158] captures the relationship between quantization bitwidth and LUTs consumption on FPGA, and developed a NAS algorithm under the constraint of LUTs. Besides FPGA, [48, 81] conduct co-design on top of processing in memory (PIM) platforms. In [114], the authors integrate several model compression techniques in the search framework and use quantization to reduce the latency of weight loading. [139] proposes a uniformed differentiable search algorithm using Gumbel-softmax to sample discrete implementation hyperparameters including quantization bitwidth.

Although previous methods consider hardware design choices, the size of searchable space is still limited by the efficiency of searching algorithms and the total computation budget. Consequently, enlarging hardware search space may result in the shrinkage of software search space. Moreover, in order to explore the co-design space, manual design or heuristics are often included in the design pipeline, which requires extra domain knowledge and effort. Fully automatic searching methods can save manual efforts, but they may be stuck at sub-optimal points, or take too much computational resource and time. Consequently, further studies on efficient HW-SW co-design algorithms are still necessary.

Chapter 11

HW-SW Co-Design: HAWQV3

Quantization is very effective in compressing the model size. However, co-designing and deploying quantized models on different hardware platforms, and achieving inference speedup, can take non-trivial efforts. In this Chapter, we discuss the following *key question*:

> How to automatically adapt quantization schemes for different hardware platforms, and how to efficiently deploy quantized models and achieve inference speedup?

To answer the question, we present HAWQV3, which implements ultra-low precision and mixed-precision quantization support on TVM, and applies hardware-aware quantization to directly optimize the inference speed.

11.1 Method

Quantized Matrix Multiplication and Convolution

Consider a layer with hidden activation denoted as h and weight tensor denoted as W, followed by ReLU activation. First, h and W are quantized to $S_h q_h$ and $S_w q_w$, where S_h and S_w are the real-valued quantization scales, q_h and q_W are the corresponding quantized integer values. The output result, denoted with a, can be computed as follows:

$$a = S_w S_h (q_w * q_h), \tag{11.1}$$

where $q_w * q_h$ is the matrix multiplication (or convolution) calculated with integers in low precision (e.g., INT4) and accumulated in INT32 precision. This result is then requantized and sent to the next layer as follows:

$$q_a = \text{Int}\left(\frac{a}{S_a}\right) = \text{Int}\left(\frac{S_w S_h}{S_a}(q_w * q_h)\right), \tag{11.2}$$

where S_a is the pre-calculated scale factor for the output activation.

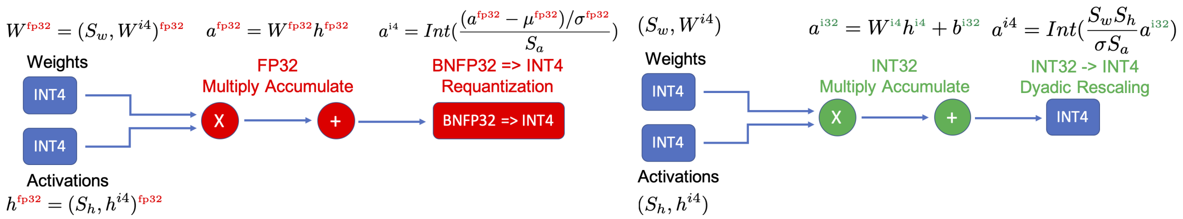

$W^{\text{fp32}} = (S_w, W^{i4})^{\text{fp32}}$
Weights
INT4
INT4
Activations
$h^{\text{fp32}} = (S_h, h^{i4})^{\text{fp32}}$
$a^{\text{fp32}} = W^{\text{fp32}} h^{\text{fp32}}$
FP32
Multiply Accumulate
X
+
$a^{\text{i4}} = Int(\frac{(a^{\text{fp32}} - \mu^{\text{fp32}})/\sigma^{\text{fp32}}}{S_a})$
BNFP32 => INT4
Requantization
BNFP32 => INT4
(S_w, W^{i4})
Weights
INT4
INT4
Activations
(S_h, h^{i4})
$a^{\text{i32}} = W^{\text{i4}} h^{\text{i4}} + b^{\text{i32}}$
INT32
Multiply Accumulate
X
+
$a^{i4} = Int(\frac{S_w S_h}{\sigma S_a} a^{\text{i32}})$
INT32 -> INT4
Dyadic Rescaling
INT4

Figure 11.1: Illustration of fake vs true quantization for convolution and batch normalization folding. For simplicity, we ignore the affine coefficient of BN. (Left) In the simulated quantization (aka fake quantization approach), weights and activations are simulated as integers with floating point representation, and all the multiplication and accumulation happen in FP32 precision. Furthermore, the BN parameters (i.e. μ and σ) are stored and computed using FP32 precision. This is undesirable but can significantly help accuracy since BN parameters are sensitive to quantization. However, with this approach, one cannot benefit from low-precision ALUs. (Right) An illustration of the integer-only pipeline with dyadic arithmetic for convolution and BN folding. The standard deviation (σ) in BN is merged into the quantization scale of the weights, and the mean is quantized to INT32 and merged as a bias into the weights (denoted by b^{i32}). Note that with this approach, all the weights and activations are stored in integer format, and all the multiplications are performed with INT4 and accumulated in INT32 precision. Finally, the accumulated result is requantized to INT4 with dyadic scaling (denoted by $\frac{S_w S_h}{\sigma S_a}$). Importantly, no floating point or even integer division is performed.

In HAWQV3, the $q_w * q_h$ operation is performed with low-precision integer-only multiplication and INT32 accumulation, and the final INT32 result is quantized by scaling it with $S_w S_h / S_a$. The latter is a floating point scaling that needs to be multiplied with the accumulated result (in INT32 precision). A naive implementation requires floating point multiplication for this stage. However, this can be avoided by enforcing the scaling to be a dyadic number. Dyadic numbers are rational numbers with the format of $b/2^c$, where b, c are two integer numbers. As such, a dyadic scaling in Eq. 11.2 can be efficiently performed using INT32 integer multiplication and bit shifting. Given a specific $S_w S_h / S_a$, we use DN (representing Dyadic Number) to denote the function that can calculate the corresponding b and c:

$$b/2^c = \text{DN}\left(S_w S_h / S_a\right). \tag{11.3}$$

An advantage of using dyadic numbers besides avoiding floating point arithmetic is that it removes the need to support division (which typically has an order of magnitude higher latency than multiplication) in the hardware. This approach is used for INT8 quantization in [107], and we enforce all the rescaling to be dyadic for low-precision and mixed-precision quantization as well.

Batch Normalization

Batch normalization (BN) is an important component of most NN architectures, especially for computer vision applications. BN performs the following operation to an input activation a:

$$\mathrm{BN}(a) = \beta \frac{a - \mu_B}{\sigma_B} + \gamma \tag{11.4}$$

where μ_B and σ_B are the mean and standard deviation of a, and β, γ are trainable parameters. During inference, these parameters (both statistics and trainable parameters) are fixed, and therefore the BN operations could be fused with the previous convolution layer. That is to say, we can combine BN and CONV into one operator as,

$$\begin{aligned} \mathrm{CONV\_BN}(h) &= \beta \frac{Wh - \mu}{\sigma} + \gamma \\ &= \frac{\beta W}{\sigma} h + (\gamma - \frac{\beta \mu}{\sigma}) \equiv \bar{W} h + \bar{b}, \end{aligned} \tag{11.5}$$

where W is the weight parameter of the convolution layer and h is the input feature map. In HAWQV3, we use the fused BN and CONV layer and quantize $\bar{W}$ to 4-bit or 8-bit based on the setting, and quantize the bias term, $\bar{b}$ to 32-bit. More importantly, suppose the scaling factor of h is S_h and the scaling factor of $\bar{W}$ is $S_{\bar{W}}$. The scaling factor of $\bar{b}$ is enforced to be

$$S_{\bar{b}} = S_h S_{\bar{W}}. \tag{11.6}$$

So that the integer components of $\bar{W}h$ and $\bar{b}$ can be directly added during inference.

However, an important problem is that quantizing the BN parameters often leads to significant accuracy degradation. As such, many prior quantization methods keep BN parameters in FP32 precision (e.g., [52, 21, 32, 33, 277, 187]). This makes such approaches not suitable for integer-only hardware. While using such techniques help accuracy, HAWQV3 completely avoids that. We fuse the BN parameters with the convolution and quantized them with the integer-only approach (Please see Figure 11.1 where we compare simulated quantization and HAWQV3 for BN and convolution.).

Another important point to discuss here is that we found the BN folding used in [107] to be sub-optimal. In their approach, BN and CONV layers are fused together while BN running statistics are still kept updating. This actually requires computing each convolution layer twice, once without BN and then with BN (as illustrated in [107]). However, we found that this is unnecessary and degrades the accuracy. Instead, in HAWQV3, we follow a simpler approach where we first keep the Conv and BN layer unfolded, and allow the BN statistics to update. After several epochs, we then freeze the running statistics in the BN layer and fold the CONV and BN layers. As we will show in Section 11.2, this approach results in better accuracy as compared to [107].

Residual Connection

Residual connection [87] is another important component in many NN architectures. Similar to BN, quantizing the residual connections can lead to accuracy degradation, and as such,

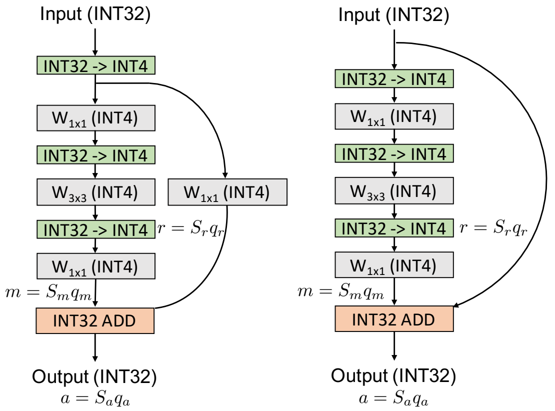


Figure 11.2: Illustration of HAWQV3 for a residual block with and without transition layer. Input feature map is given in INT32 precision, which is requantized to INT4 precision (green boxes) before any convolution layer (gray boxes). The BN layer is folded into the convolution. The residual addition is performed in INT32 precision, and the final accumulated result is re-scaled and sent to the next layer. For blocks with a transition layer, we only quantize the input once to INT4 and we use the same result for both 1 × 1 convolutions.

some prior quantization works perform the operation in FP32 precision [33, 277, 236]. However, quantization is not a linear operation, that is $Q(a+b) \neq Q(a)+Q(b)$ (a, b are floating point numbers). As such, performing the accumulation in FP32 and then quantizing is not the same as accumulating quantized values. We avoid this in HAWQV3, and use INT32 for the residual branch. We perform the following steps to ensure that the addition operation can happen with dyadic arithmetic. Let us denote the activation passing through the residual connection as $r = S_r q_r$.[1] Furthermore, let us denote the activation of the main branch before residual addition as $m = S_m q_m$, and the final output after residual accumulation by $a = S_a q_a$. Then we will have:

$$q_a = \mathrm{DN}\left(S_m/S_a\right) q_m + \mathrm{DN}\left(S_r/S_a\right) q_r. \tag{11.7}$$

Note that with this approach, we only need to perform a dyadic scaling of q_m and add the result with the dyadically scaled q_r. All of these operations can happen with integer-only arithmetic. These steps are schematically illustrated in Figure 11.2 for a residual connection with/without downsampling. Similar approach is performed for the concatenation layer as well.

Mixed Precision and Integer Linear Programming

Uniformly quantizing all the layers to low bit-width (e.g. INT4) could lead to significant accuracy degradation. However, it is possible to benefit from low-precision quantization

[1] This is either the input or the output activation after the downsampling layer.

by keeping a subset of sensitive layers at high precision [53]. An important component of HAWQV3 is that we directly consider hardware-specific metrics such as latency, to select the bit-precision configuration. This is important since a layer's latency does not necessarily halve when quantized from INT8 to INT4 precision. In fact, as we discuss in Section 11.2, there are specific layer configurations that do not gain any speed-up when quantized to low precision, and some that superlinearly benefit from quantization.[2] As such, quantizing the former will not lead to any latency improvement, and will only hurt accuracy. Therefore, it is better to keep such layers at high precision, even if they have low sensitivity. These trade-offs between accuracy and latency should be taken into consideration when quantizing them to low precision. Importantly, these trade-offs are hardware-specific as latency in general does not correlate with the model size and/or FLOPS. However, we can consider this by directly measuring the latency of executing a layer in quantized precision on the target hardware platform. This trade-off is schematically shown in Figure 4.5 (and later quantified in Figure 11.3). We can use an Integer Linear Programming (ILP) problem to formalize the problem definition of finding the bit-precision setting that has an optimal trade-off.

Assume that we have B choices for quantizing each layer (i.e., 2 for INT4 or INT8). For a model with L layers, the search space of the ILP will be B^L. The goal of solving the ILP problem is to find the best bit configuration among these B^L possibilities that result in optimal trade-offs between model perturbation Ω, and user-specified constraints such as model size, BOPS, and latency. Each of these bit-precision settings could result in a different model perturbation. To make the problem tractable, we assume that the perturbations for each layer are independent of each other (i.e., $\Omega = \sum_{i=1}^{L} \Omega_i^{(b_i)}$, where $\Omega_i^{(b_i)}$ is the i-th layer's perturbation with b_i bit)[3]. This allows us to precompute the sensitivity of each layer separately, and it only requires BL computations. For the sensitivity metric, we use the Hessian-based perturbation proposed in [52]. The ILP problem tries to find the right bit precision that minimizes this sensitivity, as follows:

$$\text{Objective: } \min_{\{b_i\}_{i=1}^{L}} \sum\nolimits_{i=1}^{L} \Omega_i^{(b_i)}, \tag{11.8}$$

$$\text{Subject to: } \sum\nolimits_{i=1}^{L} M_i^{(b_i)} \leq \text{ Model Size Limit}, \tag{11.9}$$

$$\sum\nolimits_{i=1}^{L} G_i^{(b_i)} \leq \text{ BOPS Limit}, \tag{11.10}$$

$$\sum\nolimits_{i=1}^{L} Q_i^{(b_i)} \leq \text{ Latency Limit}. \tag{11.11}$$

Here, $M_i^{(b_i)}$ denotes the size of i-th layer with b_i bit quantization, $Q_i^{(b_i)}$ is the associated latency, and $G_i^{(b_i)}$ is the corresponding BOPS required for computing that layer. The latter measures the total Bit Operations for calculating a layer [7]:

$$G_i^{(b_i)} = b_{w_i} b_{a_i} \text{MAC}_i,$$

[2] The speedup of each layer is calculated by the latency of INT8 divided by that of INT4. For uniform 4-bit and mixed-precision models, the speedup is calculated related to the uniform 8-bit model.

[3] Similar assumption can be found in [53, 52].

where MAC_i is the total Multiply-Accumulate operations for computing the i-th layer, and b_{w_i}, b_{a_i} are the bit precision used for weight and activation.[4] Note that it is not necessary to set all these constraints at the same time. Typically, which constraint to use depends on the end-user application.

We solve the ILP using open source PULP library [201] in Python, where we found that for all the configurations tested in the paper, the ILP solver can find the solution in less than 1 second given the sensitivity metric. For comparison, the RL-based method of [236] could take tens of hours to find the right bit-precision setting. Meanwhile, as can be seen, our ILP solver can be easily used for multiple constraints. We should also mention that the contemporary work of [104] also proposed an ILP formulation. However, our approach is hardware-aware and we directly deploy and measure the latency of each layer in hardware.

Hardware Deployment

Model size alone is not a good metric to measure the efficiency (speed and energy consumption) of NNs. In fact, it is quite possible that a small model would have higher latency and consume a larger amount of energy for inference. The same is also true for FLOPs. The reason is that neither model size nor FLOPs can account for cache misses, data locality, memory bandwidth, underutilization of hardware, etc. To address this, we need to deploy and directly measure the latency.

We target Nvidia Turing Tensor Cores of T4 GPU for deployment, as it supports both INT8 and INT4 precision and has been enhanced for deep learning network inference. The only API available is the WMMA kernel call which is a micro-kernel for performing matrix-matrix operations in INT4 precision on Tensor Cores. However, there is also no existing compiler that would map a NN quantized to INT4 to Tensor Cores using WMMA instructions. To address this challenge, another contribution of our work is extending TVM [27] to support INT4 inference with/without mixed precision with INT8. This is important so we can verify the speed benefits of mixed-precision inference. To accomplish this, we had to add new features in both graph-level IR and operator schedules to make INT4 inference efficient. For instance, when we perform optimizations such as memory planning, constant folding, and operator fusion, at the graph-level IR, 4-bit data are involved. However, on byte-addressable machines, manipulating 4-bit data individually leads to inefficiency in storage and communication. Instead, we pack eight 4-bit elements into an INT32 data type and perform the memory movement as a chunk. In the final code generation stage, the data type and all memory access will be adjusted for INT32. By adopting similar scheduling strategies to Cutlass [185], we implement a new direct convolution schedule for Tensor Cores for both 8-bit and 4-bit data in TVM. We set the knobs for the configurations such as thread size, block size, and loop ordering so that the auto-tuner in TVM could search for the best latency settings.

[4] b_{w_i} and b_{a_i} are always the same in HAWQV3. As such, HAWQV3 does not need to cast lower-precision integer numbers, e.g., INT4, to higher-precision integer numbers, e.g., INT8, which is more efficient than [52, 21, 236].

Another important point is that we have completed the pipeline to test directly the trained weights and to avoid using random weights for speed measurements. This is important, since small discrepancies between the hardware implementation may go unnoticed from the quantization algorithm in the NN training framework (PyTorch in our case) which does not use TVM for the forward and backward propagation. To avoid any such issue, we made sure that the results between TVM and PyTorch match for every single layer and stage to machine-precision accuracy, and we verified the final Top-1 accuracy when executed in the hardware with integer-only arithmetic.

11.2 Experiments

Low Precision Integer-Only Quantization Results

We first start with ResNet18/50 and InceptionV3 quantization on ImageNet, and compare the performance of HAWQV3 with other approaches, as shown in Table 11.1.

Uniform 8-bit Quantization. Our 8-bit quantization achieves similar accuracy compared to the baseline. Importantly, for all the models HAWQV3 achieves higher accuracy than the integer-only approach of [107]. For instance, on ResNet50, we achieve 2.68% higher accuracy as compared to [107]. This is in part due to our BN folding strategy that was described in Section 11.1.

Uniform 4-bit Quantization. The accuracy results for ResNet18/50, and InceptionV3 are quite high, despite the fact that all of the inference computations are restricted to be integer multiplication, addition, and bit shifting. While there is some accuracy drop, this should not be incorrectly interpreted that uniform INT4 is not useful. On the contrary, one has to keep in mind that certain use cases have strict latency and memory footprint limits for which this may be the best solution. However, higher accuracy can be achieved through mixed-precision quantization.

Mixed 4/8-bit Quantization. The mixed-precision results improve the accuracy by several percentages for all the models, while slightly increasing the memory footprint of the model. For instance, the mixed-precision result for ResNet18 is 1.88% higher than its INT4 counterpart with just a 1.9MB increase in model size. Further improvements are also possible with distillation (denoted as HAWQV3+Dist in the table). For ResNet50, the distillation can boost the mixed-precision by 1.34%. We found that distillation helps most for mixed-precision quantization, and we found little to no improvement for uniform INT8, or uniform INT4 quantization cases.[5]

Overall, the results show that HAWQV3 achieves comparable accuracy to prior quantization methods including both uniform and mixed-precision quantization (e.g., PACT, RVQuant, OneBitwidth, HAQ which use FP32 arithmetic, and/or non-standard bit preci-

[5] We used simple distillation without extensive tuning. One might be able to improve the results further with more sophisticated distillation algorithms.

Table 11.1: Quantization results for ResNet18/50 and InceptionV3. Here, we abbreviate Integer-Only Quantization as "Int", Uniform Quantization as "Uni", the Baseline Accuracy as ”BL”, Weight Precision and Activation Precision as "Precision", Model Size as "Size" (in MB), Bit Operations as "BOPS" (in G), and Top-1 Accuracy as "Top-1". Also, "WxAy" means weight with x-bit and activation with y-bit, and 4/8 means mixed precision with 4 and 8 bits. "MP" means mixed precision with bitwidth ranging from 1-bit to 8-bit, and "W1*" means the bitwidth is 1-bit but the network architecture is changed (by using more channels). Our result with/without distillation is represented as HAWQV3+Dist/HAWQV3.

(a) ResNet18

Method	Int	Uni	BL	Precision	Size	BOPS	Top-1
Baseline	✓	–	71.47	W32A32	44.6	1858	71.47
RVQuant [187]	✓	✓	69.91	W8A8	11.1	116	70.01
HAWQV3	✗	✗	71.47	W8A8	11.1	116	**71.56**
PACT [33]	✓	✗	70.20	W5A5	7.2	50	69.80
LQ-Nets [277]	✓	✓	70.30	W4A32	5.8	225	70.00
HAWQV3	✗	✗	71.47	W4/8A4/8	6.7	72	70.22
HAWQV3+Dist	✗	✗	71.47	W4/8A4/8	6.7	72	**70.38**
CalibTIB[104]	✓	✗	71.97	W4A4	5.8	34	67.50
HAWQV3	✗	✗	71.47	W4A4	5.8	34	**68.45**

(b) ResNet50

Method	Int	Uni	BL	Precision	Size	BOPS	Top-1
Baseline	✗	✗	77.72	W32A32	97.8	3951	77.72
Integer Only [107]	✗	✗	76.40	W8A8	24.5	247	74.90
RVQuant [187]	✓	✓	75.92	W8A8	24.5	247	75.67
HAWQV3	✗	✗	77.72	W8A8	24.5	247	**77.58**
PACT [33]	✓	✗	76.90	W5A5	16.0	101	76.70
LQ-Nets [277]	✓	✓	76.50	W4A32	13.1	486	76.40
RVQuant [187]	✓	✓	75.92	W5A5	16.0	101	75.60
HAQ [236]	✓	✓	76.15	WMPA32	9.62	520	75.48
OneBitwidth [32]	✓	✗	76.70	W1*A8	12.3	494	76.70
HAWQV3	✗	✗	77.72	W4/8A4/8	18.7	154	75.39
HAWQV3+Dist	✗	✗	77.72	W4/8A4/8	18.7	154	**76.73**
CalibTIB[104]	✓	✗	77.20	W4A4	13.1	67	73.70
HAWQV3	✗	✗	77.72	W4A4	13.1	67	**74.24**

(c) InceptionV3

Method	Int	Uni	BL	Precision	Size	BOPS	Top-1
Baseline	✓	✗	78.88	W32A32	90.9	5850	78.88
Integer Only [107]	✗	✗	78.30	W8A8	22.7	366	74.20
RVQuant [187]	✓	✓	74.19	W8A8	22.7	366	74.22
HAWQV3	✗	✗	78.88	W8A8	22.7	366	**78.76**
Integer Only [107]	✗	✗	78.30	W7A7	20.1	280	73.70
HAWQV3	✗	✗	78.88	W4/8A4/8	19.6	265	74.65
HAWQV3+Dist	✗	✗	78.88	W4/8A4/8	19.6	265	**74.72**
HAWQV3	✗	✗	78.88	W4A4	12.3	92	70.39

sion such as 5 bits, or different bit-width for weights and activations). Similar observations hold for InceptionV3, as reported in Table 11.1.

