Diss. ETH No. 28528

Enabling Deep Learning on Edge Devices

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by
ZHONGNAN QU
M.Sc. TU Munich

born on 05.05.1992
citizen of
China
Henan

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Olga Saukh, co-examiner

2022

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 000

Zhongnan Qu

Enabling Deep Learning on Edge Devices

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH NO. 28528

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Olga Saukh, co-examiner
Examination date: July 26, 2022

To my family.
致我的家人。

Abstract

Deep neural networks (DNNs) have succeeded in many different perception tasks, e.g., computer vision, natural language processing, reinforcement learning, etc. The high-performed DNNs heavily rely on intensive resource consumption. For example, training a DNN requires high dynamic memory, a large-scale dataset, and a large number of computations (a long training time); even inference with a DNN also demands a large amount of static storage, computations (a long inference time), and energy. Therefore, state-of-the-art DNNs are often deployed on a cloud server with a large number of super-computers, a high-bandwidth communication bus, a shared storage infrastructure, and a high power supplement.

Recently, some new emerging intelligent applications, e.g., AR/VR, mobile assistants, Internet of Things, require us to deploy DNNs on resource-constrained edge devices. Compare to a cloud server, edge devices often have a rather small amount of resources. To deploy DNNs on edge devices, we need to reduce the size of DNNs, i.e., we target a better trade-off between the resource consumption and the model accuracy.

In this thesis, we study four edge intelligent scenarios and develop different methodologies to enable deep learning in each scenario. Since current DNNs are often over-parameterized, our goal is to find and to reduce the redundancy of the DNNs in each scenario. We summarize the four studied scenarios as follows,

- **Inference on Edge Devices.** Firstly, we enable efficient inference of DNNs given the fixed resource constraints on edge devices. Compared to cloud inference, inference on edge devices avoids transmitting the data to the cloud server, which can achieve a more stable, fast, and energy-efficient inference. Regarding the main resource constraints from storing a large number of weights and computation during inference, we proposed an Adaptive Loss-aware Quantization (ALQ) for multi-bit networks. ALQ reduces the redundancy on the quantization bitwidth. The direct optimization objective (i.e., the loss) and the learned adaptive bitwidth assignment allow ALQ to acquire extremely low-bit networks with an average bitwidth below 1-bit while yielding a higher accuracy than state-of-the-art binary networks.

- **Adaptation on Edge Devices.** Secondly, we enable efficient adaptation of DNNs when the resource constraints on the target

edge devices dynamically change during runtime, e.g., the allowed execution time and the allocatable RAM. To maximize the model accuracy during on-device inference, we develop a new synthesis approach, Dynamic REal-time Sparse Subnets (DRESS) that can sample and execute sub-networks with different resource demands from a backbone network. DRESS reduces the redundancy among multiple sub-networks by weight sharing and architecture sharing, resulting in storage efficiency and re-configuration efficiency, respectively. The generated sub-networks have different sparsity, and thus can be fetched to infer under varying resource constraints by utilizing sparse tensor computations.

- **Learning on Edge Devices.** Thirdly, we enable efficient learning of DNNs when facing unseen environments or users on edge devices. On-device learning requires both data- and memory-efficiency. We thus propose a new meta learning method p-Meta to enable memory-efficient learning with only a few samples of unseen tasks. p-Meta reduces the updating redundancy by identifying and updating structurewise adaptation-critical weights only, which saves the necessary memory consumption for the updated weights.

- **Edge-Server System.** Finally, we enable efficient inference and efficient updating on edge-server systems. In an edge-server system, several resource-constrained edge devices are connected to a resource-sufficient server with a constrained communication bus. Due to the limited relevant training data beforehand, pretrained DNNs may be significantly improved after the initial deployment. On such an edge-server system, on-device inference is preferred over cloud inference, since it can achieve a fast and stable inference with less energy consumption. Yet retraining on the cloud server is preferred over on-device retraining (or federated learning) due to the limited memory and computing power on edge devices. We proposed a novel pipeline Deep Partial Updating (DPU) to iteratively update the deployed inference model. Particularly, when newly collected data samples from edge devices or from other sources are available at the server, the server smartly selects only a subset of critical weights to update and send to each edge device. This weightwise partial updating reduces the redundant updating by reusing the pretrained weights, which achieves a similar accuracy as full updating yet with a significantly lower communication cost.

Zusammenfassung

Deep Neural Networks (DNNs) haben sich bei vielen verschiedenen Wahrnehmungsaufgaben bewährt, z. B. Computer Vision, Verarbeitung natürlicher Sprache, Verstärkungslernen usw. Die leistungsstarken DNNs sind stark auf einen intensiven Ressourcenverbrauch angewiesen. Beispielsweise erfordert das Training eines DNN einen hohen dynamischen Speicher, einen großen Datensatz und eine große Anzahl von Berechnungen (eine lange Trainingszeit); Selbst die Inferenz mit einem DNN erfordert auch eine große Menge an statischem Speicher, Berechnungen (eine lange Inferenzzeit) und Energie. Daher werden moderne DNNs häufig auf einem Cloud-Server mit einer großen Anzahl von Supercomputern, einem Kommunikationsbus mit hoher Bandbreite, einer gemeinsam genutzten Speicherinfrastruktur und einem Hochleistungszusatz eingesetzt.

In letzter Zeit erfordern einige neu entstehende intelligente Anwendungen, z. B. AR/VR, mobile Assistenten, Internet of Things, den Einsatz von DNNs auf ressourcenbeschränkten Edge-Geräten. Im Vergleich zu einem Cloud-Server verfügen Edge-Geräte oft über eine eher geringe Menge an Ressourcen. Um DNNs auf Edge-Geräten einzusetzen, müssen wir die Größe von DNNs reduzieren, d. h. wir streben einen besseren Kompromiss zwischen dem Ressourcenverbrauch und der Modellgenauigkeit an.

In dieser Doktorarbeit untersuchen wir vier intelligente Edge-Szenarien und entwickeln verschiedene Methoden, um Deep Learning in jedem Szenario zu ermöglichen. Da aktuelle DNNs oft überparametrisiert sind, ist unser Ziel, die Redundanz der DNNs in jedem Szenario zu finden und zu reduzieren. Wir fassen die vier untersuchten Szenarien wie folgt zusammen,

- **Inferenz auf Edge-Geräten.** Erstens ermöglichen wir eine effiziente Inferenz von DNNs angesichts der festen Ressourcenbeschränkungen auf Edge-Geräten. Im Vergleich zur Cloud-Inferenz wird bei der Inferenz auf Edge-Geräten die Übertragung der Daten an den Cloud-Server vermieden, wodurch eine stabilere, schnellere und energieeffizientere Inferenz erreicht werden kann. In Bezug auf die wichtigsten Ressourcenbeschränkungen, die sich aus der Speicherung einer großen Anzahl von Gewichten und Berechnungen während der Inferenz ergeben, haben wir eine Adaptive Loss-aware Quantization (ALQ) für Multibit-Netzwerke vorgeschlagen.

ALQ reduziert die Redundanz in der Quantisierungsbitbreite. Das direkte Optimierungsziel (d. h. der Verlust) und die erlernte adaptive Bitbreitenzuweisung ermöglichen es ALQ, Netze mit extrem niedrigen Bits mit einer durchschnittlichen Bitbreite unter 1-Bit zu erfassen und gleichzeitig eine höhere Genauigkeit als moderne binäre Netze zu erzielen.

- **Anpassung auf Edge-Geräten.** Zweitens ermöglichen wir eine effiziente Anpassung von DNNs, wenn sich die Ressourcenbeschränkungen auf den Zielgeräten während der Laufzeit dynamisch ändern, z. B. die erlaubte Ausführungszeit und der zuweisbare RAM. Um die Modellgenauigkeit während der Inferenz auf dem Gerät zu maximieren, entwickeln wir einen neuen Syntheseansatz, Dynamic REal-time Sparse Subnets (DRESS), der Subnetze mit unterschiedlichen Ressourcenanforderungen von einem Backbone-Netz abtasten und ausführen kann. DRESS reduziert die Redundanz in mehreren Subnetzen durch gemeinsame Nutzung von Gewicht und Architektur, was zu Speichereffizienz bzw. Rekonfigurationseffizienz führt. Die erzeugten Subnetze weisen unterschiedliche Sparsamkeit auf und können daher abgerufen werden, um unter variierenden Ressourcenbeschränkungen durch Verwendung von spärliche Tensorberechnungen zu folgern.

- **Lernen auf Edge-Geräten.** Drittens ermöglichen wir ein effizientes Lernen von DNNs, wenn Sie mit unsichtbaren Umgebungen oder Benutzern auf Edge-Geräten konfrontiert sind. Lernen auf dem Edge-Gerät erfordert sowohl Dateneffizienz als auch Speichereffizienz. Wir schlagen daher eine neue Meta-Lernmethode p-Meta vor, die speichereffizientes Lernen mit nur wenigen Datenbeispielen von unbekannten Aufgaben ermöglicht. p-Meta reduziert die Aktualisierungsredundanz, indem es nur strukturweise anpassungskritischen Gewichte identifiziert und aktualisiert, wodurch der notwendige Speicherverbrauch für die aktualisierten Gewichte eingespart wird.

- **Edge-Server-System.** Schließlich ermöglichen wir effiziente Inferenz und effiziente Aktualisierung auf Edge-Server-Systemen. In einem Edge-Server-System sind mehrere ressourcenbeschränkte Edge-Geräte mit einem ressourcenstarken Server mit einem eingeschränkten Kommunikationsbus verbunden. Aufgrund der begrenzten Anzahl relevanter Trainingsdaten im Voraus können vortrainierte DNNs nach dem anfänglichen Einsatz erheblich verbessert werden. In einem solchen Edge-Server-System wird die Inferenz auf dem Gerät der Inferenz in der Cloud vorgezogen, da sie eine schnelle und stabile Inferenz mit weniger Energieverbrauch

erreichen kann. Aufgrund des begrenzten Speichers und der begrenzten Rechenleistung auf Edge-Geräten wird jedoch die Re-Training in der Cloud gegenüber der Re-Training auf dem Gerät (oder föderiertem Lernen) bevorzugt. Wir haben eine neuartige Pipeline, Deep Partial Updating (DPU) vorgeschlagen, um das eingesetzte Inferenzmodell iterativ zu aktualisieren. Insbesondere, wenn neu gesammelte Datenbeispielen von Edge-Geräten oder aus anderen Quellen auf dem Cloud-Server verfügbar sind, wählt der Server intelligenterweise nur eine Teilmenge kritischer Gewichte aus, um sie zu aktualisieren und an jedes Edge-Gerät zu senden. Diese gewichtsmäßige Teilaktualisierung reduziert die redundante Aktualisierung durch Wiederverwendung der vortrainierten Gewichtungen, wodurch eine ähnliche Genauigkeit wie bei der vollständigen Aktualisierung erreicht wird, jedoch mit deutlich geringeren Kommunikationskosten.

Acknowledgements

Contents

Abstract i

Zusammenfassung iii

Acknowledgements vii

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 High Resource Demands of DNNs 2
- 1.2 Cloud Intelligence 3
- 1.3 Edge Intelligence 4
- 1.4 Thesis Outline 6

2 Inference on Edge Devices 11
- 2.1 Introduction 12
- 2.2 Related Work 13
- 2.3 Preliminaries and Notations 14
- 2.4 Adaptive Loss-Aware Quantization 15
- 2.5 Activation Quantization 25
- 2.6 Experiments 26
- 2.7 Summary 39

3 Adaptation on Edge Devices 41
- 3.1 Introduction 42
- 3.2 Related Work 43
- 3.3 Dynamic Real-time Sparse Subnets 45
- 3.4 Evaluation 52
- 3.5 Deployments 65
- 3.6 Summary 66

4 Learning on Edge Devices 71
- 4.1 Introduction 72
- 4.2 Related Work 74
- 4.3 Preliminaries and Challenges 75
- 4.4 p-Meta 78
- 4.5 Theoretical Analysis on Memory and Computation 86

4.6 Experiments 89
4.7 Summary 98

5 Edge-Server System **101**
5.1 Introduction 102
5.2 Related Work 104
5.3 Notations and Settings 106
5.4 Deep Partial Updating 107
5.5 Evaluation 113
5.6 Summary 130

6 Conclusion **135**
6.1 Contributions 136
6.2 Potential Future Directions 138

Bibliography **143**

List of Publications **159**

Curriculum Vitæ **161**

List of Figures

1.1 The number of parameters in different DNNs is exponentially increased along the years. 3
1.2 Example edge intelligence applications. 4
1.3 Comparison between deep learning on the cloud server and deep learning on edge devices. 5

2.1 The overall approach of ALQ. 16
2.2 The curves of the average reconstruction error in different group size 29
2.3 Validation accuracy trained with ALQ and other STE-based baselines. 32
2.4 Distribution of the average bitwidth and the number of weights across layers. 33
2.5 The training loss curves of different steps in ALQ. 37

3.1 The computation graph of DRESS. 47
3.2 The cosine similarity between the loss gradients of different subnets. 47
3.3 Traditional CSR format of unstructured sparse tensor. . . . 50
3.4 DRESS CSR format of row-based unstructured sparse tensor. 51
3.5 Ablation studies on different row sizes. 55
3.6 The BN statistics of different subnets across layers. 56
3.7 Comparison between DRESS with sampling and DRESS without sampling. 57
3.8 Left: Comparing parallel training with iterative training. Right: Ablation studies on the correction factor γ. 60
3.9 Comparing DRESS with other baselines on image classification. 61
3.10 Comparing DRESS with other baselines on object detection and instance segmentation. 64
3.11 Comparing DRESS with traditional pruning on ResNet20 in terms of the layerwise sparsity. 68
3.12 Comparing DRESS with traditional pruning on MobileNetV2 in terms of the layerwise sparsity. 69

4.1 Meta learning and few-shot learning in the context of on-device learning. 72
4.2 A typical layer in DNNs. 78
4.3 Meta attention during meta-training. 82
4.4 Meta attention during on-device few-shot learning. 85
4.5 Layer-wise updating ratios in each updating step. 96
4.6 Cosine similarity of $\gamma_{1:L}$ between random pair of data samples. 96

5.1 The iterative process of edge-to-server communication and server-to-edge communication. 103
5.2 The overall approach of DPU. 107
5.3 Comparing full updating methods with different initialization methods at each round. 117
5.4 Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds. 119
5.5 Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds. 120
5.6 Comparison w.r.t. the mean accuracy difference (full updating as the reference) under different λ. 121
5.7 Number of updated weights across all layers when adopting different rewinding metrics with $k = 0.01$. 123
5.8 Number of updated weights across all layers when adopting different rewinding metrics with $k = 0.05$. 124
5.9 Number of updated weights across all layers when adopting different rewinding metrics with $k = 0.1$. 125
5.10 The test accuracy of partial updating methods with different rewinding metrics. 126
5.11 DPU is compared with other baselines on different benchmarks in terms of the test accuracy during multi-round updating. 126
5.12 The ratio, between the total communication cost under DPU and that under full updating, varies with the number of nodes N. 127
5.13 Comparison w.r.t. the mean accuracy difference (full updating as the reference) under different settings. 132
5.14 Comparison w.r.t. the mean accuracy under different settings. 133
5.15 Comparison w.r.t. the standard deviation of accuracy under different settings. 134

List of Tables

2.1 Comparison between uniform bitwidth and adaptive bitwidth in ALQ. 32
2.2 Comparison with unstructured pruning methods (LeNet5 on MNIST) 35
2.3 Comparison with binary networks (VGGNet on CIFAR10) 36
2.4 Comparison with quantized networks (ResNet18/34 on ImageNet). 38

3.1 The average test accuracy over all sub-networks, the theoretical storage (MB) required by all sub-networks and the average number of theoretical MFLOPs over all sub-networks. 62
3.2 The average inference time (ms) on RaspberryPi 4. 65

4.1 Summary of major notations. 76
4.2 Memory and total computation of inference and training in example few-shot learning. 77
4.3 5-Way 1-shot few-shot image classification results on 4Conv and ResNet12. 92
4.4 5-Way 5-shot few-shot image classification results on 4Conv and ResNet12. 92
4.5 Few-shot reinforcement learning results on 2D navigation and robot locomotion tasks. 93
4.6 Ablation results of meta attention on 4Conv. 95
4.7 Ablation results of sparse $\boldsymbol{x}_{l-1}$ and sparse $\boldsymbol{g}(\boldsymbol{y}_l)$. 97
4.8 Comparison between different pooling and normalization layers. 97
4.9 Ablation results of sample batch sizes. 98

5.1 Comparing training loss after rewinding and the final test accuracy under different metrics. 121
5.2 The average accuracy difference (full updating as the reference) over all rounds and the ratio of communication cost over all rounds related to full updating. 127
5.3 The test accuracy of single-round updating on different initial deployed models. 130

6.1 Static memory of the model and the training samples in example self-supervised learning. 140

1

Introduction

Deep learning is a new disruptive technology that extremely drives the development of artificial intelligence. Deep neural networks (DNNs) are widely used in deep learning, which can make predictions according to the given inputs. A DNN consists of a large number of cascaded layers, where each layer often comprises (*i*) trainable weights that can perform matrix multiplication on the layer's input to output extracted features, (*ii*) a non-linear function that can bring non-linear behaviors. DNNs can often achieve superior performance than prior computational models or even human beings in many areas, e.g., computer vision, natural language processing, mathematics, biochemistry, etc.

In image classification, AlexNet [KSH12] automatically learns the features by training a deep convolutional neural network with GPUs, and the competition results on ImageNet Large Scale Visual Recognition Challenge (ILSVRC) show that AlexNet surpasses the prior classifiers that are built based on hand-crafted features e.g., random forest and support vector machine, by a large margin (over 10% accuracy gain). AlphaGo Zero [SHM+16] reinforce-learns a deep policy model to predict the movement on the Go board via playing games against itself, and the learned model can even defeat a human world champion of Go games. BERT [DCLT19] pretrains deep bidirectional representations from unlabeled text and then fine-tunes the pretrained model, which exhibits a better performance in language understanding on SQuAD test than humans. Recently, graph convolutional neural networks [EAGT19] have also been applied to many biological and chemical problems e.g., predicting protein function, predicting binarized gene expression, etc. As a result, DNNs not only can conduct some intelligent tasks that previously must rely on cumbersome human efforts in our daily life, but also may bring new scientific inspirations that are less explored in the long-term human history.