Table 11.2: Mixed-precision quantization results for ResNet18 and ResNet50 with different constraints. Here, we abbreviate constraint level as "Level". Model Size as "Size", Bit Operations as "BOPS", the speedup as compared to INT8 results as "Speed", and Top-1 Accuracy as "Top-1", The last column of Top-1 represents the results of HAWQV3 and HAWQV3+Dist. Note that for uniform INT8 ResNet50 (ResNet18), the latency is 1.06ms (0.40ms) per images.

(a) ResNet18

	Level	Size (MB)	BOPS (G)	Speed	Top-1
INT8	–	11.2	114	1x	71.56
Size	High	**9.9**	103	1.03x	71.20/71.59
	Medium	**7.9**	98	1.06x	70.50/71.09
	Low	**7.3**	95	1.08x	70.01/70.66
BOPS	High	8.7	**92**	1.12x	70.40/71.05
	Medium	6.7	**72**	1.21x	70.22/70.38
	Low	6.1	**54**	1.35x	68.72/69.72
Latency	High	8.7	92	**1.12x**	70.40/71.05
	Medium	7.2	76	**1.19x**	70.34/70.55
	Low	6.1	54	**1.35x**	68.56/69.72
INT4	–	5.6	28	1.48x	68.45

(b) ResNet50

	Level	Size (MB)	BOPS (G)	Speed	Top-1
INT8	–	24.5	247	1x	77.58
Size	High	**21.3**	226	1.09x	77.38/ 77.58
	Medium	**19.0**	197	1.13x	75.95/76.96
	Low	**16.0**	168	1.18x	74.89/76.51
BOPS	High	22.0	**197**	1.16x	76.10/76.76
	Medium	18.7	**154**	1.23x	75.39/76.73
	Low	16.7	**110**	1.30x	74.45/76.03
Latency	High	22.3	199	**1.13x**	76.63/76.97
	Medium	18.5	155	**1.21x**	74.95/76.39
	Low	16.5	114	**1.28x**	74.26/76.19
INT4	–	13.1	67	1.45x	74.24

Mixed-precision Results with Different Constraints

We consider three different thresholds for each of the constraints and study how the ILP balances the trade-offs to obtain an optimal quantized model. We also focus on the case where the practitioner is not satisfied with the performance of the INT4 quantization and wants to improve the performance (accuracy, speed, and model size) through mixed-precision quantization (INT4 and INT8). The ILP formulation enables the practitioner to set each or all of these constraints. Here, we present results when only one constraint is set at a time.

We start with the model size and BOPS constraints for ResNet18. The model size of pure INT4 quantization is 5.6MB, and INT8 is 11.2MB. However, the accuracy of INT4 quantization is 68.45% which may be low for a particular application. The practitioner then has the option to set the model size constraint to be slightly higher than pure INT4. One option is to choose 7.9MB which is almost in between INT4 and INT8. For this case, the ILP solver finds a bit-precision setting that results in 71.09% accuracy which is almost the same as INT8. This model is also 6% faster than INT8 quantization.

Another possibility is to set the speed/latency as a constraint. The results for this setting are represented under the "Latency" row in Table 11.2. For example, the practitioner could request the ILP to find a bit-precision setting that would result in 19% faster latency as compared to the INT8 model (see "Medium" row). This results in a model with an accuracy of 70.55% with a model size of only 7.2MB. A similar constraint can also be made for BOPS.

Several very interesting observations can be made from these results. (i) The correlation between model size and BOPS is weak which is expected. That is, a larger model size does not mean higher BOPS and vice versa. For example, compare Medium-Size and High-BOPS for ResNet18. The latter has lower BOPS despite being larger (and is actually faster as well). (ii) The model size does not directly correlate with accuracy. For example, for ResNet50, High-BOPS has a model size of 22MB and an accuracy of 76.76%, while High-Size has a smaller model size of 21.3MB but higher accuracy of 77.58%.

In summary, although directly using INT4 quantization may result in large accuracy degradation, we can achieve significantly improved accuracy with much faster inference as compared to INT8 results. This gives the practitioner a wider range of choices beyond INT8 quantization. Finally, note that the accuracy and speed for all the results have been verified by directly measuring them when executed in quantized precision in hardware through TVM. As such, these results are what the practitioner will observe, and are not simulated results.

ILP Result Interpolation

We plot the bit-precision setting for each layer of ResNet18 that the ILP solver finds for different latency constraints. Additionally, we also plot the sensitivity (Ω_i in Eq. 14.5) and the corresponding speed up for each layer computed by quantizing the respective layer in INT8 quantization versus INT4. As can be seen, the bit configuration chosen by the ILP solver is highly intuitive based on the latency speed-up and the sensitivity. Particularly, when the mixed-precision model is constrained by the High-Latency setting (the first row

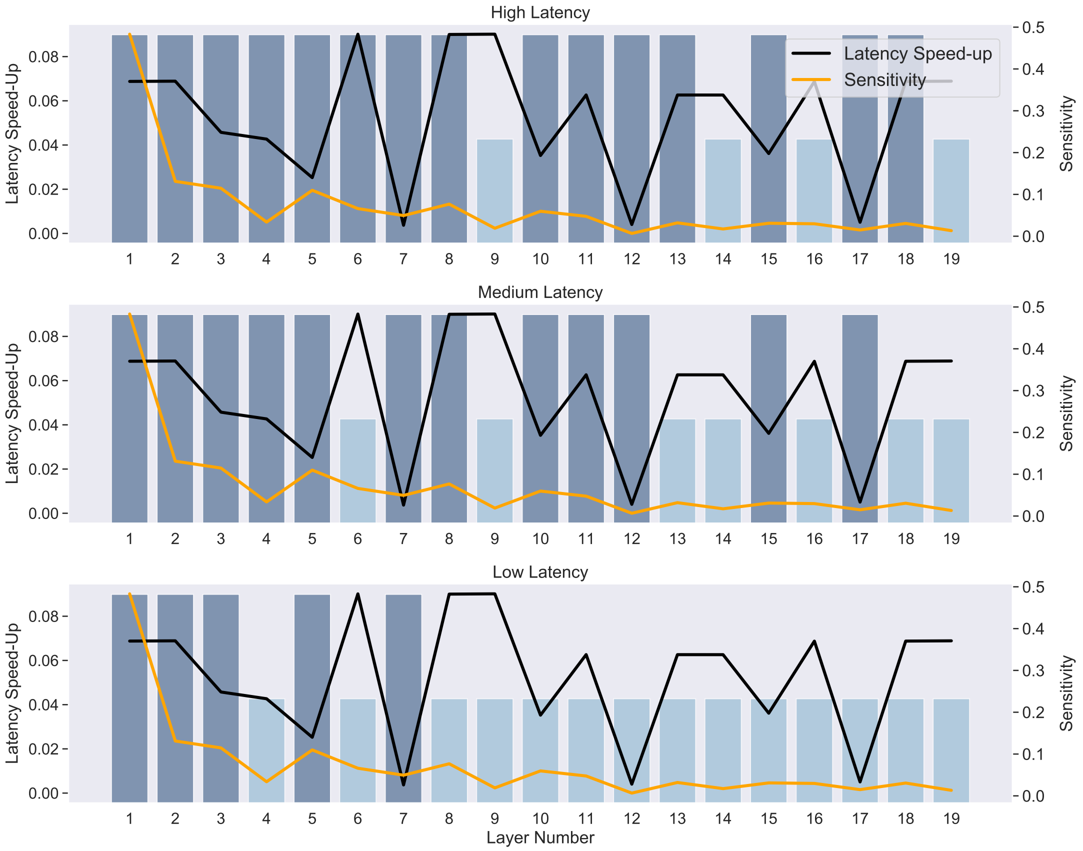


Figure 11.3: Illustration of the final model specification that the ILP solver finds for ResNet18 with latency constraint. The black line shows the percentage of latency reduction for a layer executed in INT4 versus INT8, normalized by total inference reduction. Higher values mean higher speedup with INT4. The orange line shows the sensitivity difference between INT8 and INT4 quantization using second order Hessian sensitivity [52]. The bit-precision setting found by ILP is shown in bar plots, with the blue and taller bars denoting INT8, and cyan and shorter bars denoting INT4. Each row corresponds to the three results presented in Table 11.2 with latency constraints. For the low latency constraint, the ILP solver favors assigning INT4 for layers that exhibit large gains in latency when executed in INT4 (higher values in the dark plot) and that have low sensitivity (lower values in the orange plot).

of Figure 11.3), only relatively insensitive layers, along with those that enjoy high INT4 speed-up, are quantized (i.e., layers 9, 14, and 19). However, for the more strict Low-Latency setting (last row of Figure 11.3), only very sensitive layers are kept at INT8 precision (layer 1, 2, 3, 5, and 7).[6]

[6] Note that here layer 7 is the downsampling layer along with layer 5, so it is in the same bit setting as layer 5 even though the latency gain of layer 7 is limited.

Chapter 12

HW-SW Co-Design: CoDeNet

In HAWQV3, only the quantization settings can be adjusted in a hardware-aware manner, while the hardware specifications and the neural architectures are fixed. In this Chapter, we try to include more design space into the HW-SW Co-Design pipeline, and we focus on answering the *key question*:

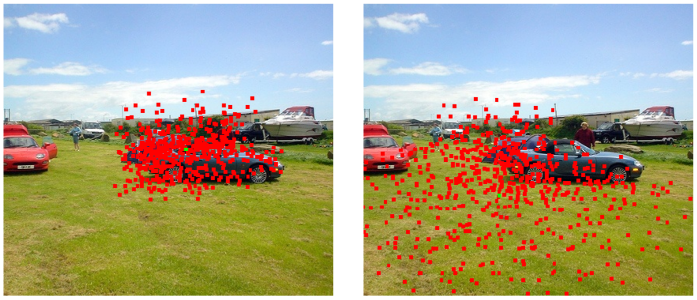
> How much improvement can be obtained by jointly co-designing neural architectures, quantization settings, and hardware configurations?

We take neural architectures with deformable convolution on object detection tasks as an example to perform HW-SW co-design. Our proposed CoDeNet can run real-time detection with decent accuracy on customized FPGA accelerators.

12.1 Introduction

Introduction to Deformable Convolution

Compared to image classification, one challenge in object detection is to capture geometric variations of each object, such as scale, pose, viewpoint, and part deformation. Besides, different objects located in different regions of the same image can be geometrically different, making it hard to capture all features in one pass. State-of-the-art approaches [30][136][152][208][289] address these challenges by harnessing deformable convolution [40][292]. As demonstrated in Figure 12.2, deformable convolution samples the input feature map using the offsets dynamically predicted from the same input feature map, after which it performs a regular convolution over the features sampled from the predicted offsets. The convolution layer for generating the offsets is typically composed of one 1×1 or 3×3 convolution layer. It is jointly trained with the rest of the network using standard backpropagation in an end-to-end manner. This way the gradient updates not only the weights of the convolutions but also the sampling locations for the convolutions. Such operation design enables more flexible and adaptive sampling on different input feature maps.



(a) car (b) lawn

Figure 12.1: Example for the input-adaptive deformable convolution sampling locations and offset range distribution for different active detection units. (a) the sampling locations for the car as an active unit. (b) the sampling locations for the lawn in the background.

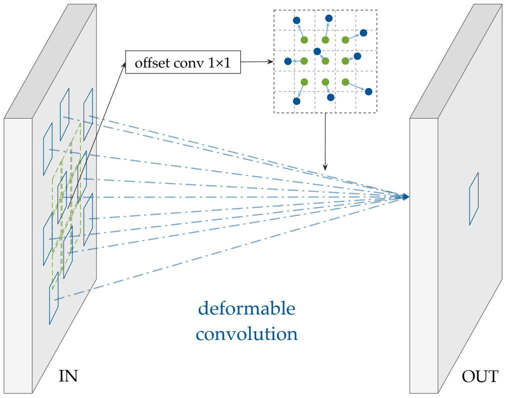


Figure 12.2: Deformable convolution with input-adaptive offsets generation. Deformable convolution in our design first generates the sampling offsets from the input feature map using a 1×1 convolution. Then it samples the same input feature map based on the generated offsets and performs a 3×3 convolution to aggregate the corresponding spatial features.

Unlike the regular convolution with fixed geometry, the receptive fields of deformable convolution can be of various shapes to capture objects with different scales, aspect ratios, and rotation angles. In addition, deformable convolution is both spatial-variant and input-adaptive. In other words, its sampling patterns and offsets vary for different output pixels in the same input feature map and also vary across different input feature maps. In Figure 12.1(a)(b), we show how the sampling locations (red dots) change with the different active detection units (the object with a green dot on it). Most of the offsets are within the $[-1, 4]$ range for the example image. Albeit the operation augments and enhances the capability of the existing convolution for object detection, its dynamic nature poses extra challenges to the existing hardware.

Algorithm-hardware Co-design for Object Detection

Many prior acceleration works [293, 163, 180, 83, 280, 256, 247, 101] have demonstrated the effectiveness of the co-design methodology for the deployment of real-time object detection on FPGAs. [163] customizes SSD300 [153] by replacing operations, such as dilated convolutions, normalization, and convolutions with larger strides, with more efficiently supported ones on FPGAs. [180] adapts YOLOv2 [197] by introducing a binarized network as the backbone for feature extraction to leverage the low-precision support of FPGA. Meanwhile, the FINN-R framework [17] further explores the benefits of integrating quantized neural networks (QNN) into Yolo-based object detection systems. Real-time object detection for live video streaming system [190] is then developed with the FINN-based QNNs. [83] devised an automatic co-design flow on embedded FPGAs for the DJI-UAV [257] dataset with 95 categories targeting unmanned aerial vehicles. The flow first constructs DNN basic building blocks called bundles, estimates their corresponding latency and cost on hardware, and selects the ones on the Pareto frontier for latency and resources trade-off. Then it starts a two-phase DNN evaluation to search for the bundles on the Pareto frontier of the accuracy-latency trade-off and then fine-tune the design of the selected bundles. SkyNet [280] searched by this co-design flow achieves the best performance (based on a combination of throughput, power, and detection accuracy) on embedded GPUs and FPGAs.

12.2 Method

Deformable Operation Co-design

Although deformable convolution augments the neural network design with input-adaptive sampling, it is challenging to provide efficient support for the operation in its original form on hardware accelerators due to the following reasons:

1. the limited reuse of input features
2. the irregular input-dependent memory access patterns
3. the computation overhead from the bilinear interpolation
4. the memory overhead of the deformable offsets

In this work, we perform a series of modifications to deformable convolution with the objective to enable more data reuse and a higher degree of parallelism for FPGA acceleration. A comprehensive ablation study is done to demonstrate the impact of each algorithmic modification on accuracy. We perform our study with standard object detection benchmarks, VOC, and COCO. We then design a specialized hardware engine optimized for each algorithmic modification on FPGA and show the performance improvement on FPGA from each modification. The accuracy and hardware efficiency trade-off is studied for each modification

Table 12.1: Ablation study of operation choices for object detection on VOC and COCO. The top half shows the baselines with various kernel sizes, from 3×3 to 9×9. The bottom half shows the comparison of different designs for deformable convolution.

Operation	**Depthwise**	**Bound**	**Square**	**VOC**			**COCO**					
				AP	**AP50**	**AP75**	**AP**	**AP50**	**AP75**	**APs**	**APm**	**APl**
3×3				39.2	60.8	41.2	21.4	36.5	21.5	7.3	24.1	33.0
3×3	✓			39.1	60.9	40.9	19.8	34.3	19.7	6.3	22.6	31.5
5×5	✓			40.6	62.4	42.6	21.3	36.4	21.3	6.7	23.7	34.2
7×7	✓			41.9	63.8	43.8	21.7	37.2	21.5	6.9	24.0	35.2
9×9	✓			42.3	64.8	44.3	22.2	37.8	22.1	7.0	24.3	35.4
deform	✓			42.9	64.4	45.7	23.0	38.4	23.3	6.9	24.4	37.8
deform	✓	✓		41.0	63.0	42.9	21.3	36.4	21.1	7.2	23.6	34.4
deform	✓	✓	✓	41.1	63.1	43.7	21.5	36.8	21.5	6.5	23.7	34.8

we propose. We will be using the following notations: n - batch size, h - height, w - width, ic - input channel size, oc - output channel size, k - kernel size, Δp - offsets.

Algorithm Modifications

We choose average precision (AP) as the main metric for benchmarking object detection performance on VOC and COCO datasets. ShuffleNet V2 [160] is used as the feature extractor in all experiments. As for the decoder, we follow the practice of CenterNet [289] and use the stack of deformable convolution, nearest 2× upsample, and ReLU activation layers. Table 12.1 lists the modifications we make to the original deformable convolution as well as a comparison among deformable convolutions of different forms and regular convolutions with different kernel sizes. From the comparison, we see that the original deformable convolution achieves higher accuracy on Pascal VOC compared to convolution with 9×9 kernel (42.9 vs 42.3) while requiring $\frac{9\times9}{3\times3} = 9\times$ fewer MACs and weight parameters. Here we discuss how we further improve the efficiency of deformable convolution for hardware step-by-step.

Depthwise Convolution We first replace the full 3×3 deformable convolutions with 3×3 depthwise deformable convolutions and 1×1 convolutions, similar to the depthwise separable convolution practice in Xception [36]. Such modification makes the whole network more uniform and smaller, so the weights of the deformable convolution can be all buffered on-chip for maximal reuse.

Bounded Range Our next algorithmic modification to facilitate efficient hardware acceleration is to restrict the offsets to a positive range. Such constraint limits the size of the working set of feature maps so that a pre-defined fixed-size buffer can be added to the hardware, in order to further exploit the temporal and spatial locality of the inputs. Assume a uniform distribution for the generated offsets in a 3×3 convolution kernel with stride 1, each pixel is expected to be used nine times. If all inputs within the range can be stored in the buffer, all except the first access to the same address will be from on-chip memory with $1 \sim 3$ cycle latency. We impose this constraint during training by adding a *clipping* operation after the offset generation layer to truncate offsets that are smaller than 0 or larger

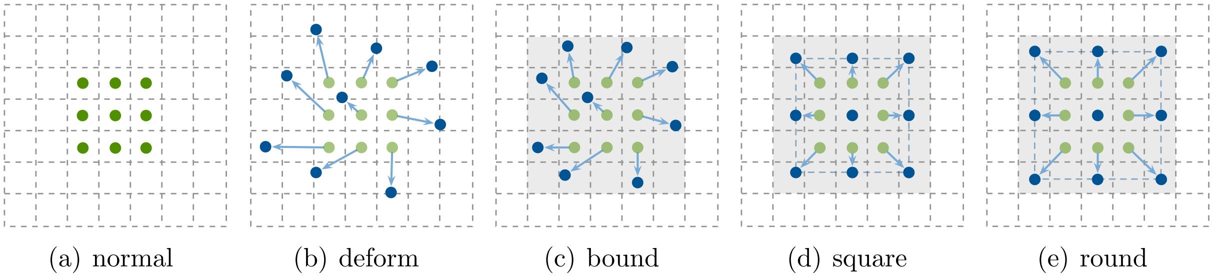


Figure 12.3: Major algorithm modifications for deformable convolution operational co-design. (a) is the default 3×3 convolutional filter. (b) is the original deformable convolution with unconstrained non-integer offsets. (c) sets an upper bound to the offsets. (d) limits the geometry to a square shape. (e) shows that the predicted offsets are rounded to integers.

than N, so all offsets $\Delta p_x, \Delta p_y \in [0, N]$. Table 12.1 shows that setting the bound N to 7 results in 1.9 and 1.7 AP degradation on VOC and COCO respectively.

Square Shape Another obstacle to efficiently supporting the deformable convolution is its irregular data access patterns, which leads to serialized memory accesses to multi-banked on-chip memory. To address this issue, we further constrain the offsets to be on the edges of a square. Instead of using $3 \times 3 \times 2 = 18$ numbers to represent the Δp_x and Δp_y offsets for all nine samples, only one number Δp_d, representing the distance from the center to the sides of the square, needs to be learned. This is similar to a dilated convolution with spatial-variant adaptive dilation factors. Adding this modification leads to a 0.1 and 0.2 AP increase on VOC and COCO.

Rounded Offsets In the original deformable design, the generated offsets are typically fractional and a bilinear interpolation needs to be performed to produce the target sampling value. Bilinear interpolation calculates a weighted average of the neighboring pixels for a fractional offset based on its distance to the neighboring pixels. It introduces at least six multiplications to the sampling process of each input, which is a significant increase $(6 \times h \times w \times ic)$ to the total FLOPs. We thus round the offsets to be integers during inference to reduce the total computation. The dynamically-generated offsets are thus rounded to integers. In practice, we round the generated offset during the quantization step.

As shown in Table 12.1, together with the modifications above, our co-designed deformable convolution achieves 41.1 and 21.5 AP on VOC and COCO respectively, which is 1.8 and 1.5 lower than the original depthwise deformable convolution. Note that the accuracy of the modified deformable convolution still achieves higher accuracy compared to the large 5×5 kernel, while requiring $\frac{3\times3}{5\times5} = 36\%$ fewer MACs and parameters.

Hardware Optimizations

Many hardware optimization opportunities are exposed after we perform the aforementioned modifications to deformable convolution. We implement a hardware deformable convolution

Table 12.2: Co-designed hardware performance comparison. The top half shows the performance of co-designed hardware corresponding to each algorithmic change to the default 3×3 convolution. The bottom half shows the results for the depthwise 3×3 convolution.