1.1 High Resource Demands of DNNs

The high performance of state-of-the-art DNNs benefits from the intensive resource consumption during both the *training* phase and the *inference* phase.

During the training phase, current DNNs are often optimized on high-performance cloud servers with a large-scale dataset over a long time, which may take (*i*) many human labor resources to prepare the dataset or the training implementation, (*ii*) a large amount of time and money cost, (*iii*) a remarkable CO2 emission [CGW+20]. For example, the widely-used DNN ResNet50 [HZRS16] needs to be trained with ImageNet dataset which contains 1.2 million well-labeled internet images collected from 1000 balanced fine classes; the GPT-3 model published by OpenAI [BMR+20] takes 3.14×10^{23} floating-point multiply-accumulate operations (FLOPs) for a single training run, which equivalent to 355 GPU-years and 4.6M US dollars, according to the theoretical 2.8×10^{13} FLOPs of high-performed Nvidia V100 GPU and the lowest 3-year reserved cloud pricing we could find [Ope20].

Even during the inference phase, the pretrained DNNs still demand a rather significant amount of computing resources from e.g., memory, computation, latency, and energy. For example, the Faster-RCNN model [RHGS15] requires several hundreds of GFLOPs for a single inference, thus can only achieve around 5 frames per second for object detection on a state-of-the-art GPU; the current language models [BMR+20] contain billions of parameters which often require several GPUs with GB-level memory on a cloud server with high-bandwidth communication bus for the parallelism during inference. Note that the state-of-the-art GPU often has a minimal operation power requirement of around 500W [Nvi22]. All the examples mentioned above indicate the inherent resource-intensive characteristics of DNNs.

However, the resource demands of DNNs still keep growing. As noted in [BSH+21], although the state-of-the-art DNNs continuously improve the accuracy level, the number of parameters (as well as the number of FLOPs) in these DNNs also increases along the years, even with an *exponential* increasing rate, as shown in Figure 1.1. On the other hand, the research and development of hardware often require a long cycle and a high investment. As a result, the growth rate of the model size is far larger than the growth rate of the computing power of the state-of-the-art high-performance computers, e.g., GPUs. For example, the number of parameters has increased more than 2000 times from AlexNet in 2012 [KSH12] to GPT-3 in 2020 [BMR+20], whereas at the same time, the memory of Nvidia GPU has only increased 22 times from Geforce GTX 660 to Geforce RTX 3090, and the computing power (FLOPs/second) has increased around 17 times [Wik22e].

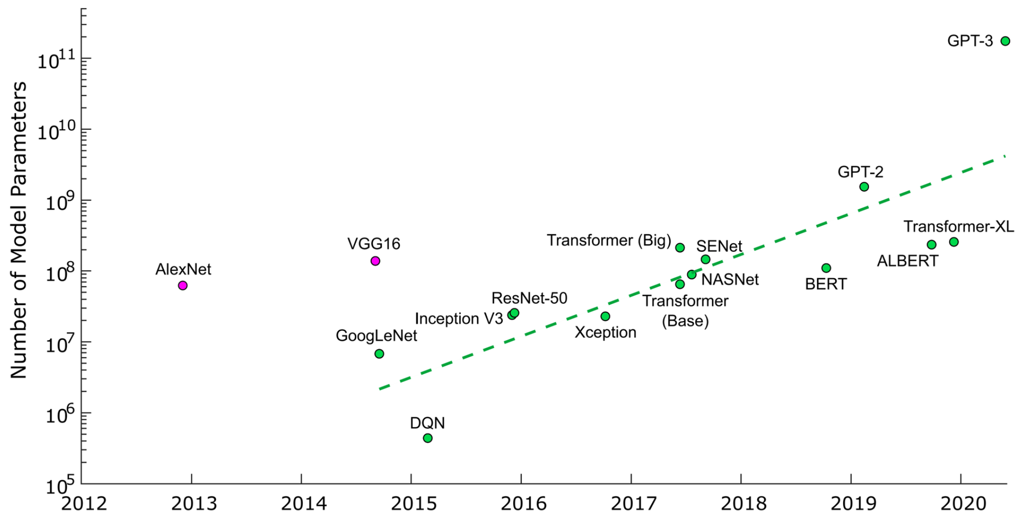

Figure 1.1: The number of parameters in different DNNs is exponentially increased along the years. Note that the two outlying nodes (pink) are AlexNet and VGG16, now considered over-parameterized. The figure is originally from [BSH+21].

1.2 Cloud Intelligence

As mentioned above, there exists a large gap between the computing power of available hardware and the resource demands of DNNs. The common solution to such a conflict is to gather multiple high-performance computers and build a cluster-based server in the cloud, also known as cloud computing [Wik22b]. A cloud server is a group of two or more computers that can share the computing resource, communicate with others and distribute the workload of the same task according to the predefined scheduling system [Cap20]. Some commercial cloud servers include Amazon Web Services (AWS), Google Cloud, Microsoft Azure, etc. These cloud servers may contain high-performance computers of CPUs, GPUs, TPUs, the communication bus with a high bandwidth, the on-demand shared storage infrastructures, and the high power supplement.

Particularly, a DNN can be deployed on a cloud server to perform some resource-intensive intelligent applications e.g., gradient-based training, machine translation, question answering systems, etc. The high resource demands from these applications can be delegated to multiple computers, and if necessary the results from these computers are aggregated afterwards. Cloud intelligence has become a prevailing solution for many intelligent services, which require a large amount of resources (e.g., memory, computation) whereas a single computer is often not unable to meet these requirements.

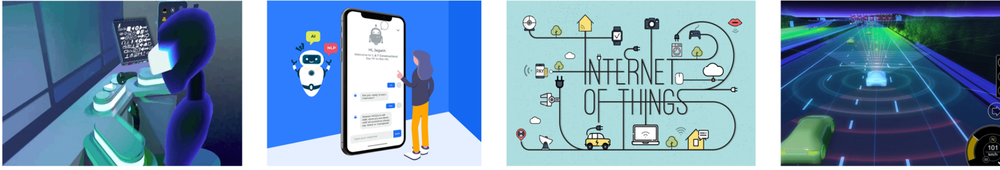

Figure 1.2: Example edge intelligence applications. From Left to Right: Augmented/Virtual Reality, Mobile Assistants, Internet of Things, Autonomous Driving. The images are from Google.

1.3 Edge Intelligence

In addition to cloud intelligence, some new emerging edge intelligent applications further require us to deploy DNNs on *edge devices*. The term edge refers to an entry point [Wik22c]. Accordingly, the collected data (at the entry point) are processed by DNNs locally, i.e., on devices. Edge devices have a large variety, including mobile phones, wearable devices, sensor nodes, etc. Some example edge intelligent applications (see in Figure 1.2) include but are not limited to,

- **Augmented/Virtual Reality.** Augmented/Virtual reality (AR/VR) can visualize the digital information as the real world via wearable devices, e.g., glasses [Wik22a, Wik22h]. To bridge the gap between the physical world and the virtual environment, many AR/VR tasks, e.g., hand detection, eye tracking, digital humans, require deep learning methods to provide high-quality interaction.

- **Mobile Assistants.** Mobile assistants are software agents that can perform tasks or services on mobile platforms for an individual based on commands or questions [Wik22g]. Individual users can input voice, images, or text to mobile assistants. Given the inputs from users, DNNs are utilized to recognize, understand, and communicate with users.

- **Internet of Things.** Internet of Things (IoT) describes physical objects with sensors, processing ability, software, and other technologies that connect with other devices over communication networks [Wik22d]. IoT applications use DNNs for automatic sensing and reasoning, e.g., detecting intruders in a "smart home" monitor system.

- **Autonomous Driving.** Autonomous cars can sense their surroundings and move safely with little or no human inputs [Wik22f]. Thanks to the rapid development of deep learning, many DNNs in computer vision tasks, e.g., object detection, 3D localization,

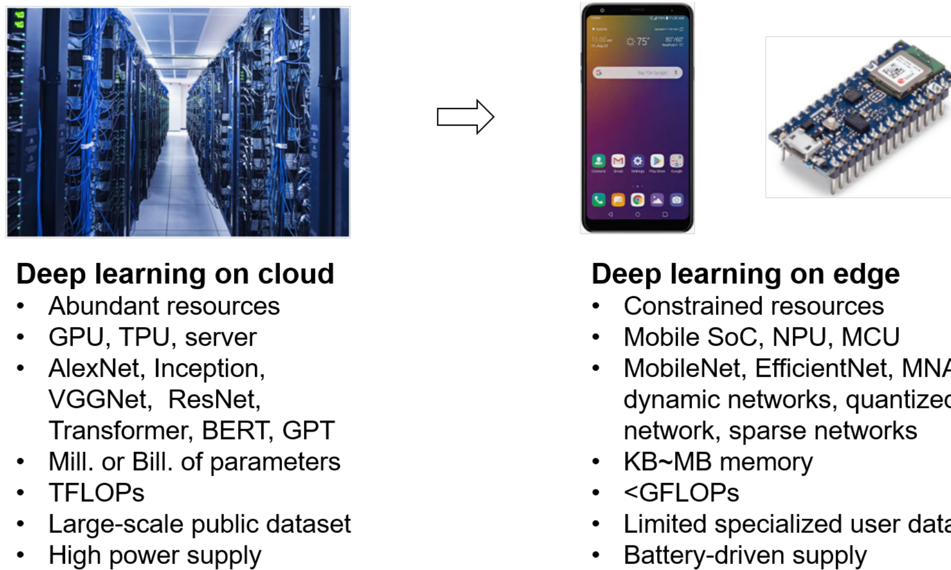

Figure 1.3: Comparison between deep learning on cloud and deep learning on edge. The figure is originally from [Sor21].

semantic segmentation, have been widely adopted to interpret sensory information and identify appropriate navigation paths.