Operation	**Deform**	**Bound**	**Square**	**Without LLC**		**With LLC**	
				Latency (ms)	GOPs	Latency (ms)	GOPs
default 3×3 conv				43.1	112.0	41.6	116.2
	✓			59.0	81.8	42.7	113.1
	✓	✓		43.4	111.5	41.8	115.5
	✓	✓	✓	43.4	111.5	41.8	115.6
depthwise 3×3 conv				1.9	9.7	2.0	9.6
	✓			20.5	0.9	17.8	1.1
	✓	✓		3.0	6.2	3.4	5.5
	✓	✓	✓	2.1	9.2	2.3	8.2

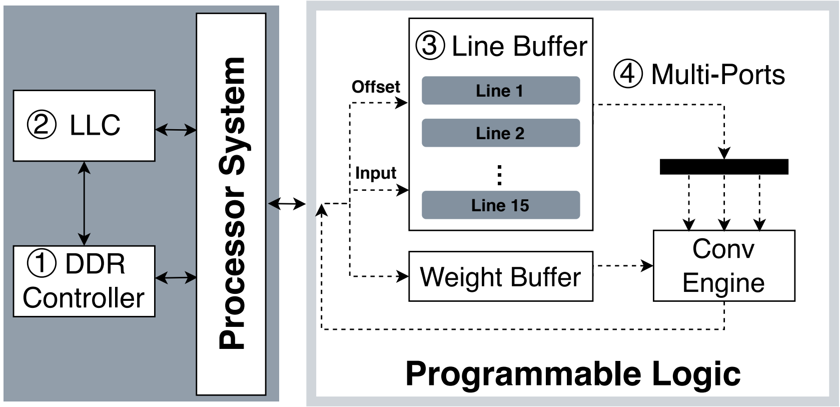


Figure 12.4: Hardware engine for deformable convolution.

engine on FPGA SoC as shown in Figure 12.4 and tailor the hardware engine to each algorithm modification. The experiments are run on the Ultra96 board featuring a Xilinx Zynq XCZU3EG UltraScale+ MPSoC platform. The accelerator logic accesses the 1MB 16-way set-associative LLC through the Accelerator Coherency Port (ACP). The data cache uses a pseudo-random replacement policy. Table 12.2 lists the speed and throughput performance for different customized hardware running a kernel of size $h = 64, w = 64, k = 256, c = 256$. In all experiments, we round the dynamically-generated offsets to integers. We use $8 \times 8 \times 9$ Multiply-Accumulate (MAC) units in the 3×3 convolution engine for all full convolution experiments and 16×9 MACs for depthwise convolution experiments.

Baseline The baseline hardware implementation for the original 3×3 deformable convolution directly accesses the DRAM without going through any cache or buffering. In Figure 12.4, the baseline implementation directly accesses the input and output data through HP ports and ① DDR controller. The input addresses are first calculated from the offsets loaded from DRAM. The 3×3 *Deform M2S* engine then fetches and packs the inputs into parallel data streams to feed into the MAC units in the 3×3 *Conv* engine. This baseline design resembles accelerator designs with only a scratchpad memory that cannot leverage the temporal locality of the dynamically loaded inputs for deformable convolution.

Caching One hardware optimization to leverage the temporal and spatial locality of the nonuniform input accesses is to add a cache to the accelerator system. As shown in Figure 12.4, we load the inputs from ② LLC through the ACP port in this implementation to reduce the memory access latency of the cached values. Since the inputs are sampled from offsets without specific patterns in the original deformable convolution, the cache provides adequate support to buffer inputs that might be reused in the near future. As shown in Table 12.2, adding LLC results in 27.6% and 13.2% reduction in latency for the original full and depthwise deformable convolution respectively.

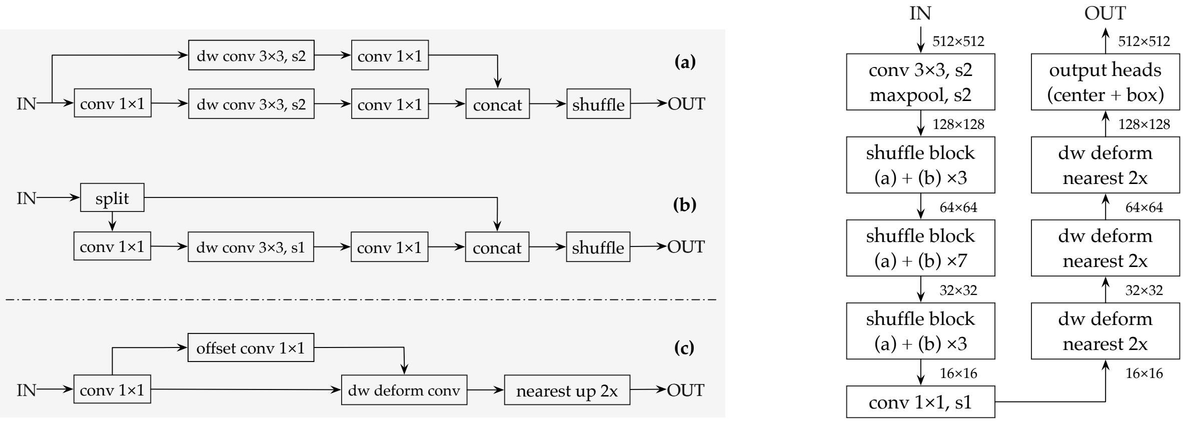
Buffering With the bounded range modification to the algorithm, we are able to use the on-chip memory to buffer all possible inputs. Similar to a line-buffer design for the original 3×3 convolution that stores two lines of inputs to exploit all input locality, we store $2N$ lines of inputs so that it is sufficient to buffer all possible inputs for reuse. This implementation includes the ③ Line Buffer in Figure 12.4. With the effective buffering strategy, we can see in Table 12.2 that the latency of a bounded deformable is reduced by 26.4% and 85.3% for full and depthwise convolution respectively in a system without LLC. In a system with LLC, the reduction is 2.1% and 80.9% respectively. The depthwise deformable convolution benefits more from adding the buffer as it is a more memory-bound operation. The compute-to-communication ratio for its input is oc times lower than the full convolution.

Parallel Ports The algorithm change to enforce a square-shape sampling pattern not only reduces the bandwidth requirements for loading the input indices in hardware, but also helps to improve the on-chip memory bandwidth. With a non-predictable memory access pattern to the on-chip memory, only one input can be loaded from the buffer at each cycle if all sampled inputs are stored in the same line buffer. By constraining the shape of deformable convolution to a square with variable dilation, we are guaranteed to have three different line buffers each storing three sampled points. We can thus have three parallel ports (④ Multi-ports in Figure 12.4) accessing different line buffers concurrently. This co-optimization improves the on-chip memory bandwidth and leads to another $\sim 30\%$ reduction in latency for depthwise deformable convolution.

With the co-design methodology, our final result shows a 1.36× and 9.76× speedup respectively for the full and depthwise deformable convolution on the embedded FPGA accelerator. These optimizations can also be beneficial to other hardware with line buffers and parallel port support.

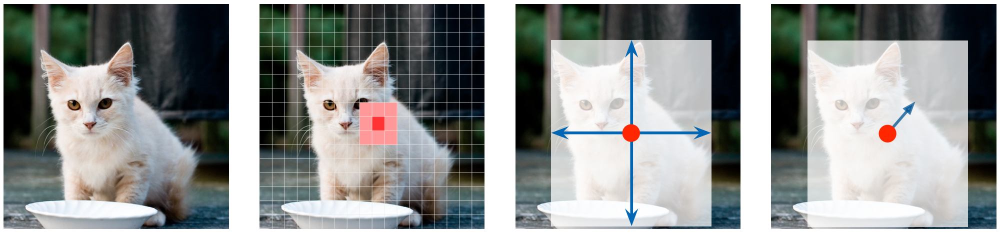
Detection System Co-Design

In addition to the deformable convolution operation, the design of the feature extractor, detection heads, and quantization strategy also significantly impact the accuracy and efficiency of our detection system. In this section, we introduce CoDeNet for an efficient detector and a specialized FPGA accelerator design to support it.



(i) building blocks (ii) model architecture

Figure 12.5: The architecture diagrams of our building blocks and model architecture.



(a) image (b) center heatmap (c) width & height (d) local shift

Figure 12.6: The output heads of CenterNet for object detection.

CoDeNet Design

To exploit the full potential of hardware acceleration, we carefully select and integrate the operations and building blocks in CoDeNet. We devise CoDeNet to have the following embedded hardware compatible properties compared to other off-the-shelf network designs: 1) more uniform operation types to reduce the control complexity in the accelerator and to increase the accelerator utilization, 2) less computation to lower the overall latency to run on the embedded accelerator with limited computing capability, 3) smaller weights and inputs to be buffered on-chip for maximal reuse on the accelerator. Figure 12.5 shows the basic building blocks as well as the overall network architecture of CoDeNet.

Building Blocks and Feature Extractor The shaded part of Figure 12.5 shows the basic building blocks of CoDeNet. Building block (a) is used to downsample the input images. A 3×3 depthwise convolution block with stride 2 is added to both of its branches together with 1×1 convolution to aggregate information across the channel dimension. Building block (b) splits the input features into two streams across the channel dimension. One branch is directly fed to the concatenation. The other streams through a sub-block of 1×1, 3×3 depthwise, and 1×1 convolution. This technique is referred to as identity mapping [88], which is commonly used to address the vanishing gradient problem during deep neural network training. Building blocks (a) and (b) together form a shuffle block as shown in the left branch of the overall architecture in Figure 12.5, as part of the feature extractor ShuffleNetV2. We choose ShuffleNetV2 as it is one of the state-of-the-art efficient network designs. ShuffleNetV2 1x configuration only requires 2.3M parameters (4.8× smaller than ResNet-18 [87]) and 146M FLOPs of computation with resolution 224×224 (12.3x smaller than ResNet-18). Its top-1 accuracy is 69.4% on ImageNet (0.36% lower than ResNet-18).

The deformable operation is used in building block (c). Building block (c) is used to upsample the backbone features. The first 1×1 convolution is designed to map input channels to output channels. The following 3×3 depthwise deformable convolution samples the previous feature map, according to the offsets generated by 1×1 convolution. After that, a 2× upsampling layer, operated by the nearest neighbor kernel, is utilized to interpolate the higher resolution features. Note that, aside from the first layer, we only use 1×1 convolution and 3×3 depthwise (deformable) convolution in our build blocks. This way the building blocks of the whole network become more uniform and simple to support with specialized hardware.

Detection Heads We use the anchor-free CenterNet [289] method to directly predict a gaussian distribution for object keypoints over the 2D space for object detection. Given an image $I \in \mathbb{R}^{W \times H \times 3}$, our feature extractor generates the final feature map $F \in \mathbb{R}^{\frac{W}{R} \times \frac{H}{R} \times D}$, where R is the output stride and D is the feature dimension. We set $R = 4$ and $D = 64$ for all the experiments. As illustrated in Figure 12.6, the outputs include:

1. the keypoint heatmap $\hat{Y} \in [0, 1]^{\frac{W}{R} \times \frac{H}{R} \times C}$
2. the object size $\hat{S} \in \mathbb{R}^{\frac{W}{R} \times \frac{H}{R} \times 2}$
3. the local offset $\hat{O} \in \mathbb{R}^{\frac{W}{R} \times \frac{H}{R} \times 2}$

Here C is pre-defined as 20 and 80 for VOC and COCO, respectively. In order to reduce the computation, we follow the class-agnostic practice, using the single size and offset predictions for all categories. To construct bounding boxes from the keypoint prediction, we first collect the peaks in the keypoint heatmap $\hat{Y}$ for each category independently. Then we only keep the top 100 responses that are greater than its eight-connected neighborhood. Specifically, we use the keypoint values $\hat{Y}_{x_i y_i c}$ as the confidence measure of the i-th object for category c. The corresponding bounding box is decoded as $(\hat{x}_i + \delta\hat{x}_i - \hat{w}_i/2, \;\; \hat{y}_i + \delta\hat{y}_i - \hat{h}_i/2, \hat{x}_i + \delta\hat{x}_i + \hat{w}_i/2, \;\; \hat{y}_i + \delta\hat{y}_i + \hat{h}_i/2)$, where $(\delta\hat{x}_i, \delta\hat{y}_i) = \hat{O}_{\hat{x}_i\hat{y}_i}$ is the offset prediction and $(\hat{w}_i, \hat{h}_i) = \hat{S}_{\hat{x}_i\hat{y}_i}$ is the size prediction.

Quantization Quantization is a crucial step towards the efficient deployment of the GPU pre-trained model on FPGA accelerators. Although many previous works treat quantization as a separate process outside the algorithm-hardware co-design loop, we note that quantization performance greatly depends on the network architecture. As an example, the residual connection will enlarge the activation range of specific layers, which makes a uniform

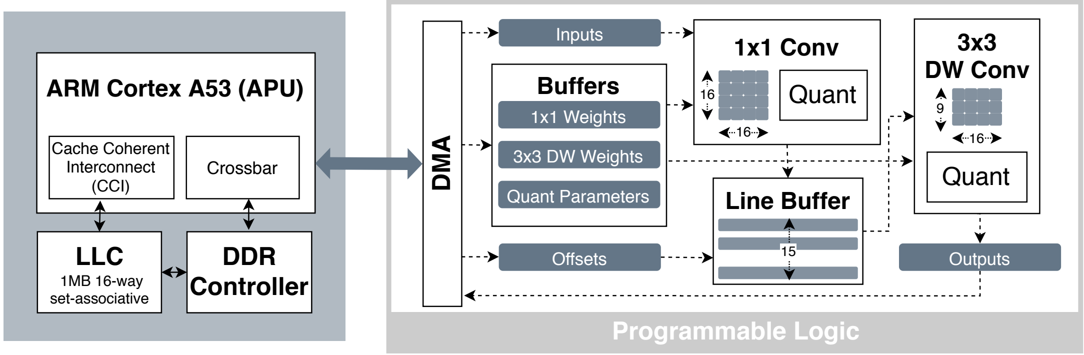


Figure 12.7: Architectural diagram of the FPGA accelerator.

quantization setting sub-optimal. And it requires a special design for addition in int32 format, otherwise, extra steps of quantization are needed to support the low-precision addition. With this prior knowledge, we use concatenation instead of residual connection throughout CoDeNet, and we do not use techniques such as layer aggregation [273], in order to achieve a simpler hardware design.

In order to achieve better AP, we perform 4-bit channel-wise quantization [124] for weights. Meanwhile, to ease the hardware design and accelerate the inference, we choose a symmetric uniform quantizer rather than a non-uniform quantizer, and we use 8-bit layer-wise quantization for activations. During quantization-aware fine-tuning, we use Straight-Through Estimator (STE) [14] to achieve the backpropagation of gradients through the discrete operation of quantization.

For the deformable convolution, quantization comprises two parts: 1) quantize the corresponding weights and activations, and 2) round and bound the sampling offsets of the deformable convolution. Compared to the standard convolution, the variable offsets will not significantly change the network's sensitivity or the allowable quantization bit-width. Regarding the original fractional offsets, we bound and round them to be integers within the range $[-8, 7]$. This modification eliminates the need for bilinear interpolation and results in a 1.9 AP drop on VOC as shown in Table 12.1.

Dataflow Accelerator

We develop a specialized accelerator to support the aforementioned CoDeNet design on an FPGA SoC. As shown in Figure 12.7, the FPGA SoC includes the programmable logic (PL), memory interfaces, a quad-core ARM Cortex-A53 application processor with 1MB LLC, and etc. Our accelerator on the PL side communicates to the processor through an AXI system bus. The High Performance (HP) and Accelerator Coherency Port (ACP) interfaces on the AXI bus allow the accelerator to directly access the DRAM or perform cache-coherent accesses to the LLC and DRAM. The processor provides software support to invoke the accelerator and to run functions that are not implemented on the accelerator.

With our co-design methodology, we are able to reduce the types of operations to support in the accelerator. Excluding the first layer for the full 3×3 convolution, CoDeNet only consists of the following operations: (i) 1×1 convolution, (ii) 3×3 depthwise (deformable) convolution, (iii) quantization, (iv) split, shuffle and concatenation. This helps us simplify the complexity of the control logic and thus saves more FPGA resources for the actual computation. We partition the CoDeNet workload so that the frequently-called compute-intensive operations are offloaded to the FPGA accelerator while the other operations are run by software on the processor. The operations we choose to accelerate are 1×1 convolution and 3×3 depthwise (deformable) convolution.

To leverage both the data-level and the task-level parallelism, we devise a spatial dataflow accelerator engine to execute a subgraph of the CoDeNet at a time and store the intermediate outputs to the DRAM. In the dataflow engine, the execution of computing units is determined by the arrival of the data and thus further reduces the overhead from the control logic. As illustrated in the architectural diagram in Figure 12.7, our accelerator executes 1×1 convolution with quantization and 3×3 depthwise (deformable) convolution with quantization in order. We implement the accelerator with Vivado HLS and its dataflow template. All functional engines are connected to each other through data FIFOs. Extra bypass signals can be asserted if the user would like to bypass either of the main computation blocks. By co-designing the network to use operations with fewer weight parameters, such as depthwise convolution, we are able to buffer the weights for all operations in the on-chip memory and enable the maximal reuse of the weights once they are on-chip. We also add a line buffer for the 3×3 depthwise (deformable) convolution to maximize the reuse of inputs on-chip. This optimization is enabled by the operation co-design discussed in Section 12.2. The line buffer stores 15 rows of the input image. The size of this buffer is larger than $15 \times w \times ic$ of any layers in the CoDeNet design. Our input tensors are laid out in the NHWC manner, allowing the data along channel dimension C to be stored in contiguous memory blocks.

1×1 **convolution** The compute engine for the 1×1 convolution is composed of 16×16 multiply-accumulate (MAC) units. At each round of the run, the engine takes 16 inputs along its channel dimension and broadcasts each of them to 16 MAC units. Meanwhile, it unicasts 16×16 weights for 16 input channels and 16 output channels to their corresponding MAC unit. There are 16 reduction trees of size 16 connected with the MAC units to generate 16 partial sums of the products. The partial sums are stored on the output registers and are accumulated across each round of the run. Every time the engine finishes the reduction along the input channel dimension, it feeds the values of the output registers to the output FIFO and resets their values to zero.

3×3 **depthwise (deformable) convolution** This engine directly reads 16 sampled 3×3 inputs from the line buffer design and multiplies them by 3×3 weights from 16 corresponding channels. Then it computes the outputs with 16 reduction trees to accumulate the partial sums along 3×3 spatial dimension. Both the original and the deformable depthwise convolutions can be run on this engine. The original depthwise operation is realized by hardcoding the offset displacement to be 1.

Quantization To convert the output from the 16-bit sum to 8-bit inputs, we add a quantization unit at the end of each compute engine. The quantization unit multiplies each output with a scale, and then adds a bias to it. It returns the lower 8 bits of the result as the quantized value. The parameters, such as the scale and bias for each channel, are preloaded to the on-chip buffer to save memory access time. Note that we also merge the batch normalization and ReLU in this compute unit. We follow the practice introduced in [107] to perform integer inference for our quantized model.

Our accelerator design can execute $16 \times 1 \times 250 \times 2 = 128$ GOPs for 1×1 convolution and $9 \times 16 \times 250 \times 2 = 72$ GOPs for 3×3 depthwise convolution simultaneously. On our target FPGA with 6GB/s DDR bandwidth, we can load 4 Giga pairs of 8-bit inputs and 4-bit weights per second. The arithmetic intensity required to reach the compute-bound is $128/4 = 32$ OPs/pair for 1×1 convolution and $72/4 = 18$ OPs/pair for 3×3 depthwise convolution. Our buffering strategy allows us to reach the compute-bound through the reuse of weights and activations.

Table 12.3: Quantized CoDeNet on VOC object detection.

Detector	Resolution	DownSample	Weights	Activations	Model Size	MACs	AP50
Tiny-YOLO	416×416	MaxPool	32-bit	32-bit	60.5 MB	3.49 G	57.1
CoDeNet1× (config a)	256×256	Stride4	32-bit	32-bit	6.06 MB	0.29 G	53.0
			4-bit	8-bit	0.76 MB	0.29 G	51.1
CoDeNet1× (config b)	256×256	Stride2+MaxPool	32-bit	32-bit	6.06 MB	0.29 G	57.5
			4-bit	8-bit	0.76 MB	0.29 G	55.1
CoDeNet1× (config c)	512×512	Stride4	32-bit	32-bit	6.06 MB	1.14 G	64.6
			4-bit	8-bit	0.76 MB	1.14 G	61.7
CoDeNet2× (config d)	512×512	Stride4	32-bit	32-bit	23.2 MB	3.54 G	69.6
			4-bit	8-bit	2.90 MB	3.54 G	67.1
CoDeNet2× (config e)	512×512	Stride2+MaxPool	32-bit	32-bit	23.2 MB	3.58 G	72.4
			4-bit	8-bit	2.90 MB	3.58 G	69.7

Table 12.4: Quantized CoDeNet on COCO object detection.

Detector	Weights	Model Size	MACs	AP	AP50	AP75	APs	APm	APl
CoDeNet1×	32-bit	6.07MB	1.24G	22.2	38.3	22.4	5.6	22.3	38.0
	4-bit	0.76MB	1.24G	18.8	33.9	18.7	4.6	19.2	32.2
CoDeNet2×	32-bit	23.4MB	4.41G	26.1	43.3	26.8	7.0	27.9	43.5
	4-bit	2.93MB	4.41G	21.0	36.7	21.0	5.8	22.5	35.7

12.3 Experiments

We implement CoDeNet in PyTorch, train it with a pretrained ShuffleNetV2 backbone, and quantize the network to use 8-bit activations and 4-bit weights. We devise several configurations of CoDeNet to facilitate the latency-accuracy tradeoffs for our final object detection solution on the embedded FPGAs. Different configurations of the CoDeNet are listed in Table 12.3 and 12.4 showing the accuracies for object detection on Pascal VOC and Microsoft COCO 2017 dataset.

Table 12.5: Performance comparison with prior works.

	Platform	Input Resolution	Framerate (fps)	Test Dataset	Precision	Accuracy
DNN1 [83]	Pynq-Z1	-	17.4	DJI-UAV	a8	IoU(68.8)
DNN3 [83]	Pynq-Z1	-	29.7		a16	IoU(59.3)
Skynet [280]	Ultra96	160 × 360	25.5		w11a9	IoU(71.6)
AP2D [131]	Ultra96	224 × 224	30.5	AD2P	w(1-24)a3	IoU(55)
Finn-R [17] [190]	Ultra96	-	16	VOC07	w1a3	AP50(50.1)
Tiny-Yolo-v2 [62]	Zynq-706 XC7Z045	224 × 224	43.1		w16a16	AP50(48.5)
Ours (config a)	Ultra96	256 × 256	32.2	VOC07	w4a8	AP50(51.1)
Ours (config b)		256 × 256	26.9			AP50(55.1)
Ours (config c)		512 × 512	9.3			AP50(61.7)
Ours (config d)		512 × 512	5.2			AP50(67.1)
Ours (config e)		512 × 512	4.6			AP50(69.7)

In Table 12.3, we show different configurations of CoDeNet with an accuracy-efficiency trade-off. *config c, d* and *e* use image size 512 × 512, which is the default resolution of CenterNet. Compared to Tiny-YOLO, our *config c* model is 10× smaller without quantization and 79.6× smaller with quantization, while achieving higher accuracy. In addition, the total MACs count of our compact design is 3.1× smaller than Tiny-YOLO. It can be seen that quantizing the model to 4–8 bits causes a minor accuracy drop, but can significantly reduce the model size (> 8×). In order to further save the MACs, we reduce the resolution to be 256 × 256, corresponding to *config a*, where we can still get 53 AP50 with about 1/4 total MACs compared with *config c*. Moreover, we found the downsampling strategy of the first layer play an important role, where a larger stride can benefit the speed (shown later in Table 13.2), but a smaller stride processes more information and can therefore improve accuracy (corresponding to *config b*). For scenarios that require more accurate detectors, we expand the channel size of *config c* (CoDeNet1×) by a factor of 2, which gives us config d that can achieve 69.6 AP50. After quantization, *config d* has a 67.1 AP50 with comparable MACs but 21× smaller memory size compared to Tiny-YOLO. By doubling the channel size (CoDeNet2×) and using a smaller stride, we have *config e*, which can achieve the highest 72.4 AP50 among all the configurations.