In comparison to cloud intelligent applications, edge intelligent applications have the following advantages, (*i*) it does not encounter privacy issues and can be used on sensitive/confidential data, as the data are processed locally; (*ii*) it reduces the reliance on the cloud server, and can achieve a stable inference even with congested/interrupted communication channels; (*iii*) it can realize a real-time inference if the communication bandwidth is limited; (*iv*) it can save energy by avoiding to transfer data to the cloud server which often costs significant amounts of energy than sensing and computation [PW19, Guo18, LCI+19].

Unfortunately, deploying DNNs on edge devices is not trivial, as current DNNs contradict the *resource-constrained* nature of edge devices. Unlike plenty of high-performance computers (e.g., GPUs and TPUs) in the cloud server, the processors on edge devices are commonly mobile SoCs, NPUs, or even MCUs, which have a rather small amount of resources and limited scalability. We compare the difference between deep learning on the cloud server and deep learning on edge devices in Figure 1.3. The edge devices often use battery-driven energy and have only several *KB* to *MB* allocatable RAM. Their parallel computing capabilities are also relatively low due to the small number of computing cores. In addition, the number of user data collected on edge devices is also limited in comparison to the large-scale datasets used in cloud training. To deploy DNNs on these edge devices, the complexity of DNNs needs to be trimmed down to fit the limited resource budget.

1.4 Thesis Outline

In this thesis, we will study how to *enable deep learning on edge devices in different scenarios*. Deploying DNNs on edge devices always targets a trade-off between *the resource demands* and *the model accuracy*. Since DNNs often consume a large amount of resources, we hypothesize that there exists redundancy in the DNNs. Our goal is to identify and reduce the redundancy according to the main resource constraints in different scenarios. This thesis is partitioned into four separate scenarios. In each scenario, we will (*i*) analyze its main resource constraints, (*ii*) review the drawbacks in the currently available solutions, (*iii*) propose our solution to reduce the redundancy in the DNNs; (*iv*) verify the effectiveness of our solution experimentally or theoretically. The four studied scenarios are summarized as follows.

1.4.1 Inference on Edge Devices (Chapter 2)

Scenario. We first enable an efficient inference on edge devices. Inference on edge devices does not rely on the connection to the cloud server, thus it is especially preferred if the communication is highly constrained, or a stable and fast inference is required. The main resource constraints of inference on edge devices are the limited static storage and the limited computational ability, as DNNs often contain a large number of parameters to be stored and require a large number of FLOPs for inference. In this scenario, according to the given resource constraints on edge devices, we train a compressed DNN on a cloud server with a large-scale dataset collected beforehand. The well-trained compressed network is then deployed on the edge devices and is able to conduct inference with limited resources.

Related Work. To reduce the storage cost and the computation cost, plenty of works propose to (*i*) design efficient network architectures manually [HZC+17, SHZ+18] or automatically using neural architecture search methods [CGW+20, YH19a, YJL+20]; (*ii*) quantize weights into lower bitwidth to use cheaper operations and reduce the storage consumption [CBD15, RORF16, ZYYH18]; (*iii*) structured [LMZ+19, LWS+20]/unstructured [HMD16, RFC20, EGM+21] pruning unimportant weights as zeros to reduce the number of operations and the number of nonzero weights. We focus on quantizing a pretrained DNN into multi-bit form among others for the following reasons, (*i*) it utilizes the cheaper operations of bitwise `xnor` and `popcount` to replace expensive FLOPs; (*ii*) it achieves a high compression ratio without introducing irregular computations; (*iii*) it explores the lower bound of quantized networks. The state-of-the-art multi-bit networks [GLYB14, GYZC17, HWC18, LZP17, XYL+18, ZYYH18] first assign an empirical global bitwidth across layers and then are optimized by minimizing the reconstruction error to

the full precision weights, which often results in a subpar performance.

Our Solution. To resolve the above drawbacks, we propose an adaptive loss-aware trained quantizer for multi-bit quantization, that (*i*) allocates an adaptive bitwidth to different weights w.r.t. the loss, (*ii*) optimizes the multi-bit quantizer by directly minimizing the loss. We aim at reducing the *redundant quantization bitwidth* of the weights that are less critical to the loss, to achieve a better trade-off between the model accuracy and the resource demands.

1.4.2 Adaptation on Edge Devices (Chapter 3)

Scenario. The compressed DNNs trained with the methods in Chapter 2 can achieve an efficient inference, if the available resources on edge devices are fixed and provided before training on the cloud server. However, the resource constraints on the target edge devices may dynamically change during runtime e.g., the allowed execution time, the allocatable RAM, and the battery energy. To maximize the model accuracy during on-device inference, the deployed DNN should maintain a dynamic capacity, such that the DNN can be adapted and executed under varying resource constraints. In order to quantify the varying resource constraints mentioned earlier, we choose two proxies, (*i*) the storage of weights, which affects the amount of memory fetching and static memory consumption, and (*ii*) the number of operations for inference, which is relevant to the computing energy and the inference latency.

Related Work. The most straightforward solution could be for example deploying multiple individual compressed DNNs with different resource demands on edge devices, yet it consumes several times more storage than a single DNN. Some prior works [HCL+18, HDHB17, YYX+19, YH19b, CGW+20, LN20] proposed to optimize a backbone network (a.k.a. supernet), such that different candidate sub-networks can be sampled from the backbone network while reaching a similar accuracy level as training them individually. However, these works often sample sub-networks along hand-crafted structured dimensions, e.g., kernel size, width, depth, thus the generated sub-networks have different network architectures. This not only results in a sub-optimal performance but also leads to extra re-configuration overhead for storing multiple compiled network architectures.

Our Solution. We overcome the above disadvantages through sampling sub-networks in a row-based unstructured manner, and propose a novel compressed sparse row (CSR) format to efficiently execute different sub-networks on edge devices. Our solution reduces *the architecture redundancy* by reusing a single compiled network architecture among multiple sparse sub-networks, achieving re-configuration efficiency. In

addition, we also reduce *the weight redundancy* by imposing nonzero weight sharing among sub-networks, achieving storage efficiency.

1.4.3 Learning on Edge Devices (Chapter 4)

Scenario. In Chapter 2 and Chapter 3, we train a compressed DNN on a cloud server with a large number of available data samples, such that this pretrained DNN can be deployed on edge devices to conduct inference under *fixed* and *varying* resource constraints, respectively. However, the pretrained DNN may not achieve satisfactory performance when the inference environments on edge devices have a large variance in comparison to the prior environments used to collect data samples for cloud training. In other words, when facing unseen environments or users on edge devices, it is crucial to adapt the pretrained DNN to deliver consistent performance and customized services. New data samples collected by edge devices are often private and have a large diversity across users/devices. Hence, on-device learning is preferred over uploading the data to cloud server. Compared to the number of data samples used in cloud training, the number of collected data on each edge device is significantly smaller (a.k.a. few-shot) due to the limited labor resources. Furthermore, training a DNN, i.e., optimizing its weights, requires storing all the intermediate values of each layer, which often consumes several orders of magnitude more peak memory than inference. Thus, in this scenario, we target memory-efficient and data-efficient on-device learning.

Related Work. Meta learning is a prevailing solution to few-shot learning [HAMS20], where the meta-trained model can learn an unseen task from a few training samples, i.e., data-efficient learning. However, most meta learning algorithms [AES19, FAL17, VOZK+21] optimize the backbone network for better generalization yet ignore the workload if the meta-trained backbone is deployed on low-resource edge platforms for few-shot learning. Existing memory-efficient training schemes include for example, low-precision training [CBG+20, WCB+18], trading memory with computation [CXZG16, GMD+16]. However, they are mainly designed for high-throughput cloud training on large-scale datasets, which are not suitable for on-device learning with only a few data samples.

Our Solution. We ground our work (i.e., memory-efficient few-shot learning) on gradient-based meta learning methods for their wide applicability in various tasks. To avoid the high dynamic memory cost in few-shot learning, we focus on reducing *the updating redundancy*. In other words, we think not all weights in the learner are equally critical for adaptation. Thus, we propose to meta-train a selection mechanism, which can identify and update adaptation-critical weights only during

few-shot learning. This way, only the relevant subset of the intermediate values needs to be stored, leading to memory efficiency.