Table 12.4 shows the accuracy of CoDeNets on the Microsoft COCO 2017 dataset. Microsoft COCO is a more challenging dataset compared to Pascal VOC, where COCO has 80 categories but Pascal VOC has 20. Our results here are obtained with default 512 × 512 resolution, and with stride 2 convolution and max-pooling as the downsampling strategy. Besides AP50, COCO primarily uses AP as the evaluation metric, which is the average among AP[0.5:0.95] (namely AP50, AP55, ..., AP95). As we can see in the table, CoDeNet1× can achieve 22.2 AP with a model size of 6.07 MB. Applying quantization will cause a minor accuracy degradation, but can get an 8× smaller model. The same trend holds for CoDeNet2× where our model can get 26.1 and 21.0 AP, with and without quantization respectively.

We evaluate our accelerator customized for each CoDeNet configuration on the Ultra96 development board with Xilinx Zynq XCZU3EG UltraScale+ MPSoC device. Our accelerator design runs at 250 MHz after synthesis, and place and route. Table 12.6 shows the overall resource utilization of our implementation. We observe a 100% utilization of both

Table 12.6: FPGA resource utilization.

LUT	**FF**	**BRAM**	**DSP**
34144 (48.4%)	41827 (29.6%)	216 (100%)	360 (100%)

DSPs and BRAMs. Most DSPs are mapped to the 4-8 bit MAC units, and BRAMs are mainly used for the line buffer design. Our power measurements are obtained via a power monitor. We measured 4.3W on the Ultra96 power supply line with no workload running on the programming logic side and 5.6W power when running our network. On CoDeNet *config a*, our accelerator achieves 5.75 fps / W in terms of power efficiency.

We provide a Pareto curve in Figure 12.8 showing the latency-accuracy tradeoff for various CoDeNet design points with acceleration. Configuration a and b in this curve are trained and inferenced with images of size 256×256 instead of the original size 512×512. The smaller input image size leads to $\sim 4\times$ reduction in MACs. In configuration a, c and d, the stride of the first layer is increased from 2 to 4, which greatly reduces the first layer runtime on the processor. In configuration d and e, we use the CoDeNet $2\times$ model, where the channel size is doubled in the network, to boost the accuracy. The latency evaluation on our accelerator is done with a batch size equal to 1 without any runtime parallelization. We run the first layer of the network on the processor for all configurations.

A comparison of our solutions against previous works is shown in Table 13.2. We found that very few prior works on embedded FPGAs attempt to target the standard dataset like VOC or COCO for object detection, primarily due to the challenges of limited hardware resources and inefficient model design. Two state-of-the-art FPGA solutions that meet the real-time requirement in the DAC-UAV competition target the DJI-UAV dataset for drone image detection. However, object detection on DJI-UAV is a less generic and less challenging task than object detection on VOC or COCO. The images in the DJI-UAV dataset are taken from the top-down view. They typically contain very few overlapped objects. In addition, the DJI-UAV dataset is designed for single-object detection whereas VOC and COCO can be used for multi-object detection. Hence, in this work, we target VOC and COCO to provide a more general solution for multi-object detection and for images from a first-person view.

As shown in Figure 12.8 and Table 13.2, compared to the results from FINN-R [17] [190], the state-of-the-art embedded FPGA accelerator design targeting VOC, our configuration a and b (with single-batch inference latency of 31ms and 37ms respectively) achieve both higher accuracy, higher framerate, and lower latency. Another SOTA work Tiny-Yolo-v2 [62] attains low latency, but with lower accuracy, and it runs on a different FPGA platform.

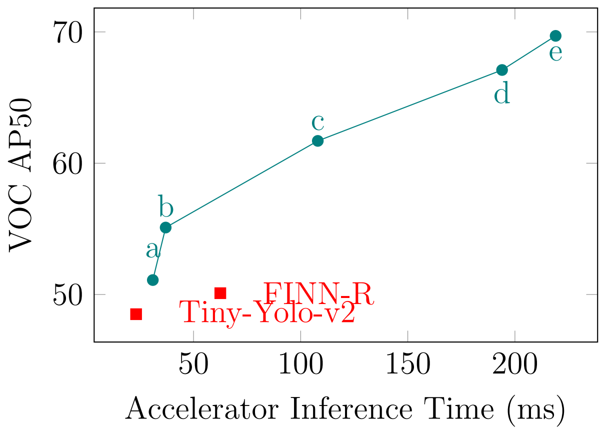


Figure 12.8: Latency-accuracy trade-off on VOC.

Chapter 13

HW-SW Co-Design: HAO

In CoDeNet we have shown that manual HW-SW co-design of the neural architectures, the quantization settings, and the hardware engine can have a significant improvement in performance. In this Chapter, we further discuss the following *key question*:

> How to effectively automate the HW-SW co-design pipeline so that heuristic knowledge and manual efforts are no longer required?

Here we introduce HAO, which is an automatic HW-SW co-design method based on configurable hardware subgraphs.

13.1 Method

In HAO, we expose a large design space in both hardware and algorithm configurations to accelerate DNNs. To efficiently navigate the search space, we first apply integer programming to prune the hardware configuration space by minimizing the latency subject to a set of hardware resource constraints. We then narrow the DNN architecture space by adopting Monte Carlo tree search (MCTS) [121] to minimize the quantization accuracy perturbation while satisfying a given latency constraint. In addition, we develop an accuracy predictor to estimate the accuracy of the DNN to further reduce the overall feedback time for each sample. Our flow produces a Pareto-optimal curve between latency and accuracy.

Hardware Design

We target FPGA in this work to demonstrate how co-designed hardware and DNN fully exploit the optimization opportunities in hardware with limited resources while achieving on-par accuracy. In this section, we model the resource consumption and the computation latency for different types of convolution kernels. On top of that, we formulate the overall resource constraints and latency objectives as an integer programming problem for

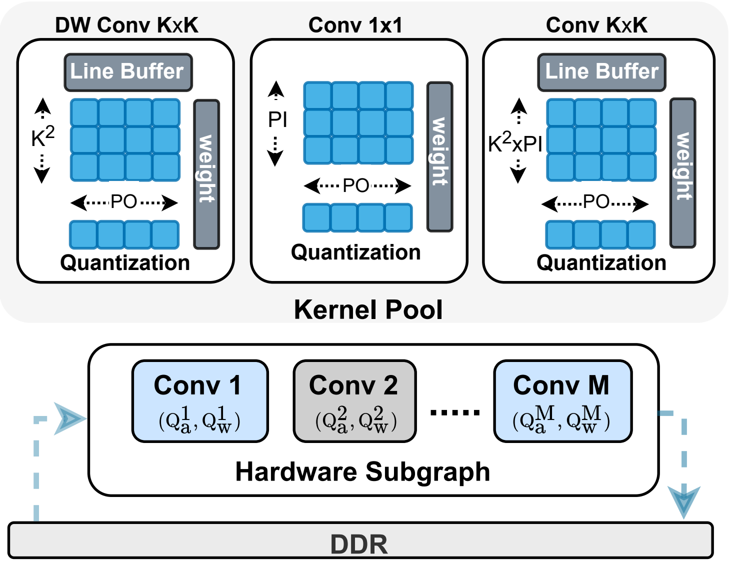


Figure 13.1: Hardware design space. The dataflow accelerator template consists of M convolution kernels that are selected from the kernel pool and spatially mapped to hardware. The tunable design parameters include the number of compute kernels M, the kernel type, filter size K, input and output channel parallelization factor PI and PO.

the subgraph-based design, which will serve as the latency simulator in the following DNN architecture optimization.

Hardware Subgraph Template

As shown in Figure 13.1, in HAO, we adopt a subgraph-based hardware design. A subgraph consists of several convolution kernels that are spatially mapped on hardware, which also corresponds to the major building block of neural architecture. For a given hardware subgraph, the possible building blocks for neural architecture include all the sub-layers of the subgraph since each kernel is implemented with a skip signal to bypass its compute in hardware. Each invocation to the accelerator computes one subgraph in the DNN architecture. The intra-subgraph results are buffered and streamed on FPGA and the inter-subgraph activations are communicated through DRAM.

We implement a parameterizable accelerator template in high-level synthesis (HLS). The generated dataflow accelerator can contain M convolution kernels chained through FIFOs to exploit pipeline-level parallelism. Each convolution kernel can be chosen from one of the three convolutions from the kernel pool: Conv $k \times k$, Depthwise Conv $k \times k$ [36], and Conv 1×1. The hardware implementation of each kernel typically comprises a weight buffer, a line buffer, a MAC engine, and a quantization unit to rescale outputs. All the computational units are implemented using integer-only arithmetics as in [107].

Table 13.1: Notations for hardware design.

Notation	Description	Notation	Description
H	feature map height	PI	parallelism on input channel
W	feature map width	PO	parallelism on output channel
Q	quantization setting	PF	array partition factor
Q_a	activation bitwidth	L_M	LUTs usage of a Multiplier
Q_w	weights bitwidth	L_A	LUTs usage of an Adder
Q_p	partial sum bitwidth	B_l	line buffer BRAM usage
k	kernel size	B_w	weights BRAM usage
Lat_{comp}	computation latency	N_w	number of weights buffered
$Lat_{\text{on/off}}$	latency of activation communication	N_{dsp}	total DSP usage of a kernel
		N_{bram}	total BRAM usage of a kernel
Lat_w	latency of loading weights	N_{luts}	total LUTs usage of a kernel
S	hardware subgraph	N_{wbuf}	BRAM usage for weights buffer
A	neural architecture	N_{sbuf}	BRAM usage for scale buffer
M	number of kernels in S	N	number of layers in A

Hardware Resource Modeling

This section describes the modeling details of different FPGA resources. We adopt a bottom-up design flow to model the utilization of LUTs and DSPs for low-bit multiply-accumulate (MAC) operations on FPGA. In addition, our model derives the BRAM utilization based on data size and precisions as well as the parallelization factors of the compute kernels. Table 13.1 lists the notations used in this paper.

LUTs Both DSPs and LUTs can be used for computation on FPGA. It is more efficient to perform ultra low-bit computation on LUTs compared with DSPs. We use pragma to direct the mapping of low-precision MAC operations to LUTs in HLS. To build a precise model, we perform full logic synthesis to obtain the LUTs consumption on low bitwidth multipliers and adders. Figure 13.2 shows the LUTs consumption on different activation and weight bitwidths ranging from 2 to 8. We denote the LUT resource lookup function of multipliers as $L_M(Q_w, Q_a)$ where Q_w and Q_a represent the bitwidth of weights and input activations respectively. Derived from the logic synthesis results, the LUT consumption of the adders $L_A(Q_p)$ for carrying out Q_p bit partial sum accumulation can be expressed as $L_A(Q_p) = Q_p + 7$.

DSP The embedded DSP slice on FPGA supports the MAC operation in the following format:

$$P \mathrel{+}= A \times (B + C) \tag{13.1}$$

In naive HLS mapping, one DSP slice is configured to support one MAC. To improve DSP throughput for low-bit operations, we use the shift-and-pack method in [65] to efficiently map

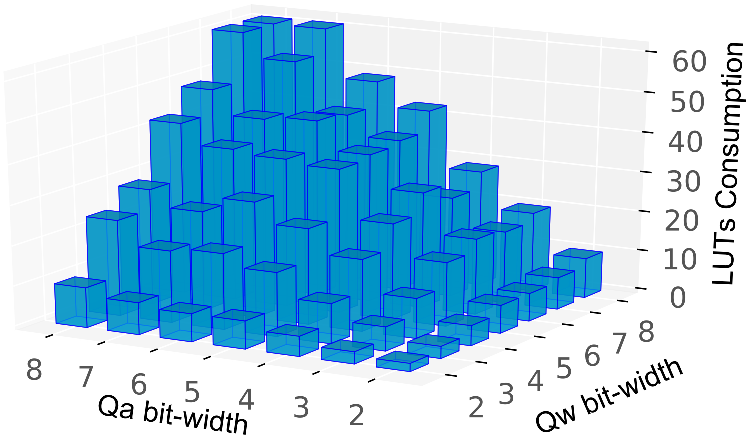


Figure 13.2: LUT usage of multipliers with different input precisions.

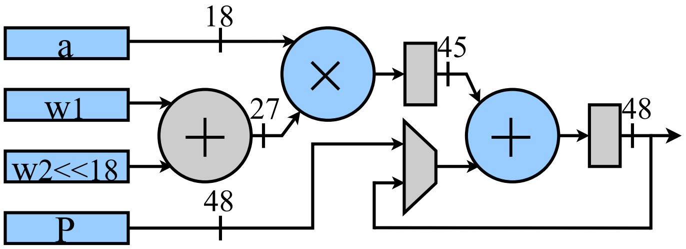


Figure 13.3: Example mapping of two low-precision MACs $a \times w_1$ and $a \times w_2$ onto a DSP with 27×18 multiplier support. The multiplexer in DSP can choose between self-accumulating or chaining mode.

two MACs on one DSP by leveraging the additional pre-adder. Given the input activation a and the weights w_1 and w_2 for two different output channels, as shown in Fig. 13.3, the packing algorithm first sign-extends w_1 to 27 bits and left shifts w_2 by 18 bits. The output P can be further accumulated with the partial sum or separated into two products P_1 and P_2. This shift-and-pack method can be applied to the situation when w_1 and w_2 are no larger than 8 bits.

BRAM We assume a buffering scheme in which we fully exploit reuse opportunities. The 18-Kb BRAMs usage B_w for the weight buffer can be calculated as:

$$B_w = \lceil N_w \times Q_w / PF / 18\text{Kb} \rceil \times PF \tag{13.2}$$

where N_w is the maximum number of weights to store on-chip, Q_w is the bitwidth of weights, and PF is the BRAM partition factor of the weights buffer. For convolution kernel with size $k > 1$, we implement a line buffer to maximize input reuse. The number of BRAMs B_l needed for line buffer is:

$$B_l = \lceil (W \times C)_{\max} \times Q_a / 18\text{Kb} \rceil \times k \tag{13.3}$$

where $(W \times C)_{max}$ is the maximum product between the size of image width W and channel C over the entire network. Our line buffer implementation merges the input width and channel dimension of the feature map into one dimension, and k rows of line buffers are allocated for the $k \times k$ convolution kernel.

Hardware Resource Allocation

With the resource modeling, we can further estimate the optimal resource allocation for a hardware subgraph under the resource constraints of the target FPGA. For full $k \times k$ *Conv*, given the input channel parallelization factor PI and output channel parallelization factor PO, the compute engine loads $k^2 \times PI$ inputs in parallel and computes PO output partial sums. The total BRAM usage N_{wbuf} for on-chip buffers is:

$$N_{\text{wbuf}} = \begin{cases} B_w + B_l & k > 1 \\ B_w & k = 1 \end{cases} \tag{13.4}$$

The engine is composed of $k^2 \times PI \times PO$ MAC units that can be mapped to either DSPs or LUTs, incurring usage in LUTs N_{luts} or DSPs N_{dsp}:

$$\begin{aligned} N_{\text{dsp}} &= k^2 \times PI \times PO/2 \\ N_{\text{luts}} &= k^2 \times PI \times PO \times (L_M(Q_w, Q_a) + L_A(Q_p)) \end{aligned} \tag{13.5}$$

For $k \times k$ *Depthwise Conv* where each output channel result is corresponding to the inputs from the same channel, we use only PO to denote the channel dimension parallel factor. The $k \times k$ computation engine takes $k^2 \times PO$ input and computes PO partial sums concurrently. Similarly, the BRAM usage for the compute kernel is:

$$N_{\text{wbuf}} = B_w + B_l \tag{13.6}$$

The LUT or DSP usage to support depthwise convolution grows linearly with the PO parallelism factor:

$$\begin{aligned} N_{\text{dsp}} &= k^2 \times PO \\ N_{\text{luts}} &= k^2 \times PO \times (L_M(Q_w, Q_a) + L_A(Q_p)) \end{aligned} \tag{13.7}$$

Regarding the *Quantization* unit that converts partial sum in high-precision to quantized input for the next layer, we implement it with DSP with a parallelization factor of PO. Its overall resource usage is:

$$N_{\text{dsp}} = PO, N_{\text{sbuf}} = B_s \tag{13.8}$$

Since we perform channel-wise quantization on weights, each output channel has its own quantization scale. We thus set the number of buffered scales N_s to OC. The calculation of B_s is similar to B_w in equation 13.2. The bitwidth of scale Q_s ranges from 16-24 depending on the actual value range after obtaining the integer scale using the inference scheme in

[107]. The total BRAM usage N_{bram} is a sum of weight buffer usage N_{wbuf} and scale buffer usage N_{sbuf}:

$$N_{\text{bram}} = N_{\text{wbuf}} + N_{\text{sbuf}} \tag{13.9}$$

Hardware Latency Objective

Given a layer with input channel size IC, output channel size OC, input height H and width W, the compute latency is:

$$Lat_{\text{comp}} = \begin{cases} H \times W \times \lceil IC/PI \rceil \times \lceil OC/PO \rceil & \text{if } \textit{full} \\ H \times W \times \lceil IC/PO \rceil & \text{if } \textit{depthwise} \end{cases} \tag{13.10}$$

depending on if the kernel type is full or depthwise convolution. The communication latency for loading the activation on-chip and off-chip can be roughly calculated as:

$$\begin{aligned} Lat_{\text{on}} &= H \times W \times IC \times Q_a/bw \\ Lat_{\text{off}} &= H \times W \times OC \times Q_a/bw \end{aligned} \tag{13.11}$$

where bw is the practical bandwidth of off-chip memory. Similarly, the latency of loading weights can be estimated as:

$$Lat_w = \begin{cases} k^2 \times IC \times OC \times Q_w/bw & \text{if } \textit{full} \\ k^2 \times IC \times Q_w/bw & \text{if } \textit{depthwise} \end{cases} \tag{13.12}$$

Based on the latency model for a single layer, we can further derive the latency of computing a subgraph. A hardware subgraph design with M convolution kernels can be represented as $S = \{K_1, K_2, ...K_M\}$ with specific quantization bitwidths $Q = \{(Q_a^1, Q_w^1), ..., (Q_a^M, Q_w^M)\}$. For a given network architecture $A = \{a_1, a_2, ...a_N\}$, the subgraph mapping $\{g_1, ..., g_L\}$ can be generated using a grouping function f_m:

$$\{g_1, g_2, ..., g_L\} = f_m(\{a_1, a_2, ..., a_N\}) \tag{13.13}$$

To model the overlapping of the dataflow architecture, the latency of computing each g_i can be approximated using the maximum latency over all the subgraph layers. Besides, to execute each layer on hardware, the accelerator will preload the weights to the on-chip buffer before the kernel starts, and apply double-buffering to hide the communication overhead of the input activations. The overall latency for computing a subgraph can be written as:

$$\begin{aligned} Lat(g_i) = &\max(Lat_{\text{on}}^{i1}, Lat(a_{i1}), ..., Lat(a_{iM}), Lat_{\text{off}}^{iM}) \\ &+ \sum\nolimits_{j=1}^{M} Lat_w^{ij} \end{aligned} \tag{13.14}$$

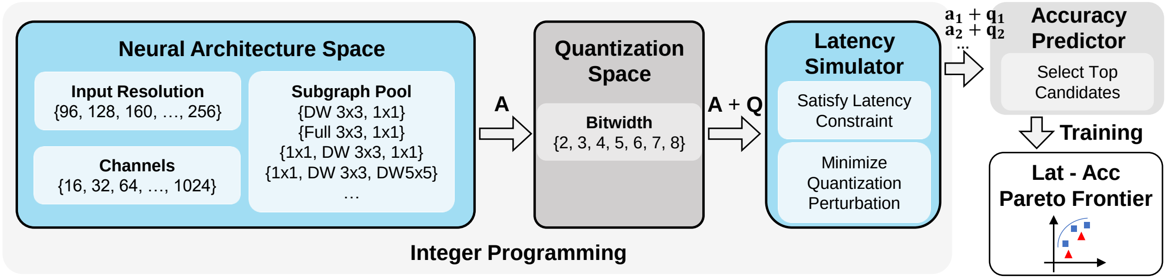


Figure 13.4: Illustration of HAO pipeline.

With the hardware analytical model above, we can then formulate the automatic hardware design problem as an integer programming that minimizes the overall latency:

$$\begin{aligned} \min \quad & \sum_{i=1}^{L} Lat(g_i) \\ \text{s.t.} \quad & \sum_{k \in S} N_{\text{dsp}}^k \leq T_{\text{dsp}} \\ & \sum_{k \in S} N_{\text{luts}}^k \leq T_{\text{luts}} \times \beta \\ & \sum_{k \in S} N_{\text{bram}}^k \leq T_{\text{bram}} \end{aligned} \tag{13.15}$$

where T_{dsp}, T_{luts}, T_{bram} are the total resources available on the target FPGA device. Note that β is an empirical parameter describing the percentage of total LUTs allocated for MAC computation, which is set to 50% in our experiments. We treat this formulation as a sub-program to the DNN design optimization which will be covered in the next section. Given the explicitly expressed constraints and objective, we are able to directly generate the corresponding hardware implementation that minimizes the latency for different DNN design choices with different quantization schemes and kernel types.

DNN Design

Co-search for hardware-friendly neural network architectures and mixed quantization precisions is computationally intensive and time-consuming. In HAO, we formulate the search to an integer programming problem. In Section 13.1 we present our search space of neural architectures. Given a latency constraint, we can first search feasible neural architectures and corresponding mixed-precision bitwidth settings by applying the aforementioned hardware latency model as well as a model quantifying the effect of quantization perturbation. We then use an accuracy predictor to compare across different networks and find the Pareto-optimal architectures and quantization settings among all candidates.

Search Space of Neural Architectures

In HAO, we construct the neural network architectures from subgraphs with feasible hardware mappings on FPGAs. Our subgraphs are combinations of operations such as convolution or depthwise convolution with a kernel size of 1×1 or $k \times k$ as mentioned in the previous section. Although only one subgraph can be chosen on hardware, the possible building blocks for neural architecture search include the sub-layers of the subgraph. This is because each layer in the subgraph can be decided whether to bypass or not using a skip signal in hardware. We set no limit on the total number of subgraphs and choose the channel size for different layers from $\{16, 32, 64, 128, 256, 512, 1024\}$. We also consider input resolution in HAO with potential configuration from $\{96, 128, 160, 192, 224, 256\}$. Consequently, our search space is significantly larger compared to the prior work [19, 150, 250, 258, 225]. For example, in [225], the same cell configuration is repeated within every block. A standard search setting is to use 5 blocks with 3 identical cells in each block, and each cell, typically with 3 layers, has a sub-search space of 432, resulting in a search space of size $432^5 \approx 10^{13}$. In comparison, even with a simple subgraph $\{$*1x1 convolution*, *3x3 depthwise convolution*$\}$, assume the number of layers is 45 (same as [225]), the size of search space in HAO is $(2 \times 7)^{45} \approx 10^{51}$. The large search space of HAO makes it more likely to encompass designs with good efficiency and high accuracy for broader deployment scenarios with various hardware and latency constraints.

Integer Programming

Given a latency constraint Lat_0, we use integer programming to obtain feasible neural architectures and corresponding quantization settings. Specifically, based on the aforementioned hardware simulator, inference latency (Lat) is a function (denoted as $\mathbb{L}$) of neural architecture (A) and the quantization setting (Q) for the subgraph. In equation 13.16, i and j are layer indexes, N represents the total number of layers, and M represents the number of layers in a subgraph.

$$\begin{gathered} Lat = \mathbb{L}(A, Q), \\ A = \{k_i, H_i, W_i, IC_i, OC_i, S_i, i \in [1, N]\}, \\ Q = \{Q_a^j, Q_w^j, j \in [1, M]\} \end{gathered} \tag{13.16}$$

In HAO, perturbation, denoted as $Pert$, is used to estimate the accuracy degradation caused by quantization. For a given neural architecture, the accuracy of the full-precision pretrained model is irrelevant to the quantization setting Q. The perturbation models the relative accuracy change to the pretrained network among different Q. As shown in equation 13.17, the perturbation should be multiplied with a constant λ to have the same scale as accuracy, but this will not change relative accuracy ranking since $PretrainedAcc$ in equation 13.17 is a constant. As in [52], the total perturbation $Pert$ can be estimated by summing the perturbation contributed from each layer $Pert_i$. Using the norm of ΔW_i (the distance between the quantized tensor and the original tensor W_i) and the trace of Hessian matrix H_i, the $Pert_i$ can be calculated as follows (i is the layer index).

$$\begin{aligned} Acc &= PretrainedAcc - \lambda Pert, \\ Pert &= \mathbb{P}(A, Q) = \sum_{i=1}^{N} Pert_i, \\ Pert_i &= \overline{Tr}(H_i) \cdot \|\Delta W_i\|_2^2, \end{aligned} \tag{13.17}$$

With a latency constraint Lat_0, we need to find feasible neural architecture A and then determine the corresponding quantization setting Q to minimize perturbation. Note that A contains integer architectural parameters (kernel size, feature resolution, channel number, stride, etc), and Q contains the bitwidths of layers in the subgraph, which are integer values chosen from $\{2, 3, 4, 5, 6, 7, 8\}$. Therefore, the task to find A and Q satisfying latency constraint Lat_0 can be formulated as an integer programming problem as shown in equation 13.18.

$$\begin{aligned} &\min_{Q} \mathbb{P}(A, Q), \\ s.t.\ \ &\mathbb{L}(A, Q) \leq Lat_0 \end{aligned} \tag{13.18}$$

The latency constraint in equation 13.18 can be modified to equation 13.19 to reduce the number of neural architecture candidates. This modification is based on the assumption that neural architectures with higher latency tend to have more complex structures and higher expression capability, and therefore higher accuracy. α here is a hyperparameter ranging from 0 to 1. A larger α can lead to a lower search cost.

$$\alpha Lat_0 \leq \mathbb{L}(A, Q) \leq Lat_0 \tag{13.19}$$

We apply Monte Carlo tree search (MCTS) [121] for better sample efficiency on finding feasible neural architectures and quantization bitwidths that satisfy equation 13.18 and equation 13.19. Benefiting from its online model, MCTS can dynamically trade-off exploration and exploitation, which makes MCTS hard to be trapped in local optimum compared to other methods such as Bayesian optimization or greedy algorithms. With the heuristic that $\mathbb{L}(A, 2bit) \leq \mathbb{L}(A, Q) \leq \mathbb{L}(A, 8bit)$, we first find A that satisfies equation 13.20 and then solve for appropriate quantization setting Q. We follow the standard to set A (then Q in the next step) as state, and our actions are selected from {increase/decrease channel, increase/decrease resolution, skip/unskip a layer, add/delete a subgraph, termination}. More details about MCTS can be found in [121, 5, 238].

$$\begin{aligned} \alpha Lat_0 &\leq \mathbb{L}(A, 8bit) \\ \mathbb{L}(A, 2bit) &\leq Lat_0 \end{aligned} \tag{13.20}$$

Accuracy Predictor

As discussed in Section 13.1, given a latency constraint Lat_0, neural architecture candidates and corresponding quantization settings can be obtained with different perturbations. To compare different neural architectures, a predictor is used to estimate the accuracy of pre-trained models with given architectures. In HAO, we directly stack architectural parameters of each layer together as the input vector, and then we apply a support vector regression (SVR) model to predict the accuracy. It should be noted that we choose the SVR predictor for simplicity and better sample efficiency, since SVR models generally require fewer data to train compared to neural networks used in [245, 228]. To quickly train the predictor, we collect {architecture, accuracy} data by training 10 large neural networks from scratch and then reusing the weights while fine-tuning them to 200 different architectures. In our experiments, all neural networks are built by linearly stacking subgraphs, meaning that they are generally similar to each other. To support more complicated architectures such as DenseNet [98] or LSTMs [220], as suggested in [245, 228], using a better strategy (such as autoencoder) for neural architecture representation, using semi-supervised learning with unlabelled data, and using graph convolutional networks (GCN) as the predictor can further improve performance, with the cost of more computation resources and time.

We use the accuracy predictor to sort candidates that satisfy the latency constraint Lat_0. Since the accuracy predictor can be shared with different subgraphs, we repeat the aforementioned process for all subgraphs and select the top neural architectures and corresponding quantization settings[1]. We finally train them from scratch on ImageNet and then quantize the models as the final results of HAO.

13.2 Experiments

Simulator Performance

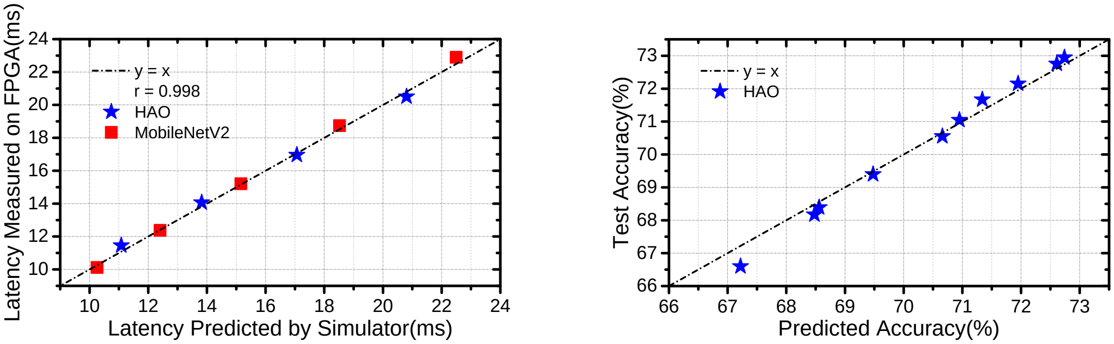


Figure 13.5: (Top) The correlation between latency predicted by the hardware simulator (after calibration) and the latency directly measured on FPGA. (Bottom) The correlation between predicted accuracy and the accuracy tested on the ImageNet validation set.

[1] In our experiments we train top 5 architectures with corresponding quantization settings and choose the best one for a given latency constraint.

In Section 13.1, we present an analytical latency simulator that can quickly estimate the inference latency given a DNN architecture. The optimization algorithm in Section 13.1 uses the simulator to obtain quick latency feedback. To test the effectiveness of our latency simulator, we synthesize several accelerators for different MobileNetV2 and HAO designs. The hardware parameters of different implementations are automatically generated by hardware optimization in equation 13.15. To calibrate our latency model for the target FPGA, we first perform linear regression to fit the cycle prediction to the hardware execution latency. We obtain a calibrated latency model $1.27 \times Lat + 3.8$ and use it for our latency prediction. Then for different accelerator implementations, we obtain the latency pairs from our simulator and the real hardware execution, and plot them in Figure 13.5. We observe a strong linear relationship ($r = 0.998$) between the real inference latency and the estimated latency.

In addition to the hardware latency simulator, HAO also uses an accuracy predictor to reduce computational costs. We show the performance of the predictor in Figure 13.5. As can be seen, for different CNN models in our search space, the results of our accuracy predictor align well with the actual test accuracies on the ImageNet validation dataset.

Experimental Results

In this section, we present the accuracy and latency results of HAO on the Ultra 96 board with a Xilinx Zynq ZU3EG FPGA. We show that HAO outperforms manually designed solutions, as well as solutions with automatically searched DNN architectures and quantization settings.

Figure 13.6 shows the Pareto frontier of HAO with respect to accuracy and latency. MobileNetV2 [205] is a popular neural architecture manually designed for efficient inference. The original MobileNetV2 is in floating-point format. To achieve a fair comparison, we quantize MobileNetV2 to 8-bit weights and 8-bit activations, and then run it on FPGA with a {1x1 convolution, 3x3 depthwise convolution, 1x1 convolution} subgraph. We follow [205] to change the channel width multiplier (selected from $\{1.0, 0.75, 0.5, 0.3\}$) and input resolution (selected from $\{224, 192, 160, 128, 96\}$) of MobileNetV2, in order to trade-off latency and accuracy. In comparison, the neural architecture (including input resolution) and quantization bitwidth setting are automatically selected in HAO. As can be seen, HAO outperforms MobileNetV2 on a wide range of latency values. HAO can achieve 72.5% top-1 accuracy with 20ms latency (50 fps), which is more than 1% higher accuracy than MobileNetV2 while running 15% faster. In the cases with a more strict latency constraint (for example autonomous vehicles), HAO can still preserve 66% accuracy with only 8ms latency (125 fps). This is significantly higher than the 63% of MobileNetV2 while being faster. Furthermore, we compare with results from MnasNet [225], which is a hardware-aware neural architecture search method. As in Figure 13.6, HAO also outperforms MnasNet by a large margin[2].

In addition to comparing Pareto-frontier performance with our own hardware implementation, we also compare HAO with various previous works in Table 13.2. [191, 115, 74, 217]

[2] Part of the MnasNet Pareto curve is out of the latency range in Figure 13.6. We present these extra results in Table 13.2.

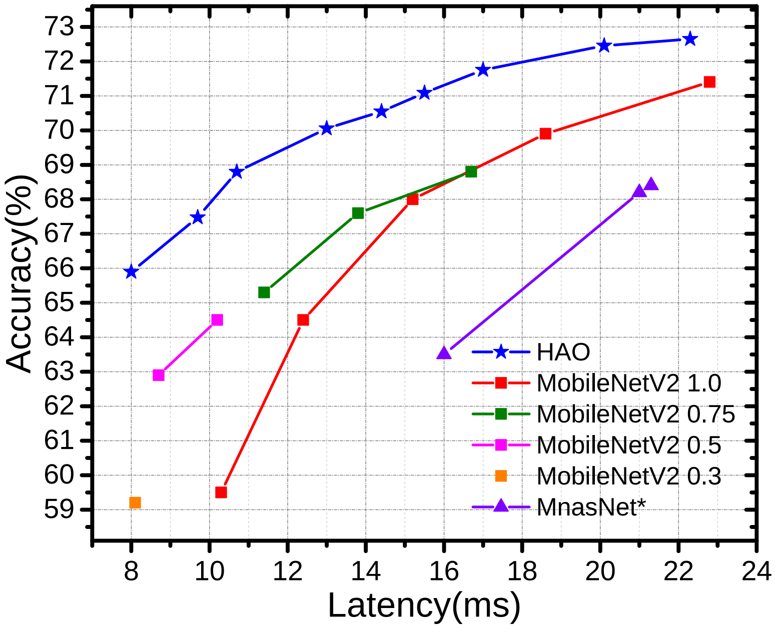


Figure 13.6: Pareto frontier for accuracy and latency. We generate the Pareto frontier of MobileNetV2 and MnasNet by varying width multipliers as well as the input resolution, as suggested in the references [205, 225]. As can be seen, HAO results outperform MobileNetV2 and MnasNet by a large margin on Zynq ZU3EG.

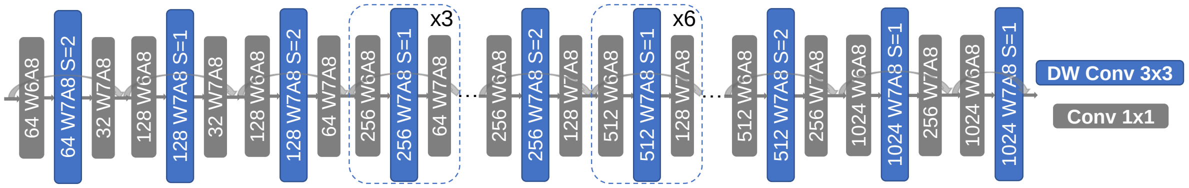


Figure 13.7: Illustration of neural architecture and quantization setting searched by HAO. W and A stand for weight and activation quantization bitwidth, and S is the stride of a specific convolutional layer. DW Conv stands for depth-wise convolution.

are manually designed solutions. [139, 114] are search-based methods. Note that these prior works target larger FPGA boards with more resources, and some use more complex neural architectures, 16-bit fixed-point or floating-point precision. For a fair comparison, we further compare HAO with [264, 17, 225, 250, 205], which have the same hardware platform (Zynq ZU3EG) as ours[3]. For HAO, we apply layer-wise quantization for activations and channel-wise quantization for weights, with standard linear quantizer and static quantization for the simplicity of deployment. As can be seen in Table 13.2, HAO achieves state-of-the-art per-

[3] Note that [225, 250] are well-known hardware-aware search algorithms, and we implement their searched results on Zynq ZU3EG for comparison.

Table 13.2: Performance comparison on ImageNet with prior works.

	Platform	**Input Resolution**	**Framerate(fps)**	**Quantization Bitwidth**	**Top-1 Accuracy(%)**
EDD-Net-2 [139]	Zynq ZU9EG	224 × 224	125.6	W16A16	74.6
HotNas-Mnasnet [114]	Zynq ZU9EG	224 × 224	200.4	NA	73.24
HotNas-ProxylessNAS [114]	Zynq ZU9EG	224 × 224	205.7	NA	73.39
EDD-Net-3 [139]	Zynq XC7Z045	224 × 224	40.2	W16A16	74.4
VGG16 [279]	Zynq XC7Z045	224 × 224	27.7	W16A16	69.3
VGG-SVD [191]	Zynq XC7Z045	224 × 224	4.5	W16A16	64.64
VGG16 [217]	Stratix-V	224 × 224	3.8	W8A16	66.58
VGG16 [74]	Zynq 7Z020	224 × 224	5.7	W8A8	67.72
Dorefa [115]	Zynq 7Z020	224 × 224	106.0	W2A2	46.10
Synetgy [264]	Zynq ZU3EG	224 × 224	66.3	W4A4	68.30
FINN-R [17]	Zynq ZU3EG	224 × 224	200.0	W1A2	50.30
MobileNetV2 [205]	Zynq ZU3EG	224 × 224	43.5	W8A8	71.40
MnasNet-A1 [225]	Zynq ZU3EG	224 × 224	22.3	W8A8	74.60
MnasNet-A1 [225]	Zynq ZU3EG	192 × 192	27.8	W8A8	73.33
MnasNet-A1-0.75 [225]	Zynq ZU3EG	224 × 224	31.0	W8A8	72.70
MnasNet-A1 [225]	Zynq ZU3EG	160 × 160	35.8	W8A8	71.35
FBNet-B [250]	Zynq ZU3EG	224 × 224	24.6	W8A8	73.20
FBNet-iPhoneX [250]	Zynq ZU3EG	224 × 224	21.3	W8A8	72.62
HAO	Zynq ZU3EG	256 × 256	44.9	W-mixed A8	72.68
HAO	Zynq ZU3EG	256 × 256	50.0	W-mixed A8	72.45
HAO	Zynq ZU3EG	224 × 224	58.9	W6A8	71.76
HAO	Zynq ZU3EG	224 × 224	77.0	W-mixed A8	70.06
HAO	Zynq ZU3EG	192 × 192	93.5	W-mixed A8	68.80

formance on embedded FPGA with limited resources. With higher top-1 accuracy (68.8% vs 68.3%), HAO solution is significantly faster than Synetgy [264] (94fps vs 66fps), albeit Synetgy is assisted by extra operations such as shift. Moreover, when the framerate is 50fps, HAO can achieve 72.5% top-1 accuracy on ImageNet, which is more than 1% higher than MnasNet-A1 (71.4%) while being 14% faster. Compared with FBNet-iPhoneX, HAO obtains slightly better accuracy (72.7% vs 72.6%), while having a much higher framerate (45 vs 21). It should be noted that for different hardware platforms or different latency constraints, previous methods need to repeat the whole search pipeline to find appropriate solutions, while the predictor in HAO can be shared so that no additional search cost will be required.

Table 13.3: Hardware resources utilization and power

LUTs	FF	DSP	BRAM	Power
61362(87.0%)	55136(39.0%)	360(100%)	431(99.8%)	5.5W

Table 13.3 shows the hardware resource utilization and power usage for HAO on Zynq ZU3EG FPGA. We observe 4.3W power consumption with no workload running on the programming logic side and 5.5W power when running the network. Besides, we are able to utilize 100% of DSP and 87% of LUTs on the FPGA, showing the effectiveness of our hardware resource modeling. In the optimization program in equation 13.15, we allocate β percent of LUTs as a computation resource to search for optimal design parameters, which makes the LUTs utilization more controllable. In this way, the simulator can automatically decide whether to implement a kernel on DSP or LUTs based on the quantization setting Q. As a result, we can achieve high resource utilization by leveraging the benefits of mix-precision operations on FPGA.

In Figure 13.7, we show one of the searched results by HAO. A subgraph {1x1 convolution, 3x3 depthwise convolution, 1x1 convolution} is used in this solution. As can be seen, HAO finds that a 6-bit/7-bit mixed-precision quantization setting is better than 8-bit uniform quantization for weights. In general, lower bit-width means more computation units under the same resource constraints, but it can lead to larger quantization perturbation. HAO can balance the efficiency and perturbation, and we observe that the 8-bit counterpart of HAO 6/7-bit result runs 5% slower with negligible accuracy gain. Moreover, the results of HAO show that, for our implementation on Zynq ZU3EG, solutions with solely 3×3 depthwise convolution perform better than those with a mixture of 3×3 and 5×5 depthwise convolution. This is due to the fact that when using a mixture of 3×3 and 5×5 depthwise convolution, either 3×3 or 5×5 kernel will be idle when invoking the accelerator, which is a waste on platforms with limited hardware resources.

Chapter 14

HW-SW Co-Design: ETA

Although HAO can automatically conduct HW-SW co-design, the number of operations within a subgraph is still limited, and the cost of training has room to improve. In this Chapter, we want to solve the following *key problem*:

> The time and computational cost of most co-design or neural architecture search algorithms increase drastically as the number of candidate operations increases.

To tackle this problem and involve more advanced candidate operations, we develop ETA, which enables fast pretraining by effectively and efficiently leveraging the teacher-based block-wise knowledge distillation.

14.1 Method

In this paper, we aim to transform a pretrained and accurate teacher model into an efficient yet accurate student model. We achieve this goal via three phases: 1) we conduct layer-wise knowledge distillation to obtain initial weights for different candidate operations, 2) we perform non-linear integer optimization to solve for efficient networks, based on an accuracy predictor, and 3) we finetune the selected operators across all layers in a joint manner.

Phase 1: Pretraining Operators

Given hardware budget constraints, in order to find an appropriate neural architecture and the corresponding quantization scheme, we first generate a large pool of operations containing common building operations of neural networks as alternative candidates for operator replacement. We apply layer-wise knowledge distillation to pretrain the weights for each candidate operator, easing the hardness of model selection in the next phase. We filter a set of operators in line with [174] covering operations from both widely used convolutional networks and vision transformers.

Elastic Width

Since we are conducting layer-wise knowledge distillation to get pretrained weights for every candidate operator in the pool, the time consumption of step 1 is highly correlated with the number of candidates. Considering the fact that one basic operator type (for example, the residual block) can have many variants with different expansion ratios, we apply elastic width to jointly train a basic operator type instead of training all its variants separately. Specifically, we train the super operator that has the largest number of channels per operator type. During pre-training, to pretrain all variants at once, we rank the internal channels based on their magnitude and randomly select a subset to do forward-backward (that means the top-ranking channels are always selected, and the low-ranking channels are only trained when a large subset is chosen). This offers the freedom to select sub-variants from the superblock during the searching phase next. Experimental results in Section 14.3 show that using elastic width can significantly cut down on the associated pre-training time cost of phase 1, while offering similar efficacy.

Layer-wise Knowledge Distillation

We apply a layer-wise knowledge distillation [174] to simultaneously obtain the weights for all candidate operators. Given a teacher architecture T and a student architecture S, we define the loss approximating the feature map of the student to the teacher on an input tensor x:

$$\mathcal{L}(T, S, x) = \sum\nolimits_{i=1}^{N} \|s_i(x) - t_i(x)\|_2^2. \tag{14.1}$$

Note that N is the number of layers in teacher T, t_i and s_i represent the i^{th} layer in the teacher and student network, respectively. In our experiments, we use a single epoch for layer-wise knowledge distillation to pretrain the weights for candidate operators.

Phase 2: Model Selection

After obtaining weights for candidate operators in phase 1, we learn a simple accuracy predictor that can predict the final performance given neural architecture as input. Given an estimate of the operator-accuracy relationship, we then apply quantization-aware selection on top of the accuracy predictor to jointly consider quantization degradation if any[1]. Finally, we apply non-linear integer optimization to select the final operators of the network and the corresponding quantization bitwidth. The optimization aims to maximize the expected accuracy of the accuracy predictor. We discuss details next.

[1] Note that the quantization-aware selection is for mixed-precision quantization, which can be skipped if mixed-precision is not supported or if applying uniform 8-bit quantization is already sufficient.

Accuracy Predictor

Conventional accuracy predictors that gauge model-accuracy relationships requires extensive sets of runs with full-recipe training to get final model performances. However, such cost can be dramatically alleviated given already pretrained operators that already hint at their strengths. As opposed to training entire networks end-to-end, we quickly evaluate the performance of the pretrained, not yet finetuned, candidate networks with a validation set constructed by 5% of training data. And then given different student architecture S, we obtain the accuracy Acc on the validation set and collect the $\{S, Acc\}$ pairs to form a dataset to train our accuracy predictor $\mathcal{P}$. The student architectures are randomly selected from the search space satisfying single or multiple latency constraints (also discussed in Section 14.3). Exploring the encoding scheme of neural network architectures has been an active research topic [245, 41, 184, 135, 25]. In this work, we conduct experiments with two different encoding schemes: 1) we use the index of operators in each layer to form a vector as the encoding of the whole network[2], and 2) we concatenate the one-hot vector of each layer to form the vector representation of the whole network. Denote encoding scheme as $\mathcal{E}$ and the number of $\{S, Acc\}$ pairs in a batch as M, we train the accuracy predictor $\mathcal{P}$ using the following mean square error (MSE) loss:

$$\mathcal{L}(S, Acc) = \frac{1}{M} \sum\nolimits_{i=1}^{M} (Acc - \mathcal{P}(\mathcal{E}(S)))^2, \tag{14.2}$$

In terms of the accuracy predictor $\mathcal{P}$, we experimented with using perceptrons with different numbers of layers and different activation functions. We also explore the influence of training strategy and the amount of data being used during training (Section 14.3). Comparisons between different encoding schemes, as well as different accuracy predictors, are shown in the Appendix of [50]. Briefly, a predictor using one-hot encoding with 2 layers and tanh activation, trained with 1000 $\{S, Acc\}$ data and Adam, can achieve 0.88 Spearman correlation on the validation set.