1.4.4 Edge-Server-System (Chapter 5)

Scenario. In Chapter 2, Chapter 3 and Chapter 4, we explored enabling deep learning on a single edge platform in three different scenarios. In addition to a single edge device, edge-server system is another commonly used infrastructure for edge intelligent applications. In edge-server system, several edge devices are connected to a remote server, and some information is allowed to be communicated between edge devices and the server. In Chapter 5, we design a new pipeline to enable efficient inference and efficient updating for edge-server system. On such an edge-server system, on-device inference is preferred over cloud inference, since it can achieve a fast and stable inference with less energy consumption. Due to a possible lack of relevant training data at the initial deployment, pretrained DNNs may either fail to perform satisfactorily or be significantly improved after the initial deployment. However, the resources on edge devices are often limited e.g., memory, computing power, and energy; the wireless communication is also constrained, e.g., limited bandwidth. An efficient updating/learning that satisfies the resource constraints mentioned above is needed.

Related Work. Communication-efficient federated learning [LHM+18, KMA+19, LSW+20] studies how to compress multiple gradients (to be communicated to the server) calculated on different sets of non-*i.i.d.* local data, such that the aggregation of these (compressed) gradients could result in a similar convergence performance as centralized training on all data. However, federated learning (as well as other on-device retraining methods) has the following main shortages, (*i*) it conducts resource-intensive gradient calculation on edge devices; (*ii*) the collected data are continuously accumulated on memory-constrained edge devices; (*iii*) it needs to label a large number of samples on edge devices.

Our Solution. We propose a two-stage iterative process for a continuous improvement of the deployed model's accuracy, (*i*) at each round, edge devices collect new data samples and send them to the server, and (*ii*) the server retrains the network using all collected data, and then sends the updates to each edge device. An essential challenge herein is that the transmissions in the server-to-edge stage are highly constrained by the limited communication resource (e.g., bandwidth, energy) in comparison to the edge-to-server stage for the following reasons. (*i*) A batch of samples that can lead to reasonable updates is relatively smaller in size than the DNN model, especially for the low-resource data type used on edge devices; (*ii*) the server may also receive data from other sources, e.g., through data augmentation or new data collection campaigns. We reduce

the communication cost in the server-to-edge stage by distinguishing *the redundant updated weights* given newly collected samples. In our proposed solution, the server only selects and updates a small subset of critical weights that have a large contribution to the loss reduction during the retraining.

In the rest of this thesis, we first present our four scenarios of enabling deep learning on edge devices, i.e., inference on edge devices in Chapter 2, adaptation on edge devices in Chapter 3, learning on edge devices in Chapter 4, edge-server-system in Chapter 5, respectively; finally conclude and discuss the future work in Chapter 6.

2

Inference on Edge Devices

We attempt to enable an efficient inference of DNNs on resource-constrained edge devices in this chapter. Particularly, we focus on quantizing a pretrained DNN to fit the given resource constraints on edge devices while with the minimal accuracy drop.

Main Resource Constraints. State-of-the-art DNNs often contain a large number of floating-point weights and require a significant amount of floating-point multiply-accumulate operations, which are essential for conducting accurate inference. However, edge devices have neither powerful computational ability nor enormous storage. Thus, for inference on edge devices, we consider that the main resource constraints are the *limited static storage* and *the limited computing power*.

Principles. Unlike prior quantized networks that (*i*) often assign an empirical global bitwidth across layers, (*ii*) train the quantizer by minimizing the reconstruction error to the full precision weights, we propose an adaptive loss-aware trained quantizer for multi-bit quantization, that (*i*) allocates an adaptive bitwidth to different weights w.r.t. the loss, (*ii*) optimizes the multi-bit quantizer by minimizing the loss. The adaptive bitwidth assignment and the direct optimization objective allow our methods to find and remove more redundant bitwidth, thus achieving both storage efficiency and computation efficiency.

The contents of this chapter are established mainly based on the paper "Adaptive Loss-aware Quantization for Multi-bit Networks" that is published on IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020.

2.1 Introduction

To take advantage of the various pretrained models for efficient inference on resource-constrained edge devices, it is common to compress the pretrained models via pruning [HMD16], quantization [GLYB14, GYZC17, LZP17, XYL+18, ZYYH18], among others. We focus on *quantization*, especially quantizing both the full precision weights and activations of a deep neural network into binary encodes and the corresponding scaling factors [CBD15, RORF16], which are also interpreted as binary basis vectors and floating-point coordinates in a geometry viewpoint [GYZC17]. Neural networks quantized with binary encodes replace expensive floating-point operations by bitwise operations, which are supported even by microprocessors and often result in small memory footprints [MNCM18]. Since the space spanned by only one-bit binary basis and one coordinate is too sparse to optimize, many researchers suggest a multi-bit network (MBN) [GLYB14, GYZC17, HWC18, LZP17, XYL+18, ZYYH18], which allows to obtain a small size without notable accuracy loss and still leverages bitwise operations. An MBN is usually obtained via quantization-aware training. Recent studies [PTT18] leverage bit-packing and bitwise computations for efficient deploying binary networks on a wide range of general devices, which also provides more flexibility to design multi-bit/binary networks.

Challenges. Most MBN quantization schemes [GLYB14, GYZC17, HWC18, LZP17, XYL+18, ZYYH18] predetermine a global bitwidth, and learn a quantizer to transform the full precision parameters into binary bases and coordinates such that the quantized models do not incur a significant accuracy loss. However, these approaches have the following drawbacks:

- A *global bitwidth* may be sub-optimal. Recent studies on fixed-point quantization [KL18, LTA16] show that the optimal bitwidth varies across layers.
- Previous efforts [LZP17, XYL+18, ZYYH18] retain inference accuracy by minimizing *the weight reconstruction error* rather than the loss function. Such an indirect optimization objective may lead to a notable loss in accuracy. Furthermore, they rely on approximated gradients, e.g., straight-through estimators (STE) to propagate gradients through quantization functions during training.
- Many quantization schemes [RORF16, ZYYH18] keep *the first and last layer in full precision empirically*, because quantizing these layers to low bitwidth tends to dramatically decrease the inference accuracy [WSL+18, MM18]. However, these two full precision layers can be a significant storage overhead compared to other low-bit layers (see Section 2.6.5.3). Also, floating-point operations in

both layers can take up the majority of computation in quantized networks [LRB+19].

We overcome the above challenges and drawbacks via a novel **A**daptive **L**oss-aware **Q**uantization scheme (ALQ). Instead of using a uniform bitwidth, ALQ assigns an adaptive different bitwidth to each group of weights. More importantly, ALQ directly minimizes the loss function w.r.t. the quantized weights, by iteratively learning a quantizer that (*i*) smoothly reduces the number of binary bases (also the quantization bitwidth) and (*ii*) alternatively optimizes the remaining binary bases and the corresponding coordinates.

2.2 Related Work

ALQ follows the trend to quantize the DNNs using discrete bases with lower bitwidth to reduce expensive floating-point operations as well as the static storage consumption. Commonly used bases include fixed-point [ZWN+16], power of two [HCS+17, ZYG+17], and $\{-1, 0, +1\}$ [CBD15, RORF16]. We focus on quantization with binary bases i.e., $\{-1, +1\}$ among others for the following considerations. (*i*) If both weights and activations are quantized with the same binary basis, it is possible to evaluate 32 floating-point multiply-accumulate operations (FLOPs) with only 3 instructions on a 32-bit microprocessor, i.e., bitwise `xnor`, `popcount`, and accumulation. This will significantly speed up the `conv` operations [HCS+17, PTT18]. (*ii*) Multi-bit quantization can be considered as the non-uniform counter-part of fixed-point (integer) quantization. A network quantized to fixed-point requires specialized integer arithmetic units and/or specialized integer storage units with various bitwidth for efficient computing [ADJ+17, KL18], whereas a network quantized with multiple binary bases adopts the same operations mentioned before as binary networks. Therefore, multi-bit networks may also achieve a hardware efficiency than fixed-point network in adaptive bitwidth quantization. Popular networks quantized with binary bases include *Binary Networks* and *Multi-bit Networks*.

2.2.1 Quantization for Binary Networks

BNN [CBD15] is the first network with both binarized weights and activations. It dramatically reduces the memory and computation but often with notable accuracy loss. To resume the accuracy degradation from binarization, XNOR-Net [RORF16] introduces a layerwise full precision scaling factor into BNN. However, XNOR-Net leaves the first and last layers unquantized, which consumes more memory. SYQ [FFBL18] studies the efficiency of different structures during binarization/ternarization. LAB [HYK17] is the first loss-aware

quantization scheme which optimizes the weights by directly minimizing the loss function.

ALQ is inspired by recent loss-aware binary networks such as LAB [HYK17]. Loss-aware quantization has also been extended to fixed-point networks in [HK18]. However, existing loss-aware quantization schemes proposed for binary and ternary networks [HYK17, HK18, ZYWC18] are inapplicable for MBNs. This is because multiple binary bases dramatically extend the optimization space with the same bitwidth (i.e., an optimal set of binary bases rather than a single basis), which may be intractable. Some proposals [HYK17, HK18, ZYWC18] still require full-precision weights and gradient approximation (backward STE and forward loss-aware projection), introducing undesirable errors when minimizing the loss. In contrast, ALQ is free from gradient approximation.