Quantization-aware Selection

In order to select appropriate quantization bitwidth for each operation, we consider the effect of quantization degradation on top of the accuracy predictor. After phase 1 (elastic width and layer-wise distillation), all the NN parameters and activations are stored in 32-bit floating-point precision. By applying sensitivity analysis to each operation, we can simulate the influence of different quantization bitwidths. Specifically, given a tensor x, Q_i is the quantizer (quantization mapping function) with respect to bitwidth i. We measure $\|Q_i(x) - x\|_2^2$ for every operation, as well as the accuracy degradation ΔAcc on validation set with 8-bit

[2] In the Appendix of [50] we show detailed results and emphasize that the Index encoding scheme is sub-optimal since it induces additional correlations between different candidate operations that have adjacent indexes. However, Index encoding is more compact than One-hot and is generally easier for the accuracy predictor to train on.

quantization. Based on [52], the quantization degradation of different bitwidth is approximately proportional to the quantization mismatch $\|Q_i(x) - x\|_2^2$. As such, we can quickly simulate the influence of quantization on every candidate operation, combining it into the accuracy predictor $\mathcal{P}$, and then apply quantization-aware neural architecture optimization.

Non-linear Integer Optimization

Based on the latency look-up table Lat that we collected for each operation, together with the accuracy predictor $\mathcal{P}$, we now apply a non-linear optimization to solve for decent student neural architectures S that have high accuracy while satisfying the latency constraint C:

$$\text{Objective: } \max_S \mathcal{P}(\mathcal{E}(S)), \tag{14.3}$$

$$\text{Subject to: } \sum\nolimits_{i=1}^{N} Lat(s_i) \le C \tag{14.4}$$

where N is the number of layers in student architecture S and s_i represents the i^{th} block. Our implementation is based on Gekko [13], which is a general-purpose optimization library on Python. We apply the non-dynamic steady mode of Gekko and we iteratively solve for multiple solutions satisfying different constraints.

Although accuracy predictors have been actively used in neural architecture search algorithms, it should be noticed that our method is the first to attempt to use it as an objective under maximization, which can be less time-consuming and computationally costly compared to previous arts. Additionally, our method is more accurate than the integer linear optimization counterpart used in [174], which assumes an additive relationship between contributions from different operators to the final accuracy:

$$\text{Objective: } \min\nolimits_{\{s_i\}_{i=1}^{N}} \sum\nolimits_{i=1}^{N} \Delta Acc(s_i), \tag{14.5}$$

$$\text{Subject to: } \sum\nolimits_{i=1}^{N} Lat(s_i) \le C \tag{14.6}$$

$\Delta Acc(s_i)$ measures the accuracy degradation on the validation set that is solely caused by replacing t_i in the teacher network with s_i. Note that the linear integer programming method neglects the correlation among different operators at varying depths, which can be well captured by the accuracy predictor in our non-linear integer optimization method. Experimental results and an ablation study showing the effectiveness of the non-linear integer optimization are presented in Section 14.2 and Section 14.3.

Phase 3: Finetuning the Selected Model

After selecting the operators amid the target constraint, we finetune the entire network for a final performance boost, augmented by elastic resolution that further scales up the search space.

Table 14.1: ETA results on ImageNet. ETA with both EfficientNet and GENet as teacher networks are compared with SOTA compact models and NAS searched models.

Method	Resolution	Parameters(M)	FLOPs(G)	PyTorch Latency(ms)	Top-1 Acc
EfficientNetB1[223]	240	7.8	0.7	79.2	77.70
DNA[129]	224	6.4	0.6	76.2	78.40
DONNA*[175]	224	-	-	68.1	79.50
HANT(0.45B2)[174]	260	-	-	41.7	79.69
EfficientNetB2[223]	260	9.2	1.0	104.1	80.39
HANT(0.75B2)[174]	260	-	-	84.2	80.45
EfficientNetB3	300	12	1.8	170.7	81.67
ETA-EfficientNetB2	240	8.4	0.9	74.5	80.42
GENet-Normal*[142]	192	21	2.2	49.9	79.56
GENet-Large*[142]	256	31	4.6	73.9	80.80
HANT(0.6GENetLarge*)[174]	256	19	2.7	51.6	79.81
ETA-GENetLarge	224	24	2.8	52.0	80.15

Elastic Resolution

It is known that the resolution of input images can significantly affect the final performance, while most previous works lack an automatic scheme to choose the appropriate input resolution for the target network. [223] proposes to use a compound scaling strategy to determine the input resolution, which is a heuristic method and requires a costly grid search on small-scale networks. In [224], the authors show that the input resolutions selected in [223] tend to be larger than the optimal values. Additionally, [224] takes advantage of the fact that using smaller input resolution during training can improve the test accuracy, and proposes to use progressive training to gradually increase the training input resolution. However, the input resolution during the inference in [224] is still chosen by a pre-searched compound scaling rule. For teacher-student methods such as [174], the input resolution is set to be the same as the teacher network, which can be sub-optimal given that the student networks tend to have fewer parameters and FLOPs compared to the teacher.

In this work, we apply elastic resolution to further increase the search space of neural architectures. During phase 3, we apply progressive training to make sure the trained network can be adept at different target resolutions without extra fine-tuning. As such, we can generate a Pareto frontier between accuracy and latency by directly modifying the input resolutions. We then compare different candidates and select the one that has the highest accuracy while being feasible on the target hardware platform. We include more details in Section 14.3.

14.2 Experiments

CNN-based ETA on ImageNet

In Table 14.1, we present the results of our method ETA using different CNN models as the teacher network. PyTorch latency of specific models in the table are measured on Nvidia V100 GPU, by running on the ImageNet validation set and calculating the time cost of one batch (128 images). For a fair comparison, the PyTorch latency of previous works are collected on the same system as ours[3]. We use the open-sourced repo [246] and follow standard training schemes. We run all fine-tunings with 200 epochs and a cosine learning rate decay schedule. In our candidate operation pool during phase 1, we include standard network operators from EfficientNet [223], EfficientNetV2 [224], ResNets [87], GENets [142], ResNest [278], and RepVGG [45] etc. As can be seen, comparing against state-of-the-art neural architecture transformation methods DNA [129], DONNA [175] and HANT [174], ETA with EfficientNetB2 as a teacher model can achieve comparable or better accuracy while having a smaller latency.

To further show the generalization ability on different teacher networks, we also apply ETA on top of the GENet-Large model. GENets [142] are neural architectures specially designed and searched to run efficiently on GPUs. It should be noted that the original GENets are dedicatedly trained with 480 epochs, joint with extensive augmentation strategies on a different code base than ours. As such, we re-run GENet-Normal and GENet-Large on timm [246] as our baselines. As shown in the table, despite the intrinsic efficiency of GENets, we are still able to compress GENet-Large by a large margin (from 74ms latency to 52ms). ETA can achieve 80.15% Top-1 accuracy on ImageNet with a comparable latency as GENet-Normal. This is also superior to HANT accuracy (79.81%) with almost the same latency.

Transformer-based ETA on ImageNet

Transformer-based neural architectures have recently drawn great attention on vision applications. Vision transformer (ViT) [56] is the pioneering work that achieves comparable accuracy as CNNs on ImageNet. Since the mechanism of transformer-based models is different than CNNs (especially they split the input images into small patches), we further apply ETA on ViT to validate the generalization ability.

Specifically, we take ViT-Base-16 as the target teacher network, which has 12 layers with a hidden size of 768 and an MLP size of 3072. The ViT-Base-16 model has 12 attention heads and contains 76M parameters. The input resolution is 224×224, converted to patches with size 16×16.

It should be noted that ViT has been trained on different source training sets, such as ImageNet-1K, ImageNet-21K and JFT-300M, and better performance can be obtained with larger datasets. To focus on the effectiveness of searched neural architectures, we use the

[3] Since we cannot access DONNA searched models, DONNA results are converted based on its speedup over EfficientNetB0 measured on 32 images in [175].

Table 14.2: Transformer-based ETA results on ImageNet-1K.

Method	Resolution	FLOPs(G)	Latency(s)	Top-1 Acc
ViT-Base-16 [56]	224	17.6	0.56	79.67
ViT-Small-16 [276]	224	10.1	0.38	78.10
ETA-ViTBase16	224	12.5	0.43	79.05

ViT-Base-16 pretrained on ImageNet-1K as the target network without using proxy data. In contrast to CNN-based models, for our experiments on ViT, candidate operations are set to be variants of transformers with a different number of attention heads, hidden size, and MLP size. The Identity operation is also included in the operation pool to skip layers without changing the whole pipeline. From Table 14.2 we can see that ETA can compress the ViT-Base-16 model to have a comparable latency as ViT-Small-16, while being 1% higher in terms of top-1 accuracy. Our empirical results on CNNs and vision transformer variants demonstrate the universality of ETA to the model architecture.

Quantization Results of ETA

In Figure 14.1, we compare quantization results of ETA with state-of-the-art efficient solutions. DNAS [251], SPOS [76], APQ [241], OQAT [210] and FrostNet [120] are quantization-aware methods that jointly optimize neural architectures and the quantization scheme. HAQ [237], HAWQV3 [267] and BP-NAS [274] are automatic mixed-precision quantization methods that act on pretrained models with fixed neural architectures. EfficientNetB0/Lite2 [223] and ProxylessNAS [19] are compact models found through NAS, which are further quantized with uniform 8-bit in previous works. We should note that there are other previous works focusing on quantization and neural architectures that we cannot directly compare with, either due to the lack of key metrics or because they are targeting a different setting such as customized hardware platforms.

As can be seen in Figure 14.1, the accuracy of ETA can outperform previous methods by a large margin, with either 8-bit uniform quantization or mixed-precision quantization. BOPS stands for the total amount of bit operations in a given neural network, which is the standard metric to evaluate the computational complexity of quantized networks. Mixed-precision quantization results of ETA-GENetLarge can achieve fewer BOPS than HAWQV3, ResNet50+HAQ as well as quantized InceptionV4, while being comparable to BOPS of other smaller networks that have lower top-1 accuracy. It should be noted that most works in quantization-aware NAS or joint optimization are targeting on edge devices with ultra-small computational resources (consequently, we cannot include them in Figure 14.1). ETA is one of the pioneers exploring the setting of mobile devices with relatively more hardware resources but a higher requirement of accuracy.

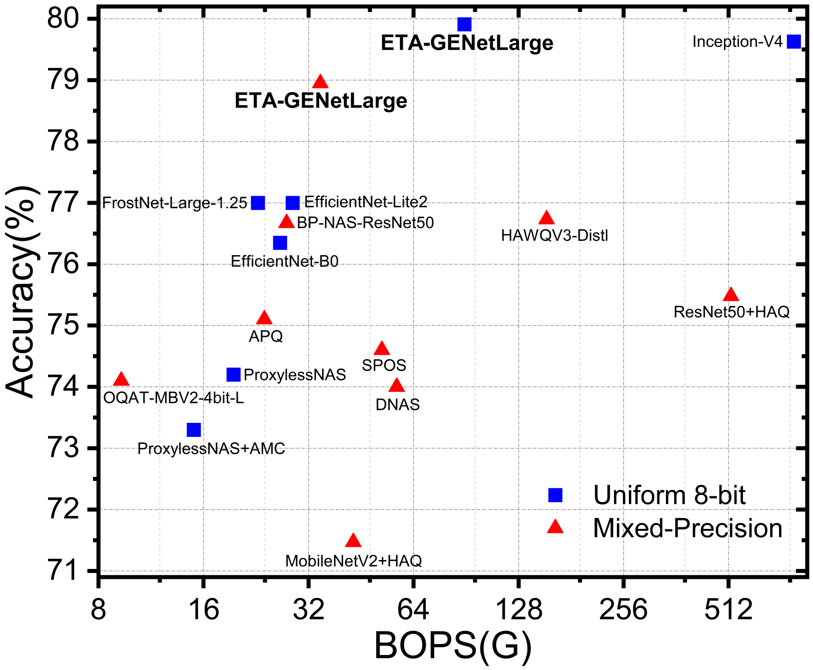


Figure 14.1: Quantization-aware ETA results. BOPS is the total amount of bit operations in a neural network.

14.3 Ablation Study

Integer Non-Linear Programming vs ILP

In Table 14.3 we show the ablation study on the effectiveness of different searching methods. The k value represents the number of searched network architectures. Time cost includes the searching and evaluation time, where searching time is measured on CPU and evaluation time on GPU. We use 5% of training data to form a validation set to quickly evaluate the sampled architectures, and the selected architectures by different methods are eventually fine-tuned on ImageNet to report the final top-1 accuracy on ImageNet validation set.

Integer linear programming (ILP) method follows [174], where an additive (linear) objective is used to predict the final accuracy of different candidate networks. Specifically, after the layer-wise knowledge distillation, [174] iteratively replaces each layer with pretrained candidate operations and measures the accuracy degradation on 5% of training data. Based on Equation 14.5, ILP tries to optimize the additive objective while satisfying the latency constraint. As in Table 14.3, ILP requires generating and evaluating more than 500 samples to obtain a decent solution, while it becomes expensive to find new solutions with a large number of existing solutions. The time cost of generating 500 unique architectures drastically increases to 324 minutes, which is over 60× more than the cost of 100 samples.

Given an accuracy predictor, the target network can also be selected by making a pre-

Table 14.3: Ablation study on different optimization methods. e represents the time cost of evaluating a sampled network on the 5% training set on GPUs. Note that ILP, Predictor, and INLP have the same latency constraint here, and the time cost of collecting data for the accuracy predictor is not included since it is done once-and-for-all.

Method	k	Time Cost(m)	5%Train Acc	Top-1 Acc
ILP	50	1 + 50e	38.18	78.94
ILP	100	5 + 100e	42.92	79.12
ILP	500	324 + 500e	45.25	79.53
Predictor	50	1	42.75	79.14
Predictor	100	5	45.63	79.43
Predictor	500	266	48.66	79.95
INLP (proposed)	5	10	50.47	80.20

diction on a pool of architectures satisfying the latency constraint (denoted as Predictor in Table 14.3). Although the evaluation time of sampled architectures can be saved due to the effectiveness of the accuracy predictor, the sampling under latency constraints is still inefficient and can lead to a large time cost. In contrast, our integer non-linear programming (INLP) method in ETA can quickly find solutions without extra evaluation or sample generation. As can be seen, the 50.47% accuracy found by INLP outperforms Predictor by a large margin, while incurring a lower time cost.

The Effectiveness of Accuracy Predictor

The success of nonlinear programming depends on the performance of the accuracy predictor. As discussed in Section 14.1, the encoding scheme of neural networks, the architecture of the accuracy predictor, the training strategy, and the amount of data can potentially affect the performance. In this ablation study, we evaluate the influence of the last two factors (more ablation study is shown in the Appendix of [50]).

Table 14.4: Effect of different optimizers and the amount of data on the performance of the accuracy predictor.

Data Amount	Constraint	Optimizer	Spearman Correlation
100	Single	SGD	0.71
1000	Multi	SGD	0.82
2000	Multi	SGD	0.83
100	Single	Adam	0.75
1000	Multi	Adam	0.88
2000	Multi	Adam	0.86

In Table 14.4, the Constraint factor represents whether the neural architectures are collected under the same latency constraint (Single), or multiple latency constraints (Multi). For the Multi setting, we use 10 different levels of latency constraints ranging from 0.25 to 0.80, with each of them contributing equally to the total amount of data. The amount of data listed in Table 14.4 is the training set for the accuracy predictor, while we have a separate validation set with 500 extra architectures. We calculate the Spearman Correlation on the validation set between rankings generated by the accuracy predictor and the rankings evaluated on 5% of the training dataset. As can be seen, the Multi setting generally performs better than the Single because of more training data, while the advantage of having more data can saturate after it contains enough information of the search space. In addition, the accuracy predictor works well with both SGD and Adam optimizers, with Adam being slightly better (0.88 vs 0.82 for SGD).

Note that our accuracy predictor in ETA models the validation accuracy on 5% of training data rather than the final validation accuracy after training. As such, there is no training involved in phase 2 of ETA after the pretraining in phase 1, making our accuracy predictor more cost-friendly compared to the counterpart in [175]. In comparison to ILP, the overhead caused by having an accuracy predictor is mainly the time to generate training data, since the training of the accuracy predictor itself can be finished within a minute on GPU. From Table 14.3 we know that collecting a small number of architectures (100 as an example) under certain constraints requires trivial time cost, making the Multi setting in Table 14.4 practical.

In Figure 14.2, we further compare the effectiveness of the accuracy predictor and the additive objective used in ILP. Accuracy predictor in ETA has a comparable range of value with the actual accuracy, while the additive objective values are located in a narrow region away from the accuracy range. In terms of ranking, the accuracy predictor also shows superiority over the additive objective, which has difficulty distinguishing specific models.

Table 14.5: Ablation Study on Elastic Resolution.

Model	Resolution	Latency(ms)	Accuracy
EfficientNetB2	320	167	81.17
EfficientNetB2	288	136	80.61
EfficientNetB2	260	104	80.39
EfficientNetB2	240	86	79.57
EfficientNetB2	224	79	79.00
ETA-EfficientNetB2	320	140	80.81
ETA-EfficientNetB2	288	107	80.59
ETA-EfficientNetB2	260	82	80.43
ETA-EfficientNetB2	240	75	80.42
ETA-EfficientNetB2	224	58	79.60

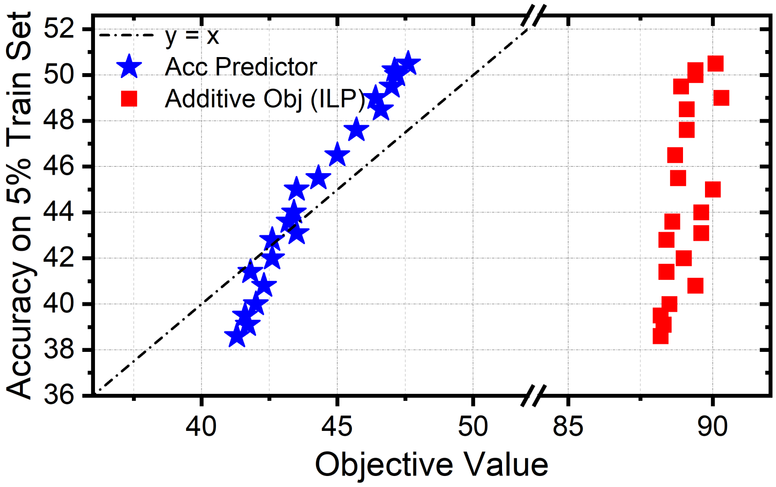


Figure 14.2: Comparison between accuracy predictor and the additive objective used in ILP [174] on 5% of training data.

Elastic Resolution

In Table 14.5, we show the effect of applying elastic resolution during progressive training. As the baseline, the EfficientNetB2 model has an original input resolution of 260. We directly evaluate EfficientNetB2 using different input resolutions without fine-tuning. As can be seen, the accuracy of EfficientNetB2 can boost to 81.17% by simply increasing the input resolution to 320. However, there is a non-trivial accuracy degradation of EfficientNetB2 with smaller resolutions. Empirically, the reason beyond this phenomenon could be that EfficientNetB2 has a high effective capacity, making the input resolution (the amount of input information) to become the bottleneck of performance. Considering that the ETA searched student networks are compressed from EfficientNetB2, which tends to have a smaller effective capacity, the optimal input resolution for EfficientNetB2 may become suboptimal. We observe this trend in Table 14.5 where the input resolution 240 shows a better latency-accuracy trade-off than 260 of the original EfficientNetB2. Consequently, we think the ability to adjust input resolution in ETA is crucial, since the bottleneck of performance can shift to network capacity after compression when the input resolution is already sufficient.

Ablation Study on Elastic Width

In Table 14.6 we show the speedup by applying elastic width during the layer-wise knowledge distillation. The operation amount counts every operation with different hyperparameters such as the expansion ratio and the kernel size. We measure GPU hours by running phase 1 for 1 epoch on 4 Nvidia V100 GPUs (32GB), with batch size 128 per GPU. As can be seen in the table, applying elastic width can significantly reduce the time cost ($\sim 2\times$) of pretraining while maintaining the final performance. Moreover, the effectiveness of elastic width is independent of the total amount of operations. We should note that elastic width is potentially able to further increase the operation amount (by exploring more options of channel width), however, it cannot bring new types of operations into the search space.

Table 14.6: Speedup achieved by applying elastic width during layer-wise pretraining.

Method	Operation Amount	GPU Hours	Top-1 Acc
Baseline	54	156	79.89
Elastic Width	54	82	79.87
Baseline	179	437	80.20
Elastic Width	179	213	80.17

Chapter 15

Conclusion on HW-SW Co-Design

Dedicated HW-SW co-design on deep learning accelerators becomes crucial and is one potential driving force for the evolution of AI processors. In this thesis, we conduct research to achieve more efficient and automatic hardware-software co-design. In hardware-aware quantization, previous works mostly use proxy efficiency metrics or simulated latency look-up tables. To address these issues, we propose HAWQV3 [267], which provides an end-to-end systematic solution to support ultra-low precision quantization on different hardware platforms. With the help of integer arithmetic and TVM auto-tuning, HAWQV3 can automatically achieve an average speedup of 1.5× for uniform 4-bit, as compared to uniform 8-bit for ResNet50 on the NVIDIA T4 GPUs.

To further increase the overall efficiency, we include hardware specifications into the search space in [102], where we co-design the deformable convolution for object detection and quantize the modified CenterNet to 4-bit weights and 8-bit activations. We show the speed-accuracy trade-offs for a set of algorithm modifications including irregular-access versus limited-range and fixed shape on a flexible FPGA accelerator. Our solution reaches 26.9 frames per second with a tiny model size of 0.76 MB while achieving 61.7 AP50 on Pascal VOC. CoDeNet can also achieve 67.1 AP50 with only a 2.9 MB model size, which is 21× smaller but 10% more accurate than Tiny-YOLO.

Another challenge in HW-SW co-design is the formidable joint search space. Differing from existing works that rely solely on expensive learning-based approaches, our work HAO [51] incorporates integer programming to prune the design space. With low computational cost, our algorithm can generate quantized networks that achieve 72.5% ImageNet top-1 accuracy on Xilinx Zynq (ZU3EG) FPGA at framerate 50, which is 60% faster than MnasNet [225] and 135% faster than FBNet [250] with comparable accuracy.

Finally, we found that a bottleneck to the efficiency of the HW-SW co-design pipeline is the pretraining process, especially when there are too many operations to be searched. In ETA [50], we apply block-wise knowledge distillation to efficiently pretrain a pool of SOTA operations for each layer, and then use an accuracy predictor to rank the potential architectures. ETA applies integer non-linear programming to maximize the output of the predictor. As a result, ETA can quickly solve for architectures under the target latency. The EfficientNetB2-based model of ETA obtains 80.42% top-1 accuracy, which is 2.72% improvement over EfficientNetB1 while running 7% faster on NVIDIA V100 GPUs.

Chapter 16

Conclusions

16.1 Impact of our work

Our quantization works HAWQ [53], HAWQV2 [52], QBERT [211], ZeroQ [21] were published in major AI conferences ICCV, NeurIPS, AAAI, CVPR, respectively. The systematic mixed-precision quantization approach we developed also drew significant attention from the industry. We presented our works at NVIDIA GTC 2021, as well as at seminars held by Google, Meta, Intel, Alibaba, Amazon, Apple, Tesla, AMD, ByteDance, Panasonic, etc. Our open-sourced codes on github became one of the top quantization tools publicly available. As an example of our use of mixed-precision quantization, we were able to achieve a 10 × compression ratio on ImageNet with only around 1% accuracy drop, and our method works well across a broad range of models and tasks. Moreover, our works were introduced by many media focusing on AI, such as Synced AI, Jiangmen AI, AI.Science, etc. Furthermore, we conducted a survey paper [69] summarizing the recent advancement of quantization, which was published as a section of the book of Low-Power Computer Vision [230].

Our HW-SW co-design works HAWQV3 [267], CoDeNet [102], HAO [51] were published in major AI and FPGA conferences ICML, FPGA, FCCM, respectively. The HW-SW co-design works also attracted interest from the industry. We presented our works at TVM Conference 2020, and at seminars held by the aforementioned companies. With our solutions, we won second place in the Visual Wake Word Challenge at CVPR 2019, and first place in the EMCC 2020 competition on both classification and object detection tracks. As an example of our use of HW-SW co-design, our 4-bit/8-bit mixed-precision model gets 67.1 AP50 on Pascal VOC with only 2.9 MB size, which is 21 × smaller but 10% more accurate than Tiny-YOLO. Moreover, we were invited by the scientific computing community to write a chapter about model compression and HW-SW co-design in the survey paper [42], which was published in the Frontiers of Big Data.

16.2 Future work

In this section, we briefly discuss several high-level challenges and opportunities for future research in quantization and HW-SW co-design.

Support for Quantization Deployment: As discussed in this thesis, it is currently straightforward to quantize and deploy different NN models to uniform INT8 without losing accuracy. There are several software packages that can be used to deploy INT8 quantized models (e.g., Nvidia's TensorRT, TVM, etc.), each with good documentation. However, the support for lower bit-width quantization is not widely available. For instance, Nvidia's TensorRT does not currently support sub-INT8 quantization. Although mixed-precision quantization is well supported on FPGA boards, its speedup is non-trivial to leverage on other hardware platforms such as GPUs. Consequently, developing efficient APIs and support for lower precision and mixed-precision quantization will have an important impact.

Quantization and Co-design of Novel Architectures and Applications: After the success of CNNs on standard computer vision tasks (classification, object detection, and semantic segmentation) and transformers on NLP tasks, new architectures of NN have been introduced that can achieve comparable or even better performance. Vision transformers [56, 232, 155] and mlp-based networks [231, 66, 162] have different operations and bottlenecks than CNNs, making it sub-optimal to directly apply quantization or co-design methods developed for CNNs. Based on our previous experiments, we found that transformers or mlp-based models tend to have significantly larger activation ranges than CNNs, making them difficult for standard quantization [282]. Besides, neural networks have been used in many new applications, such as text-to-image generation, 3D object detection in Bird-Eye-View (BEV), few-shot learning, etc. HW-SW co-design can potentially improve the performance of these applications by a large margin based on their specific characteristics. Finally, based on our experiments, emerging state-of-the-art pretrained models with a tremendous parameter size (for example, GPT-3 [18]) provide more challenges for quantization and HW-SW co-design, which could be important topics in the future.

Efficient Training with Quantization: Half-precision quantization has been widely used to accelerate NN training [39, 77, 72, 171]. However, it has been very difficult to push this further down to INT8 training. While several interesting works exist in this area [116, 168, 11, 125, 24], the proposed methods often require a lot of hyperparameter tuning, or they only work for a few NN models on relatively easy learning tasks. The basic problem is that, with INT8 precision, the training can become unstable and diverge. Addressing this challenge can have a high impact on several applications [47], especially for training at the edge. Furthermore, for on-device training with limited resources, other model compression methods (for example, pruning or sparsity) can be jointly used with quantization to boost the compression ratio.

Quantization with Unsupervised or Self-supervised Learning: Quantization and HW-SW co-design can lead to an accuracy degradation, while recent works have shown that unsupervised learning [149] or self-supervised learning [28, 29] are very capable of boosting the NN performance. Few previous works have explored whether the accuracy degradation can be fully recovered by using unlabeled data or self-supervised tasks. Additionally, the models trained with unsupervised or semi-supervised learning tend to have very high accuracy. It is interesting to see the robustness of those models to the quantization perturbation, compared to the ordinary models.

Co-Design with Other Hardware Platforms: Previous works have investigated the effect of quantization and HW-SW co-design on GPUs and FPGAs, while there are other hardware platforms to be explored. As an example, processing-in-memory (PIM) chips have the potential to significantly accelerate the inference of NNs. The values represented by the novel devices in PIM are naturally discrete [249, 99, 242], making quantization crucial in the deployment. Our previous works [291, 100, 49] showed preliminary co-design of algorithms and hardware devices, and later advanced methods have been conducted by others [219, 22]. Despite the merits, more work in the future would be necessary.

Jointly Apply Model Compression Methods: To achieve efficient deployment of NN, quantization can be jointly applied with other approaches such as pruning, knowledge distillation, and factorization. However, there is currently very little work exploring what are the optimal combinations of these methods. For instance, pruning and quantization can be applied together to a model to reduce its overhead [86, 140], but the level of pruning will affect the model's sensitivity to quantization, and it is important to understand the best combination of these different compression methods.

Bibliography

[1] Mohamed S Abdelfattah et al. “Best of Both Worlds: AutoML Codesign of a CNN and its Hardware Accelerator”. In: *arXiv preprint arXiv:2002.05022* (2020).

[2] Mohamed S Abdelfattah et al. “Codesign-NAS: Automatic FPGA/CNN Codesign Using Neural Architecture Search”. In: *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 315–315.

[3] Eirikur Agustsson and Lucas Theis. “Universally Quantized Neural Compression”. In: *Advances in neural information processing systems* (2020).

[4] Milad Alizadeh et al. “Gradient L1 Regularization for Quantization Robustness”. In: *arXiv preprint arXiv:2002.07520* (2020).

[5] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multiarmed bandit problem”. In: *Machine learning* 47.2-3 (2002), pp. 235–256.

[6] Haim Avron and Sivan Toledo. “Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix”. In: *Journal of the ACM (JACM)* 58.2 (2011), p. 8.

[7] Mart van Baalen et al. “Bayesian bits: Unifying quantization and pruning”. In: *Advances in neural information processing systems* (2020).

[8] Yu Bai, Yu-Xiang Wang, and Edo Liberty. “Proxquant: Quantized neural networks via proximal operators”. In: *arXiv preprint arXiv:1810.00861* (2018).

[9] Zhaojun Bai, Gark Fahey, and Gene Golub. “Some large-scale matrix computation problems”. In: *Journal of Computational and Applied Mathematics* 74.1-2 (1996), pp. 71–89.

[10] Ron Banner et al. “Post-training 4-bit quantization of convolution networks for rapid-deployment”. In: *arXiv preprint arXiv:1810.05723* (2018).

[11] Ron Banner et al. “Scalable methods for 8-bit training of neural networks”. In: *Advances in neural information processing systems* (2018).

[12] Chaim Baskin et al. “Uniq: Uniform noise injection for non-uniform quantization of neural networks”. In: *arXiv preprint arXiv:1804.10969* (2018).

[13] Logan D. R. Beal et al. "GEKKO Optimization Suite". In: *Processes* 6.8 (2018). ISSN: 2227-9717. DOI: 10.3390/pr6080106. URL: http://www.mdpi.com/2227-9717/6/8/106.

[14] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. "Estimating or propagating gradients through stochastic neurons for conditional computation". In: *arXiv preprint arXiv:1308.3432* (2013).

[15] Yash Bhalgat et al. "LSQ+: Improving low-bit quantization through learnable offsets and better initialization". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2020, pp. 696–697.

[16] Aishwarya Bhandare et al. "Efficient 8-bit quantization of transformer neural machine language translation model". In: *arXiv preprint arXiv:1906.00532* (2019).

[17] Michaela Blott et al. "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks". In: vol. 11. 3. ACM New York, NY, USA, 2018.

[18] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[19] Han Cai, Ligeng Zhu, and Song Han. "Proxylessnas: Direct neural architecture search on target task and hardware". In: *arXiv preprint arXiv:1812.00332* (2018).

[20] Han Cai et al. "Once-for-all: Train one network and specialize it for efficient deployment". In: *arXiv preprint arXiv:1908.09791* (2019).

[21] Yaohui Cai et al. "Zeroq: A novel zero shot quantization framework". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 13169–13178.

[22] Yi Cai et al. "Low bit-width convolutional neural network on rram". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.7 (2019), pp. 1414–1427.

[23] Zhaowei Cai et al. "Deep learning with low precision by half-wave gaussian quantization". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5918–5926.

[24] Léopold Cambier et al. "Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks". In: *arXiv preprint arXiv:2001.05674* (2020).

[25] Michail Chatzianastasis et al. "Graph-Based Neural Architecture Search With Operation Embeddings". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 393–402.

[26] Shangyu Chen, Wenya Wang, and Sinno Jialin Pan. “MetaQuant: Learning to Quantize by Learning to Penetrate Non-differentiable Quantization”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/f8e59f4b2fe7c5705bf878bbd494ccdf-Paper.pdf.

[27] Tianqi Chen et al. “TVM: An automated end-to-end optimizing compiler for deep learning”. In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 578–594.

[28] Ting Chen et al. “Big self-supervised models are strong semi-supervised learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 22243–22255.

[29] Xinlei Chen, Saining Xie, and Kaiming He. “An empirical study of training self-supervised vision transformers”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 9640–9649.

[30] Yuntao Chen et al. “Simpledet: A simple and versatile distributed framework for object detection and instance recognition”. In: *The Journal of Machine Learning Research (JMLR)* (2019).

[31] Zailong Chen et al. “LAP: Latency-aware automated pruning with dynamic-based filter selection”. In: *Neural Networks* 152 (2022), pp. 407–418.

[32] Ting-Wu Chin et al. “One Weight Bitwidth to Rule Them All”. In: *Proceedings of the European Conference on Computer Vision (ECCV)* (2020).

[33] Jungwook Choi et al. “Pact: Parameterized clipping activation for quantized neural networks”. In: *arXiv preprint arXiv:1805.06085* (2018).

[34] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. “Learning low precision deep neural networks through regularization”. In: *arXiv preprint arXiv:1809.00095* 2 (2018).

[35] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. “Towards the limit of network quantization”. In: *arXiv preprint arXiv:1612.01543* (2016).

[36] François Chollet. “Xception: Deep learning with depthwise separable convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[37] Yoni Choukroun et al. “Low-bit Quantization of Neural Networks for Efficient Inference.” In: *ICCV Workshops*. 2019, pp. 3009–3018.

[38] 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*. 2015, pp. 3123–3131.

[39] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Training deep neural networks with low precision multiplications”. In: *arXiv preprint arXiv:1412.7024* (2014).

[40] Jifeng Dai et al. "Deformable convolutional networks". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017.

[41] Xiaoliang Dai et al. "ChamNet: Towards Efficient Network Design through Platform-Aware Model Adaptation". In: *CVPR*. 2019.

[42] Allison McCarn Deiana et al. "Applications and techniques for fast machine learning in science". In: *Frontiers in big Data* 5 (2022).

[43] Lei Deng et al. "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework". In: *Neural Networks* 100 (2018), pp. 49–58.

[44] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[45] Xiaohan Ding et al. "Repvgg: Making vgg-style convnets great again". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 13733–13742.

[46] Z Dong et al. "RRAM based convolutional neural networks for high accuracy pattern recognition and online learning tasks". In: *2017 Silicon Nanoelectronics Workshop (SNW)*. IEEE. 2017, pp. 145–146.

[47] Zhen Dong, Xiaoyong Liu, and Kurt Keutzer. "Addressing Challenges in Large-scale Distributed AI Systems". In: (2022).

[48] Zhen Dong et al. "Convolutional neural networks based on RRAM devices for image recognition and online learning tasks". In: *IEEE Transactions on Electron Devices* 66.1 (2018), pp. 793–801.

[49] Zhen Dong et al. "Design Considerations of Large-Scale RRAM-Based Convolutional Neural Networks with Transfer Learning". In: (2018).

[50] Zhen Dong et al. "Efficient transformation of architectures through hardware-aware nonlinear optimization". In: 2022.

[51] Zhen Dong et al. "Hao: Hardware-aware neural architecture optimization for efficient inference". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2021, pp. 50–59.

[52] Zhen Dong et al. "HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks". In: *Advances in neural information processing systems* (2020).

[53] Zhen Dong et al. "Hawq: Hessian aware quantization of neural networks with mixed-precision". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 293–302.

[54] Zhen Dong et al. *Trace weighted hessian-aware quantization.*

[55] Zhen Dong et al. "UnrealNAS: Can We Search Neural Architectures with Unreal Data?" In: *arXiv preprint arXiv:2205.02162* (2022).

[56] Alexey Dosovitskiy et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020).

[57] Steven K Esser et al. “Learned step size quantization”. In: *arXiv preprint arXiv:1902.08153* (2019).

[58] Angela Fan et al. “Training with quantization noise for extreme model compression”. In: *arXiv e-prints* (2020), arXiv–2004.

[59] Jun Fang et al. “Near-Lossless Post-Training Quantization of Deep Neural Networks via a Piecewise Linear Approximation”. In: *arXiv preprint arXiv:2002.00104* (2020).

[60] Jun Fang et al. “Post-training piecewise linear quantization for deep neural networks”. In: *European Conference on Computer Vision*. Springer. 2020, pp. 69–86.

[61] Julian Faraone et al. “Syq: Learning symmetric quantization for efficient deep neural networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4300–4309.

[62] Fasih Ud Din Farrukh et al. “Power Efficient Tiny Yolo CNN using Reduced Hardware Resources based on Booth Multiplier and WALLACE Tree Adders”. In: *IEEE Open Journal of Circuits and Systems* (2020).

[63] Alexander Finkelstein, Uri Almog, and Mark Grobman. “Fighting quantization bias with bias”. In: *arXiv preprint arXiv:1906.03193* (2019).

[64] Abram L Friesen and Pedro Domingos. “Deep learning as a mixed convex-combinatorial optimization problem”. In: *arXiv preprint arXiv:1710.11573* (2017).

[65] Yao Fu et al. “Deep learning with int8 optimization on xilinx devices”. In: *White Paper* (2016).

[66] Francesco Fusco, Damian Pascual, and Peter Staar. “pNLP-Mixer: an Efficient all-MLP Architecture for Language”. In: *arXiv preprint arXiv:2202.04350* (2022).

[67] Sahaj Garg et al. “Confounding Tradeoffs for Neural Network Quantization”. In: *arXiv preprint arXiv:2102.06366* (2021).

[68] Sahaj Garg et al. “Dynamic Precision Analog Computing for Neural Networks”. In: *arXiv preprint arXiv:2102.06365* (2021).

[69] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[70] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference. arXiv 2021”. In: *arXiv preprint arXiv:2103.13630* (2021).

[71] Amir Gholami et al. “SqueezeNext: Hardware-Aware Neural Network Design”. In: *Workshop paper in CVPR* (2018).

[72] Boris Ginsburg et al. *Tensor processing using low precision format*. US Patent App. 15/624,577. Dec. 2017.

[73] Yunchao Gong et al. “Compressing deep convolutional networks using vector quantization”. In: *arXiv preprint arXiv:1412.6115* (2014).

[74] Kaiyuan Guo et al. “Software-hardware codesign for efficient neural network acceleration”. In: *IEEE Micro* 37.2 (2017), pp. 18–25.

[75] Yiwen Guo et al. “Network sketching: Exploiting binary structure in deep cnns”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2017, pp. 5955–5963.

[76] Zichao Guo et al. “Single path one-shot neural architecture search with uniform sampling”. In: *European Conference on Computer Vision.* Springer. 2020, pp. 544–560.

[77] Suyog Gupta et al. “Deep learning with limited numerical precision”. In: *International conference on machine learning.* PMLR. 2015, pp. 1737–1746.

[78] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. “Hardware-oriented approximation of convolutional neural networks”. In: *arXiv preprint arXiv:1604.03168* (2016).

[79] Philipp Gysel et al. “Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks”. In: *IEEE transactions on neural networks and learning systems* 29.11 (2018), pp. 5784–5789.

[80] Hai Victor Habi, Roy H Jennings, and Arnon Netzer. “HMQ: Hardware Friendly Mixed Precision Quantization Block for CNNs”. In: *arXiv preprint arXiv:2007.09952* (2020).

[81] Runze Han et al. “A novel convolution computing paradigm based on NOR flash array with high computing speed and energy efficiency”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.5 (2019), pp. 1692–1703.

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

[83] Cong Hao et al. “FPGA/DNN Co-Design: An Efficient Design Methodology for 1oT Intelligence on the Edge”. In: *2019 56th ACM/IEEE Design Automation Conference (DAC).* IEEE. 2019, pp. 1–6.

[84] Cong Hao et al. “NAIS: Neural architecture and implementation search and its applications in autonomous driving”. In: *arXiv preprint arXiv:1911.07446* (2019).

[85] Matan Haroush et al. “The Knowledge Within: Methods for Data-Free Model Compression”. In: *arXiv preprint arXiv: 1912.01274* (2019).

[86] Benjamin Hawks et al. “Ps and Qs: Quantization-aware pruning for efficient low latency neural network inference”. In: *arXiv preprint arXiv:2102.11289* (2021).

[87] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[88] Kaiming He et al. “Identity mappings in deep residual networks”. In: *European conference on computer vision*. Springer. 2016, pp. 630–645.

[89] Xiangyu He and Jian Cheng. “Learning compression from limited unlabeled data”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 752–769.

[90] Yihui He et al. “Amc: Automl for model compression and acceleration on mobile devices”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 784–800.

[91] Sepp Hochreiter and Jürgen Schmidhuber. “Flat minima”. In: *Neural Computation* 9.1 (1997), pp. 1–42.

[92] Wenjing Hong et al. “Multi-objective magnitude-based pruning for latency-aware deep neural network compression”. In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2020, pp. 470–483.

[93] Mark Horowitz. “1.1 computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE. 2014, pp. 10–14.

[94] Lu Hou, Quanming Yao, and James T Kwok. “Loss-aware binarization of deep networks”. In: *arXiv preprint arXiv:1611.01600* (2016).

[95] Andrew Howard et al. “Searching for MobilenetV3”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 1314–1324.

[96] Andrew G Howard et al. “MobileNets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[97] Peng Hu et al. “OPQ: Compressing Deep Neural Networks with One-shot Pruning-Quantization”. In: (2021).

[98] Gao Huang et al. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[99] P Huang et al. “RTN based oxygen vacancy probing method for Ox-RRAM reliability characterization and its application in tail bits”. In: *2017 IEEE International Electron Devices Meeting (IEDM)*. IEEE. 2017, pp. 21–4.

[100] Peng Huang et al. “Binary resistive-switching-device-based electronic synapse with Spike-Rate-Dependent plasticity for online learning”. In: *ACS Applied Electronic Materials* 1.6 (2019), pp. 845–853.

[101] Qijing Huang et al. “Algorithm-hardware co-design for deformable convolution”. In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE. 2019, pp. 48–51.

[102] Qijing Huang et al. “CoDeNet: Efficient Deployment of Input-Adaptive Object Detection on Embedded FPGAs”. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2021, pp. 206–216.

[103] Itay Hubara et al. “Binarized neural networks”. In: *Advances in neural information processing systems*. 2016, pp. 4107–4115.

[104] Itay Hubara et al. “Improving post training neural quantization: Layer-wise calibration and integer programming”. In: *arXiv preprint arXiv:2006.10518* (2020).

[105] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size”. In: *arXiv preprint arXiv:1602.07360* (2016).

[106] Yani Ioannou et al. “Deep roots: Improving cnn efficiency with hierarchical filter groups”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1231–1240.

[107] Benoit Jacob et al. “Quantization and training of neural networks for efficient integer-arithmetic-only inference”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

[108] Animesh Jain et al. “Efficient execution of quantized deep learning models: A compiler approach”. In: *arXiv preprint arXiv:2006.10226* (2020).

[109] Shubham Jain et al. “BiScaled-DNN: Quantizing long-tailed datastructures with two scale factors for deep neural networks”. In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6.

[110] Eric Jang, Shixiang Gu, and Ben Poole. “Categorical reparameterization with gumbel-softmax”. In: *arXiv preprint arXiv:1611.01144* (2016).

[111] Yongkweon Jeon et al. “BiQGEMM: matrix multiplication with lookup table for binary-coding-based quantized DNNs”. In: *arXiv preprint arXiv:2005.09904* (2020).

[112] Weiwen Jiang et al. “Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.

[113] Weiwen Jiang et al. “Hardware/Software co-exploration of neural architectures”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[114] Weiwen Jiang et al. “Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 4154–4165.

[115] Li Jiao et al. “Accelerating low bit-width convolutional neural networks with embedded FPGA”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2017, pp. 1–4.

[116] Jeff Johnson. “Rethinking floating point for deep learning”. In: *arXiv preprint arXiv:1811.01721* (2018).

[117] Sangil Jung et al. “Learning to quantize deep networks by optimizing quantization intervals with task loss”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4350–4359.

[118] PP Kanjilal, PK Dey, and DN Banerjee. “Reduced-size neural networks through singular value decomposition and subset selection”. In: *Electronics Letters* 29.17 (1993), pp. 1516–1518.

[119] Sehoon Kim et al. “I-BERT: Integer-only BERT Quantization”. In: *arXiv preprint arXiv:2101.01321* (2021).

[120] Taehoon Kim, YoungJoon Yoo, and Jihoon Yang. “FrostNet: Towards Quantization-Aware Network Architecture Search”. In: *arXiv preprint arXiv:2006.09679* (2020).