2.2.2 Quantization for Multi-bit Networks

MBNs denote networks that use multiple binary bases to trade-off storage and accuracy. Gong et al. propose a residual quantization process, which greedily searches the next binary basis by minimizing the residual reconstruction error [GLYB14]. Guo et al. improve the greedy search with a least square refinement [GYZC17]. Xu et al. [XYL+18] separate this search into two alternating steps, fixing coordinates then exhausted searching for optimal bases, and fixing the bases then refining the coordinates using the method in [GYZC17]. LQ-Net [ZYYH18] extends the scheme of [XYL+18] with a moving average updating, which jointly quantizes weights and activations. However, similar to XNOR-Net [RORF16], LQ-Net [ZYYH18] does not quantize the first and last layers. ABC-Net [LZP17] leverages the statistical information of all weights to construct the binary bases as a whole for all layers.

All the state-of-the-art MBN quantization schemes minimize the weight reconstruction error rather than the loss function of the network. They also rely on the gradient approximation such as STE when back propagating the quantization function. In addition, they all predetermine a uniform bitwidth for all parameters. The indirect objective, the approximated gradient, and the global bitwidth lead to a sub-optimal quantization. ALQ is the first scheme to explicitly optimize the loss function and incrementally train an adaptive bitwidth while without gradient approximation.

2.3 Preliminaries and Notations

We aim at multi-bit quantization with an adaptive bitwidth on a DNN consisting of L convolutional (`conv`) layers or fully connected (`fc`) layers. To simplify the notation, we start the discussion with a single layer and

extend to the entire network with L layers in the implementation section Section 2.4.4.

For a conv/fc layer, its weights dominate the resource consumption of storage and computation than other parameters, e.g., bias, batch normalization. We thus judiciously focus on quantizing the weight tensor of the conv/fc layer l. To allow an adaptive bitwidth, we structure the weight tensor of the layer l in *disjoint groups*. The weights in a single group will be quantized into the same bitwidth, whereas different group may have an adaptive different bitwidth. Specifically, for the *vectorized* weight tensor $\boldsymbol{w}_l \in \mathbb{R}^N$ of layer l, we divide $\boldsymbol{w}_l$ into G disjoint groups. For simplicity, we omit the subscript l in the following discussion. Each group of weights is denoted by $\boldsymbol{w}_g$, where $\boldsymbol{w}_g \in \mathbb{R}^n$ and $N = n \times G$. In other words, the overall N weights in layer l are evenly partitioned into G groups, see more details in Section 2.6.2.1. Then the multi-bit quantized weights $\hat{\boldsymbol{w}}_g$ of group g are formulated as,

$$\hat{\boldsymbol{w}}_g = \sum_{i=1}^{I_g} \alpha_i \boldsymbol{\beta}_i = \boldsymbol{B}_g \boldsymbol{\alpha}_g \tag{2.1}$$

where $\boldsymbol{\beta}_i \in \{-1, +1\}^{n\times 1}$ and $\alpha_i \in \mathbb{R}_+$ are the i-th binary basis and the corresponding coordinate; I_g represents the quantization bitwidth, i.e., the number of binary bases, of group g. $\boldsymbol{B}_g \in \{-1, +1\}^{n\times I_g}$ and $\boldsymbol{\alpha}_g \in \mathbb{R}_+^{I_g\times 1}$ are the matrix forms of the binary bases and the coordinates. We further denote $\boldsymbol{\alpha} = \boldsymbol{\alpha}_{1:G}$ as vectorized coordinates $\boldsymbol{\alpha}_g$ of all weight groups, and $\boldsymbol{B} = \boldsymbol{B}_{1:G}$ as concatenated binary bases $\boldsymbol{B}_g$ of all weight groups. A layer l quantized as above yields an average bitwidth

$$I = \frac{1}{G}\sum_{g=1}^{G} I_g \tag{2.2}$$

2.4 Adaptive Loss-Aware Quantization

2.4.1 Weight Quantization Overview

Problem Formulation. ALQ quantizes weights by directly minimizing the loss function rather than the reconstruction error. For layer l, the process can be formulated as the following optimization problem.

$$\min_{\hat{\boldsymbol{w}}_{1:G}} \quad \ell\left(\hat{\boldsymbol{w}}_{1:G}\right) \tag{2.3}$$

$$\text{s.t.} \quad \hat{\boldsymbol{w}}_g = \sum_{i=1}^{I_g} \alpha_i \boldsymbol{\beta}_i = \boldsymbol{B}_g \boldsymbol{\alpha}_g \quad \forall g \in 1, \ldots, G \tag{2.4}$$

$$\text{card}(\boldsymbol{\alpha}) = I \times G \leq I_{\min} \times G \tag{2.5}$$

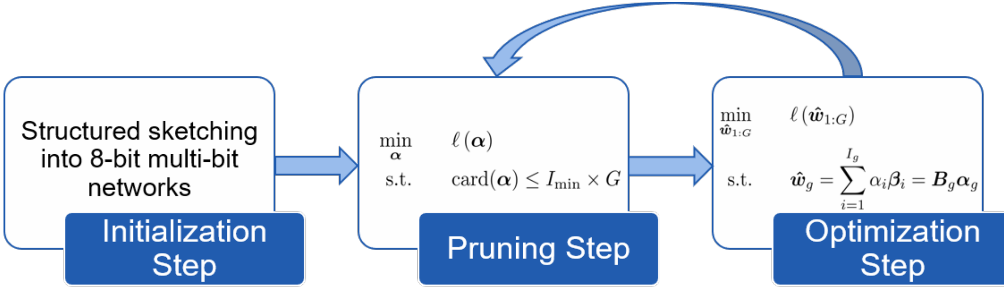

Figure 2.1: The figure depicts the overall approach of ALQ. In Initialization Step, the pretrained full precision weights are separated into disjoint groups and then are quantized into an 8-bit multi-bit form. In Pruning Step, we search an adaptive different bitwidth for each group of weights by removing the unimportant α's w.r.t. the loss. Based on the searched bitwidth assignment, we further conduct an Optimization step to train the remaining binary bases $\boldsymbol{B}_g$ and coordinates $\boldsymbol{\alpha}_g$. Both Pruning Step and Optimization Step are conducted iteratively.

where ℓ is the loss; card(.) denotes the cardinality of the set, i.e., the total number of elements in $\boldsymbol{\alpha}$; $I_{\min}$ is the desirable average bitwidth, which is determined by the storage constraints on edge devices. Since the group size n is the same in one layer, card($\boldsymbol{\alpha}$) is proportional to the storage consumption.

Solution Pipeline. The constrained domain of Eq. (2.4) and Eq. (2.5) are both discrete and non-convex. Directly conducting an exhaustive searching is NP-hard and infeasible on current DNNs. Therefore, we propose to narrow down the search space and disentangle the constraints into two sub-problems. Particularly, our ALQ solves the optimization problem in Eq. (2.3)-Eq. (2.5) by three steps. The overall approach is shown in Figure 2.1. The pseudocode of the entire pipeline is illustrated in Algorithm 2.5 in Section 2.4.4.4.

- Initialization Step: **Structured Sketching** (Section 2.4.4.1). In this step, we adapt the network sketching in [GYZC17], and propose a structured sketching algorithm. It first partitions the pretrained full precision weights $\boldsymbol{w}$ into G groups; then quantizes each $\boldsymbol{w}_g$ into its 8-bit multi-bit form $\hat{\boldsymbol{w}}_g$ by greedily searching the optimal binary basis vector $\boldsymbol{\beta}_i$ and the optimal scaling factor α_i. This step not only provides a good initial point for the following steps, but also restricts each group to a maximal 8-bit to reduce the search space.