[121] Levente Kocsis and Csaba Szepesvári. “Bandit based monte-carlo planning”. In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[122] Eli Kravchik et al. “Low-bit Quantization of Neural Networks for Efficient Inference”. In: *The IEEE International Conference on Computer Vision (ICCV) Workshops*. Oct. 2019.

[123] Raghuraman Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper”. In: *arXiv preprint arXiv:1806.08342* (2018).

[124] Raghuraman Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper”. In: *arXiv preprint arXiv:1806.08342* (2018).

[125] Hamed F Langroudi et al. “Cheetah: Mixed low-precision hardware & software co-design framework for dnns on the edge”. In: *arXiv preprint arXiv:1908.02386* (2019).

[126] Dong-Hyun Lee et al. “Difference target propagation”. In: *Joint european conference on machine learning and knowledge discovery in databases*. Springer. 2015, pp. 498–515.

[127] Jun Haeng Lee et al. “Quantization for rapid deployment of deep neural networks”. In: *arXiv preprint arXiv:1810.05488* (2018).

[128] Cong Leng et al. “Extremely low bit neural network: Squeeze the last bit out with admm”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 2018.

[129] Changlin Li et al. “Block-wisely supervised neural architecture search with knowledge distillation”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 1989–1998.

[130] Rundong Li et al. “Fully quantized network for object detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[131] Shuai Li et al. “Novel CNN-Based AP2D-Net Accelerator: An Area and Power Efficient Solution for Real-Time Applications on Mobile FPGA”. In: *Electronics* 9.5 (2020), p. 832.

[132] Tian Li et al. “Cross-domain sentiment classification with contrastive learning and mutual information maximization”. In: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2021, pp. 8203–8207.

[133] Tian Li et al. “Cross-domain sentiment classification with in-domain contrastive learning”. In: *arXiv preprint arXiv:2012.02943* (2020).

[134] Tian Li et al. “Domain-Adaptive Text Classification with Structured Knowledge from Unlabeled Data”. In: *arXiv preprint arXiv:2206.09591* (2022).

[135] Wei Li, Shaogang Gong, and Xiatian Zhu. “Neural graph embedding for neural architecture search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 4707–4714.

[136] Yanghao Li et al. “Scale-aware trident networks for object detection”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2019.

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

[138] Yuhang Li et al. “BRECQ: Pushing the Limit of Post-Training Quantization by Block Reconstruction”. In: *International Conference on Learning Representations* (2021).

[139] Yuhong Li et al. “EDD: Efficient Differentiable DNN Architecture and Implementation Co-search for Embedded AI Solutions”. In: *arXiv preprint arXiv:2005.02563* (2020).

[140] Tailin Liang et al. “Pruning and Quantization for Deep Neural Network Acceleration: A Survey”. In: *arXiv preprint arXiv:2101.09671* (2021).

[141] Zhenyu Liao, Romain Couillet, and Michael W Mahoney. “Sparse quantized spectral clustering”. In: *International Conference on Learning Representations* (2021).

[142] Ming Lin et al. “Neural architecture design for gpu-efficient networks”. In: *arXiv preprint arXiv:2006.14090* (2020).

[143] Tsung-Yi Lin et al. “Feature Pyramid Networks for Object Detection”. In: *CoRR* abs/1612.03144 (2016). arXiv: 1612.03144. URL: http://arxiv.org/abs/1612.03144.

[144] Tsung-Yi Lin et al. “Focal loss for dense object detection”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017.

[145] Tsung-Yi Lin et al. “Microsoft coco: Common objects in context”. In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[146] Wuwei Lin. *Automating Optimization of Quantized Deep Learning Models on CUDA: https://tvm.apache.org/2019/04/29/opt-cuda-quantized*. 2019.

[147] Xiaofan Lin, Cong Zhao, and Wei Pan. “Towards accurate binary convolutional neural network”. In: *arXiv preprint arXiv:1711.11294* (2017).

[148] Zhouhan Lin et al. “Neural networks with few multiplications”. In: *arXiv preprint arXiv:1510.03009* (2015).

[149] Chenxi Liu et al. "Are labels necessary for neural architecture search?" In: *European Conference on Computer Vision*. Springer. 2020, pp. 798–813.

[150] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search". In: *arXiv preprint arXiv:1806.09055* (2018).

[151] Hongyang Liu et al. "Layer Importance Estimation With Imprinting for Neural Network Quantization". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 2408–2417.

[152] Shu Liu et al. "Path aggregation network for instance segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

[153] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision (ECCV)*. 2016.

[154] Yinhan Liu et al. "RoBERTa: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[155] Ze Liu et al. "Swin transformer: Hierarchical vision transformer using shifted windows". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 10012–10022.

[156] Zechun Liu et al. "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)*. 2018, pp. 722–737.

[157] Zhi-Gang Liu and Matthew Mattina. "Learning low-precision neural networks without straight-through estimator (STE)". In: *arXiv preprint arXiv:1903.01061* (2019).

[158] Qing Lu et al. "On neural architecture search for resource-constrained hardware platforms". In: *arXiv preprint arXiv:1911.00105* (2019).

[159] M. W. Mahoney, J. C. Duchi, and A. C. Gilbert, eds. *The Mathematics of Data*. IAS/Park City Mathematics Series. AMS, IAS/PCMI, and SIAM, 2018.

[160] Ningning Ma et al. "Shufflenet V2: Practical guidelines for efficient cnn architecture design". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 116–131.

[161] Xindian Ma et al. "A tensorized transformer for language modeling". In: *Advances in Neural Information Processing Systems*. 2019, pp. 2229–2239.

[162] Xu Ma et al. "Rethinking network design and local geometry in point cloud: A simple residual MLP framework". In: *arXiv preprint arXiv:2202.07123* (2022).

[163] Yufei Ma et al. "Algorithm-hardware co-design of single shot detector for fast object detection on FPGAs". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[164] Michael W Mahoney. *Randomized algorithms for matrices and data*. Foundations and Trends in Machine Learning. Boston: NOW Publishers, 2011.

[165] Franck Mamalet and Christophe Garcia. “Simplifying convnets for fast learning”. In: *International Conference on Artificial Neural Networks*. Springer. 2012, pp. 58–65.

[166] James Martens. “Deep learning via Hessian-free optimization.” In: *ICML*. Vol. 27. 2010, pp. 735–742.

[167] Jeffrey L McKinstry et al. “Discovering low-precision networks close to full-precision networks for efficient embedded inference”. In: *arXiv preprint arXiv:1809.04191* (2018).

[168] Naveen Mellempudi et al. “Mixed precision training with 8-bit floating point”. In: *arXiv preprint arXiv:1905.12334* (2019).

[169] Eldad Meller et al. “Same, same but different: Recovering neural network quantization error through weight factorization”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 4486–4495.

[170] Paul Michel, Omer Levy, and Graham Neubig. “Are sixteen heads really better than one?” In: *arXiv preprint arXiv:1905.10650* (2019).

[171] Paulius Micikevicius et al. “Mixed precision training”. In: *arXiv preprint arXiv:1710.03740* (2017).

[172] Szymon Migacz. “Nvidia 8-bit inference with TensorRT”. In: *GPU Technology Conference* (2017).

[173] Daisuke Miyashita, Edward H Lee, and Boris Murmann. “Convolutional neural networks using logarithmic data representation”. In: *arXiv preprint arXiv:1603.01025* (2016).

[174] Pavlo Molchanov et al. “HANT: Hardware-Aware Network Transformation”. In: *arXiv preprint arXiv:2107.10624* (2021).

[175] Bert Moons et al. “Distilling optimal neural networks: Rapid search in diverse spaces”. In: *arXiv preprint arXiv:2012.08859* (2020).

[176] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. “Inceptionism: Going deeper into neural networks”. In: (2015).

[177] Markus Nagel et al. “A White Paper on Neural Network Quantization”. In: *arXiv preprint arXiv:2106.08295* (2021).

[178] Markus Nagel et al. “Data-free quantization through weight equalization and bias correction”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1325–1334.

[179] Markus Nagel et al. “Up or down? adaptive rounding for post-training quantization”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7197–7206.

[180] Hiroki Nakahara et al. “A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM. 2018, pp. 31–40.

[181] Maxim Naumov et al. "On periodic functions as regularizers for quantization of neural networks". In: *arXiv preprint arXiv:1811.09862* (2018).

[182] Renkun Ni et al. "WrapNet: Neural Net Inference with Ultra-Low-Resolution Arithmetic". In: *arXiv preprint arXiv:2007.13242* (2020).

[183] Lin Ning et al. "Simple Augmentation Goes a Long Way: {ADRL} for {DNN} Quantization". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=Qr0aRliE_Hb.

[184] Xuefei Ning et al. "A generic graph-based neural architecture encoding scheme for predictor-based nas". In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIII 16*. Springer. 2020, pp. 189–204.

[185] NVIDIA. *Cutlass Library*. 2020. URL: https://github.com/NVIDIA/cutlass.

[186] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. "Weighted-entropy-based quantization for deep neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5456–5464.

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

[188] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[189] Hieu Pham et al. "Efficient neural architecture search via parameters sharing". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4095–4104.

[190] Thomas B Preußer et al. "Inference of quantized neural networks on heterogeneous all-programmable devices". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 833–838.

[191] Jiantao Qiu et al. "Going deeper with embedded fpga platform for convolutional neural network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 26–35.

[192] Zhongnan Qu et al. "Adaptive Loss-Aware Quantization for Multi-Bit Networks". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[193] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[194] Pranav Rajpurkar et al. "SQuAD: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250* (2016).

[195] Prajit Ramachandran, Barret Zoph, and Quoc V Le. "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941* (2017).

[196] Mohammad Rastegari et al. “Xnor-net: Imagenet classification using binary convolutional neural networks”. In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[197] Joseph Redmon and Ali Farhadi. “YOLO9000: better, faster, stronger”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[198] Jorma Rissanen. “Modeling by shortest data description”. In: *Automatica* 14.5 (1978), pp. 465–471.

[199] Frank Rosenblatt. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[200] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[201] J.S. Roy and S.A. Mitchell. “PuLP is an LP modeler written in Python”. In: (2020). URL: https://github.com/coin-or/pulp.

[202] Manuele Rusci et al. “Leveraging Automated Mixed-Low-Precision Quantization for Tiny Edge Microcontrollers”. In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Springer, 2020, pp. 296–308.

[203] Tara N Sainath et al. “Low-rank matrix factorization for deep neural network training with high-dimensional output targets”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6655–6659.

[204] Dave Salvator et al. *Int4 Precision for AI Inference: https://developer.nvidia.com/blog/int4-for-ai-inference/*. 2019.

[205] Mark Sandler et al. “MobilenetV2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520.

[206] Erik F Sang and Fien De Meulder. “Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition”. In: *cs/0306050* (2003).

[207] Florian Scheidegger et al. “Constrained deep neural network architecture search for IoT devices accounting for hardware calibration”. In: *Advances in Neural Information Processing Systems*. 2019, pp. 6056–6066.

[208] Evan Shelhamer, Dequan Wang, and Trevor Darrell. “Blurring the line between structure and learning to optimize and adapt receptive fields”. In: *arXiv preprint arXiv:1904.11487* (2019).

[209] Maying Shen et al. “HALP: Hardware-Aware Latency Pruning”. In: *arXiv preprint arXiv:2110.10811* (2021).

[210] Mingzhu Shen et al. "Once Quantization-Aware Training: High Performance Extremely Low-Bit Architecture Search". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 5340–5349.

[211] Sheng Shen et al. "Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT." In: *AAAI*. 2020, pp. 8815–8821.

[212] Moran Shkolnik et al. "Robust quantization: One model to rule them all". In: *Advances in neural information processing systems* (2020).

[213] Gil Shomron et al. "Post-Training Sparsity-Aware Quantization". In: *arXiv preprint arXiv:2105.11010* (2021).

[214] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*. 2015.

[215] Richard Socher et al. "Recursive deep models for semantic compositionality over a sentiment treebank". In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[216] Pierre Stock et al. "Training with Quantization Noise for Extreme Model Compression". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=dV19Yyi1fS3.

[217] Naveen Suda et al. "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 16–25.

[218] Siqi Sun et al. "Patient knowledge distillation for bert model compression". In: *arXiv preprint arXiv:1908.09355* (2019).

[219] Xiaoyu Sun et al. "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 1423–1428.

[220] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *Thirteenth annual conference of the international speech communication association*. 2012.

[221] Christian Szegedy et al. "Rethinking the Inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[222] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. "Degree-Quant: Quantization-Aware Training for Graph Neural Networks". In: *International Conference on Learning Representations* (2021).

[223] Mingxing Tan and Quoc V Le. "EfficientNet: Rethinking model scaling for convolutional neural networks". In: *arXiv preprint arXiv:1905.11946* (2019).

[224] Mingxing Tan and Quoc V Le. "Efficientnetv2: Smaller models and faster training". In: *arXiv preprint arXiv:2104.00298* (2021).

[225] Mingxing Tan et al. “Mnasnet: Platform-aware neural architecture search for mobile”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[226] Raphael Tang et al. “Distilling task-specific knowledge from bert into simple neural networks”. In: *arXiv preprint arXiv:1903.12136* (2019).

[227] Wei Tang, Gang Hua, and Liang Wang. “How to train a compact binary neural network with high accuracy?” In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 2017.

[228] Yehui Tang et al. “A Semi-Supervised Assessor of Neural Architectures”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 1810–1819.

[229] Yi Tay et al. “Lightweight and Efficient Neural Natural Language Processing with Quaternion Networks”. In: *arXiv:1906.04393* (2019).

[230] George K Thiruvathukal et al. *Low-Power Computer Vision: Improve the Efficiency of Artificial Intelligence*. CRC Press, 2022.

[231] Hugo Touvron et al. “Resmlp: Feedforward networks for image classification with data-efficient training”. In: *arXiv preprint arXiv:2105.03404* (2021).

[232] Hugo Touvron et al. “Training data-efficient image transformers & distillation through attention”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 10347–10357.

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

[234] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[235] Alvin Wan et al. “Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 12965–12974.

[236] Kuan Wang et al. “HAQ: Hardware-Aware Automated Quantization”. In: *In Proceedings of the IEEE conference on computer vision and pattern recognition* (2019).

[237] Kuan Wang et al. “Haq: Hardware-aware automated quantization with mixed precision”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 8612–8620.

[238] Linnan Wang et al. “Sample-efficient neural architecture search by learning action space”. In: *arXiv preprint arXiv:1906.06832* (2019).

[239] Peiqi Wang et al. “HitNet: hybrid ternary recurrent neural network”. In: *Proceedings of the NIPS*. 2018, pp. 604–614.

[240] Peisong Wang et al. "Two-step quantization for low-bit neural networks". In: *Proceedings of the IEEE Conference on computer vision and pattern recognition*. 2018, pp. 4376–4384.

[241] Tianzhe Wang et al. "Apq: Joint search for network architecture, pruning and quantization policy". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 2078–2087.

[242] Xinxin Wang et al. "A novel rram-based adaptive-threshold lif neuron circuit for high recognition accuracy". In: *2018 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*. IEEE. 2018, pp. 1–2.

[243] Ying Wang, Yadong Lu, and Tijmen Blankevoort. "Differentiable joint pruning and quantization for hardware efficiency". In: *European Conference on Computer Vision*. Springer. 2020, pp. 259–277.

[244] Zhehui Wang et al. "EDCompress: Energy-Aware Model Compression for Dataflows". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[245] Wei Wen et al. "Neural predictor for neural architecture search". In: *European Conference on Computer Vision*. Springer. 2020, pp. 660–676.

[246] Ross Wightman. *PyTorch Image Models*. https://github.com/rwightman/pytorch-image-models. 2019. DOI: 10.5281/zenodo.4414861.

[247] Yasitha M Wijesinghe, Jayathu G Samarawickrama, and Dileeka Dias. "Hardware and Software Co-Design for Object Detection with Modified ViBe Algorithm and Particle Filtering Based Object Tracking". In: *2019 14th Conference on Industrial and Information Systems (ICIIS)*. IEEE. 2019, pp. 506–511.

[248] Adina Williams, Nikita Nangia, and Samuel R Bowman. "A broad-coverage challenge corpus for sentence understanding through inference". In: *arXiv preprint arXiv:1704.05426* (2017).

[249] H-S Philip Wong et al. "Metal–oxide RRAM". In: *Proceedings of the IEEE* 100.6 (2012), pp. 1951–1970.

[250] Bichen Wu et al. "FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 10734–10742.

[251] Bichen Wu et al. "Mixed Precision Quantization of ConvNets via Differentiable Neural Architecture Search". In: *arXiv preprint arXiv:1812.00090* (2018).

[252] Bichen Wu et al. "Shift: A zero flop, zero parameter alternative to spatial convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9127–9135.

[253] Hao Wu et al. "Integer quantization for deep learning inference: Principles and empirical evaluation". In: *arXiv preprint arXiv:2004.09602* (2020).

[254] Jiaxiang Wu et al. “Quantized convolutional neural networks for mobile devices”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2016, pp. 4820–4828.

[255] Chen Xu et al. “Alternating multi-bit quantization for recurrent neural networks”. In: *arXiv preprint arXiv:1802.00150* (2018).

[256] Ke Xu, Xiaoyun Wang, and Dong Wang. “A Scalable OpenCL-Based FPGA Accelerator for YOLOv2”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE. 2019, pp. 317–317.

[257] Xiaowei Xu et al. “Dac-sdc low power object detection challenge for uav applications”. In: *IEEE Transactions on pattern analysis and machine intelligence (TPAMI)* (2019).

[258] Yuhui Xu et al. “Pc-darts: Partial channel connections for memory-efficient differentiable architecture search”. In: *arXiv preprint arXiv:1907.05737* (2019).

[259] Huanrui Yang et al. “BSQ: Exploring Bit-Level Sparsity for Mixed-Precision Neural Network Quantization”. In: *arXiv preprint arXiv:2102.10462* (2021).

[260] Jiwei Yang et al. “Quantization networks”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2019, pp. 7308–7316.

[261] Lei Yang et al. “Co-exploring neural architecture and network-on-chip design for real-time artificial intelligence”. In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC).* IEEE. 2020, pp. 85–90.

[262] Tien-Ju Yang et al. “Netadapt: Platform-aware neural network adaptation for mobile applications”. In: *Proceedings of the European Conference on Computer Vision (ECCV).* 2018, pp. 285–300.

[263] Yang Yang et al. “FPNet: Customized Convolutional Neural Network for FPGA Platforms”. In: *2019 International Conference on Field-Programmable Technology (ICFPT).* IEEE. 2019, pp. 399–402.

[264] Yifan Yang et al. “Synetgy: Algorithm-hardware co-design for ConvNet accelerators on embedded FPGAs”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM. 2019, pp. 23–32.

[265] Zhaohui Yang et al. “Searching for low-bit weights in quantized neural networks”. In: *Advances in neural information processing systems* (2020).

[266] Zhilin Yang et al. “XLNet: Generalized autoregressive pretraining for language understanding”. In: *Advances in neural information processing systems.* 2019, pp. 5753–5763.

[267] Zhewei Yao et al. “Hawq-v3: Dyadic neural network quantization”. In: *International Conference on Machine Learning.* PMLR. 2021, pp. 11875–11886.

[268] Zhewei Yao et al. “HAWQV3: Dyadic Neural Network Quantization”. In: *et preprint arXiv:2011.10680* (2020).

[269] Zhewei Yao et al. “Hessian-based Analysis of Large Batch Training and Robustness to Adversaries”. In: *Advances in Neural Information Processing Systems* (2018).

[270] Zhewei Yao et al. “Large batch size training of neural networks with adversarial training and second-order information”. In: *arXiv preprint arXiv:1810.01021* (2018).

[271] Penghang Yin et al. “Understanding straight-through estimator in training activation quantized neural nets”. In: *arXiv preprint arXiv:1903.05662* (2019).

[272] Fang Yu et al. “HFP: Hardware-Aware Filter Pruning for Deep Convolutional Neural Networks Acceleration”. In: *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE. 2021, pp. 255–262.

[273] Fisher Yu et al. “Deep layer aggregation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

[274] Haibao Yu et al. “Search what you want: Barrier panelty NAS for mixed precision quantization”. In: *European Conference on Computer Vision*. Springer. 2020, pp. 1–16.

[275] Shixing Yu et al. “Hessian-aware pruning and optimal neural implant”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2022, pp. 3880–3891.

[276] Li Yuan et al. “Tokens-to-token vit: Training vision transformers from scratch on imagenet”. In: *arXiv preprint arXiv:2101.11986* (2021).

[277] Dongqing Zhang et al. “Lq-nets: Learned quantization for highly accurate and compact deep neural networks”. In: *European conference on computer vision (ECCV)*. 2018.

[278] Hang Zhang et al. “Resnest: Split-attention networks”. In: *arXiv preprint arXiv:2004.08955* (2020).

[279] Xiaofan Zhang et al. “DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[280] Xiaofan Zhang et al. “SkyNet: A Champion Model for DAC-SDC on Low Power Object Detection”. In: *arXiv preprint arXiv:1906.10327* (2019).

[281] Xinyi Zhang et al. “When neural architecture search meets hardware implementation: from hardware awareness to co-design”. In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2019, pp. 25–30.

[282] Lingran Zhao, Zhen Dong, and Kurt Keutzer. “Analysis of Quantization on MLP-based Vision Models”. In: *arXiv preprint arXiv:2209.06383* (2022).

[283] Qibin Zhao et al. “Learning efficient tensor representations with ring-structured networks”. In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2019, pp. 8608–8612.

[284] Ritchie Zhao et al. “Improving neural network quantization without retraining using outlier channel splitting”. In: *Proceedings of Machine Learning Research* (2019).

[285] Sijie Zhao, Tao Yue, and Xuemei Hu. “Distribution-Aware Adaptive Multi-Bit Quantization”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 9281–9290.

[286] Aojun Zhou et al. “Explicit loss-error-aware quantization for low-bit deep neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 9426–9435.

[287] Aojun Zhou et al. “Incremental network quantization: Towards lossless cnns with low-precision weights”. In: *arXiv preprint arXiv:1702.03044* (2017).

[288] Shuchang Zhou et al. “Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients”. In: *arXiv preprint arXiv:1606.06160* (2016).

[289] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. “Objects as points”. In: *arXiv preprint arXiv:1904.07850* (2019).

[290] Yiren Zhou et al. “Adaptive quantization for deep neural network”. In: *arXiv preprint arXiv:1712.01048* (2017).

[291] Zheng Zhou et al. “The characteristics of binary spike-time-dependent plasticity in HfO 2-based RRAM and applications for pattern recognition”. In: *Nanoscale Research Letters* 12.1 (2017), pp. 1–5.

[292] Xizhou Zhu et al. “Deformable convnets v2: More deformable, better results”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[293] Yuhao Zhu et al. “Euphrates: algorithm-SoC co-design for low-power mobile continuous vision”. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 547–560.

[294] Bohan Zhuang et al. “Towards effective low-bitwidth convolutional neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7920–7928.

[295] Barret Zoph and Quoc V Le. “Neural architecture search with reinforcement learning”. In: *arXiv preprint arXiv:1611.01578* (2016).