- Pruning Step: **Pruning in α Domain** (Section 2.4.2 and Section 2.4.4.2). This step starts from the initialized 8-bit network obtained in Initialization Step, and then progressively reduces the average bitwidth I by pruning the least important (w.r.t. the loss)

coordinates in $\boldsymbol{\alpha}$ domain. Note that removing an element α_i will also lead to the removal of the binary basis $\boldsymbol{\beta}_i$, which in effect results in a smaller bitwidth I_g for group g. This way, no sparse tensor is introduced. Note that sparse tensors could lead to a detrimental irregular computation. Since the importance of each weight group differs, the resulting I_g varies across groups, and thus contributes to an adaptive bitwidth I_g for each group. In this step, we only set some elements of $\boldsymbol{\alpha}$ to zero (also remove them from $\boldsymbol{\alpha}$ leading to a reduced I_g) without changing the others. The sub-problem for Pruning Step is:

$$\min_{\boldsymbol{\alpha}} \quad \ell(\boldsymbol{\alpha}) \tag{2.6}$$

$$\text{s.t.} \quad \text{card}(\boldsymbol{\alpha}) \leq I_{\min} \times G \tag{2.7}$$

- Optimization Step: **Optimizing Binary Bases $\boldsymbol{B}_g$ and Coordinates $\boldsymbol{\alpha}_g$** (Section 2.4.3 and Section 2.4.4.3). In this step, we retrain the remaining binary bases and coordinates to recover the accuracy degradation induced by the bitwidth reduction. Similar to [XYL+18], we take an alternative approach for better accuracy recovery. Specifically, we first search for a new set of binary bases w.r.t. the loss given fixed coordinates. Then we optimize the coordinates by fixing the binary bases. The sub-problem for Optimization Step is:

$$\min_{\hat{\boldsymbol{w}}_{1:G}} \quad \ell(\hat{\boldsymbol{w}}_{1:G}) \tag{2.8}$$

$$\text{s.t.} \quad \hat{\boldsymbol{w}}_g = \sum_{i=1}^{I_g} \alpha_i \boldsymbol{\beta}_i = \boldsymbol{B}_g \boldsymbol{\alpha}_g \quad \forall g \in 1, ..., G \tag{2.9}$$

For a higher accuracy, state-of-the-art unstructured pruning methods [HMD16, FC19] often conduct pruning and sparse fine-tuning iteratively rather than the one-shot manner. Similarly, we also conduct our Pruning Step and our Optimization Step *iteratively* until the average bitwidth reaches the desired bitwidth. Namely, the original problem of Eq. (2.3)-Eq. (2.5) is decoupled into two sub-problems of Eq. (2.6)-Eq. (2.7) and Eq. (2.8)-Eq. (2.9), and the two sub-problems are solved iteratively.

Optimizer Framework. We consider both Pruning Step and Optimization Step above as an optimization problem with *domain constraints*, and solve them using the same optimization framework: subgradient methods with projection update [DHS11].

The optimization problem in Eq. (2.8)-Eq. (2.9) imposes domain constraints on $\boldsymbol{B}_g$ because they can only be discrete binary bases. The optimization problem in Eq. (2.6)-Eq. (2.7) can be considered as with a trivial domain constraint: the output $\boldsymbol{\alpha}$ should be a subset (subvector)

of the input $\boldsymbol{\alpha}$. Furthermore, the feasible sets for both $\boldsymbol{B}_g$ and $\boldsymbol{\alpha}$ are bounded.

Subgradient methods with projection update are effective to solve problems in the form of $\min_{\boldsymbol{\theta}}(\ell(\boldsymbol{\theta}))$ s.t. $\boldsymbol{\theta} \in \Theta$ [DHS11]. We apply AMSGrad [RKK18], an adaptive stochastic subgradient method with projection update, as the common optimizer framework in Pruning Step and Optimization Step. At training iteration s, AMSGrad generates the next update as,

$$\begin{aligned} \boldsymbol{\theta}^{s+1} &= \Pi_{\Theta, \sqrt{\hat{\boldsymbol{V}}^s}}(\boldsymbol{\theta}^s - a^s \boldsymbol{m}^s / \sqrt{\hat{\boldsymbol{v}}^s}) \\ &= \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{argmin}} \,\|(\sqrt{\hat{\boldsymbol{V}}^s})^{1/2}(\boldsymbol{\theta} - (\boldsymbol{\theta}^s - \frac{a^s \boldsymbol{m}^s}{\sqrt{\hat{\boldsymbol{v}}^s}}))\| \end{aligned} \tag{2.10}$$

where Π is a projection operator; Θ is the feasible domain of $\boldsymbol{\theta}$; a^s is the learning rate; $\boldsymbol{m}^s$ is the (unbiased) first momentum; $\hat{\boldsymbol{v}}^s$ is the (unbiased) maximum second momentum; and $\hat{\boldsymbol{V}}^s$ is the diagonal matrix of $\hat{\boldsymbol{v}}^s$.

In our context, Eq. (2.10) can be written as,

$$\hat{\boldsymbol{w}}_g^{s+1} = \underset{\hat{\boldsymbol{w}}_g\in\mathbb{W}}{\operatorname{argmin}} f^s(\hat{\boldsymbol{w}}_g) \tag{2.11}$$

$$f^s = (a^s \boldsymbol{m}^s)^{\mathrm{T}}(\hat{\boldsymbol{w}}_g - \hat{\boldsymbol{w}}_g^s) + \frac{1}{2}(\hat{\boldsymbol{w}}_g - \hat{\boldsymbol{w}}_g^s)^{\mathrm{T}} \sqrt{\hat{\boldsymbol{V}}^s}(\hat{\boldsymbol{w}}_g - \hat{\boldsymbol{w}}_g^s) \tag{2.12}$$

where $\mathbb{W}$ is the feasible domain of $\hat{\boldsymbol{w}}_g$.

Pruning Step and Optimization Step have different feasible domains of $\mathbb{W}$ according to their objective (see details in Section 2.4.2 and Section 2.4.3). Eq. (2.12) approximates the loss increment incurred by $\hat{\boldsymbol{w}}_g$ around the current point $\hat{\boldsymbol{w}}_g^s$ as a quadratic model function under domain constraints [DdVB15, DHS11, RKK18]. For simplicity, we replace $a^s \boldsymbol{m}^s$ with $\boldsymbol{g}^s$ and replace $\sqrt{\hat{\boldsymbol{V}}^s}$ with $\boldsymbol{H}^s$. $\boldsymbol{g}^s$ and $\boldsymbol{H}^s$ are updated by the loss gradient of $\hat{\boldsymbol{w}}_g^s$. Thus, the required input of each AMSGrad step is $\partial\ell^s / \partial\hat{\boldsymbol{w}}_g^s$. It can be directly obtained during the backward, since $\hat{\boldsymbol{w}}_g^s$ is used as an intermediate value during the forward, .

2.4.2 Pruning in α Domain

As introduced in Section 2.4.1, we reduce the average bitwidth I by pruning the elements in $\boldsymbol{\alpha}$ w.r.t. the resulting loss. If one element α_i in $\boldsymbol{\alpha}$ is pruned, the corresponding dimension $\boldsymbol{\beta}_i$ is also removed from $\boldsymbol{B}$. Now we explain how to instantiate the optimizer in Eq. (2.11) to solve Eq. (2.6)-Eq. (2.7) of Pruning Step.

As discussed above, pruning in α domain is regarded as an optimization problem solved in multiple training iterations. Thus, the cardinality of the chosen subset (i.e., the average bitwidth) is uniformly reduced over training iterations. For example, assume there are T training

iterations in total, the initial average bitwidth is I^0 and the desired average bitwidth after T iterations I^T is $I_{\min}$. Then at each iteration t, ($M_p = (I^0 - I_{\min}) \times G/T$) of α_i^t's are pruned. This way, the cardinality after T iterations will be smaller than $I_{\min} \times G$.

When pruning in the $\boldsymbol{\alpha}$ domain, $\boldsymbol{B}$ is considered as invariant. Hence Eq. (2.11) and Eq. (2.12) become,

$$\boldsymbol{\alpha}^{t+1} = \underset{\boldsymbol{\alpha} \in \mathbb{A}}{\operatorname{argmin}} \, f_{\boldsymbol{\alpha}}^t(\boldsymbol{\alpha}) \tag{2.13}$$

$$f_{\boldsymbol{\alpha}}^t = (\boldsymbol{g}_{\boldsymbol{\alpha}}^t)^{\mathrm{T}}(\boldsymbol{\alpha} - \boldsymbol{\alpha}^t) + \frac{1}{2}(\boldsymbol{\alpha} - \boldsymbol{\alpha}^t)^{\mathrm{T}} \boldsymbol{H}_{\boldsymbol{\alpha}}^t (\boldsymbol{\alpha} - \boldsymbol{\alpha}^t) \tag{2.14}$$

where $\boldsymbol{g}_{\boldsymbol{\alpha}}^t$ and $\boldsymbol{H}_{\boldsymbol{\alpha}}^t$ are similar to the ones in Eq. (2.12) but are in the $\boldsymbol{\alpha}$ domain. If α_i^t is pruned, the i-th element in $\boldsymbol{\alpha}$ is set to 0 in the above Eq. (2.13) and Eq. (2.14). Thus, the constrained domain $\mathbb{A}$ is taken as all possible vectors with M_p zero elements in $\boldsymbol{\alpha}^t$.

AMSGrad uses a diagonal matrix of $\boldsymbol{H}_{\boldsymbol{\alpha}}^t$ in the quadratic model function, which decouples each element in $\boldsymbol{\alpha}^t$. This means the loss increment caused by several α_i^t equals the sum of the increments caused by them individually, which are calculated as,

$$f_{\boldsymbol{\alpha},i}^t = -g_{\boldsymbol{\alpha},i}^t \, \alpha_i^t + \frac{1}{2} \, H_{\boldsymbol{\alpha},ii}^t \, (\alpha_i^t)^2 \tag{2.15}$$

All items of $f_{\boldsymbol{\alpha},i}^t$ are sorted in ascending. Then the first M_p items (α_i^t) in the sorted list are removed from $\boldsymbol{\alpha}^t$, and results in a smaller cardinality $I^t \times G$. The input of the AMSGrad step in $\boldsymbol{\alpha}$ domain is the loss gradient of $\boldsymbol{\alpha}_g^t$, which can be computed with the chain rule,

$$\frac{\partial \ell^t}{\partial \boldsymbol{\alpha}_g^t} = \boldsymbol{B}_g^{t\,\mathrm{T}} \frac{\partial \ell^t}{\partial \hat{\boldsymbol{w}}_g^t} \tag{2.16}$$

$$\hat{\boldsymbol{w}}_g^t = \boldsymbol{B}_g^t \boldsymbol{\alpha}_g^t \tag{2.17}$$

Our pipeline allows to reduce the bitwidth smoothly, since the average bitwidth can be floating-point. In ALQ, since different layers have a similar group size (see in Section 2.6.2.1), the loss increment caused by pruning is sorted among all layers, such that only a global pruning number needs to be determined. More details are explained in Section 2.4.4.4. This Pruning Step not only provides a loss-aware adaptive bitwidth, but also seeks a better initialization for the successive Optimization Step, since low-bit quantized weights may be relatively far from their original full precision values.

2.4.3 Optimizing Binary Bases and Coordinates

After pruning, the loss degradation needs to be recovered. Following Eq. (2.11), the objective in Optimization Step is

$$\hat{\boldsymbol{w}}_g^{s+1} = \underset{\hat{\boldsymbol{w}}_g \in \mathbb{W}}{\operatorname{argmin}} \, f^s(\hat{\boldsymbol{w}}_g) \tag{2.18}$$

The constrained domain $\mathbb{W}$ is decided by, both binary bases and full precision coordinates. Hence directly searching for the optimal $\hat{\boldsymbol{w}}_g$ is NP-hard. Instead, we optimize $\boldsymbol{B}_g$ and $\boldsymbol{\alpha}_g$ in an alternative manner, as prior multi-bit quantization works [XYL+18, ZYYH18] that minimize the reconstruction error.

Optimizing $\boldsymbol{B}_g$. We directly search for the optimal bases with AMSGrad. In each training iteration q, we fix $\boldsymbol{\alpha}_g^q$, and update $\boldsymbol{B}_g^q$. We find the optimal increment for each group of weights, such that it converts to a new set of binary bases, $\boldsymbol{B}_g^{q+1}$. This Optimization Step searches a new space spanned by $\boldsymbol{B}_g^{q+1}$ based on the loss reduction, which prevents the pruned space to be always a subspace of the previous one.

According to Eq. (2.11) and Eq. (2.12), the optimal $\boldsymbol{B}_g$ w.r.t. the loss is updated by,

$$\boldsymbol{B}_g^{q+1} = \underset{\boldsymbol{B}_g \in \{-1,+1\}^{n \times I_g}}{\operatorname{argmin}} \; f^q(\boldsymbol{B}_g) \tag{2.19}$$

$$f^q = (\boldsymbol{g}^q)^{\mathrm{T}}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \hat{\boldsymbol{w}}_g^q) + \frac{1}{2}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \hat{\boldsymbol{w}}_g^q)^{\mathrm{T}}\boldsymbol{H}^q(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \hat{\boldsymbol{w}}_g^q) \tag{2.20}$$

where $\hat{\boldsymbol{w}}_g^q = \boldsymbol{B}_g^q\boldsymbol{\alpha}_g^q$.

Recall that $\boldsymbol{B}_g^q \in \{-1,+1\}^{n \times I_g}$. Since $\boldsymbol{H}^q$ is diagonal in AMSGrad, each row vector in $\boldsymbol{B}_g^{q+1}$ can be independently determined. For example, the j-th row is computed as,

$$\boldsymbol{B}_{g,j}^{q+1} = \underset{\boldsymbol{B}_{g,j}}{\operatorname{argmin}} \|\boldsymbol{B}_{g,j}\boldsymbol{\alpha}_g^q - (\hat{w}_{g,j}^q - g_j^q/H_{jj}^q)\|, \quad j \in 1,...,n \tag{2.21}$$

Since in general $n >> I_g$, to reduce the computation complexity, we firstly compute all 2^{I_g} possible values of

$$\boldsymbol{b}^{\mathrm{T}}\boldsymbol{\alpha}_g^q\,, \quad \boldsymbol{b}^{\mathrm{T}} \in \{-1,+1\}^{1 \times I_g} \tag{2.22}$$

Then each row vector $\boldsymbol{B}_{g,j}^{q+1}$ can be directly substituted with the optimal $\boldsymbol{b}^{\mathrm{T}}$ through an exhaustive searching in 2^{I_g} values.

Optimizing $\boldsymbol{\alpha}_g$. The above obtained set of binary bases $\boldsymbol{B}_g$ spans a new I_g-dim linear space, which is a subspace of original n-dim full space. The current $\boldsymbol{\alpha}_g$ is unlikely to be the optimal point in this I_g-dim space, so now we optimize $\boldsymbol{\alpha}_g$. Since $\boldsymbol{\alpha}_g$ is in full precision, i.e., $\boldsymbol{\alpha}_g \in \mathbb{R}^{I_g \times 1}$, there is no domain constraint and thus no need for projection updating. Similar to optimizing full precision $\boldsymbol{w}_g$, conventional training strategies can be directly used to optimize $\boldsymbol{\alpha}_g$.

Similar to Eq. (2.13) and Eq. (2.14), we use AMSGrad optimizer in $\boldsymbol{\alpha}$ domain without projection updating, for each group in the p-th training iteration as,

$$\boldsymbol{\alpha}_g^{p+1} = \boldsymbol{\alpha}_g^p - a_{\boldsymbol{\alpha}}^p \boldsymbol{m}_{\boldsymbol{\alpha}}^p / \sqrt{\hat{\boldsymbol{v}}_{\boldsymbol{\alpha}}^p} \tag{2.23}$$

We also add an L2-norm regularization on $\boldsymbol{\alpha}_g$ to enforce unimportant coordinates to zero. If there is a negative value in $\boldsymbol{\alpha}_g$, the corresponding basis is set to its negative complement, to keep $\boldsymbol{\alpha}_g$ semi-positive definite. Optimizing $\boldsymbol{B}_g$ and $\boldsymbol{\alpha}_g$ does not influence the number of binary bases I_g.

Optimization Speedup. Since $\boldsymbol{\alpha}_g$ is full precision, updating $\boldsymbol{\alpha}_g^q$ is much cheaper than exhaustively search $\boldsymbol{B}_g^{q+1}$. Even if the main purpose of the first step in Optimization Step is optimizing bases, we also add an updating process for $\boldsymbol{\alpha}_g^q$ in each training iteration q.

We fix $\boldsymbol{B}_g^{q+1}$, and update $\boldsymbol{\alpha}_g^q$. The overall increment of quantized weights from both updating processes is,

$$\hat{\boldsymbol{w}}_g^{q+1} - \hat{\boldsymbol{w}}_g^q = \boldsymbol{B}_g^{q+1}\boldsymbol{\alpha}_g^{q+1} - \boldsymbol{B}_g^q\boldsymbol{\alpha}_g^q \tag{2.24}$$

Substituting Eq. (2.24) into Eq. (2.11) and Eq. (2.12), we have,

$$\boldsymbol{\alpha}_g^{q+1} = -((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}\boldsymbol{H}^q\boldsymbol{B}_g^{q+1})^{-1} \times ((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}(\boldsymbol{g}^q - \boldsymbol{H}^q\boldsymbol{B}_g^q\boldsymbol{\alpha}_g^q)) \tag{2.25}$$

To ensure the inverse in Eq. (2.25) exists, we add a small diagonal matrix $\lambda\mathbf{I}$ to Eq. (2.25),

$$\boldsymbol{\alpha}_g^{q+1} = -((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}\boldsymbol{H}^q\boldsymbol{B}_g^{q+1} + \lambda\mathbf{I})^{-1} \times ((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}(\boldsymbol{g}^q - \boldsymbol{H}^q\boldsymbol{B}_g^q\boldsymbol{\alpha}_g^q)) \tag{2.26}$$

where $\lambda = 10^{-6}$.

2.4.4 Implementation

In this section, we discuss the detailed implementation of ALQ. We elaborate the pseudocodes of three steps and analyze their complexity. Note that the discussion in this section is extended to the entire networks with L layers, thus we reintroduce the layer index l for clarity reasons.

2.4.4.1 Implementation of Initialization Step

We adapt the network sketching in [GYZC17], and propose a structured sketching algorithm for Initialization Step, see Algorithm 2.1[1]. This algorithm partitions the pretrained full precision weights $\boldsymbol{w}_l$ of the l-th layer into G_l groups. We study the different structures of grouping in Section 2.6.2.1. The vectorized weights $\boldsymbol{w}_{l,g}$ of each group are quantized with $I_{l,g}$ linear independent binary bases (i.e., column vectors in $\boldsymbol{B}_{l,g}$) and corresponding coordinates $\boldsymbol{\alpha}_{l,g}$ to minimize the reconstruction error. This algorithm initializes the matrix of binary bases $\boldsymbol{B}_{l,g}$, the vector of floating-point coordinates $\boldsymbol{\alpha}_{l,g}$, and the scalar of integer bitwidth $I_{l,g}$ in each group across layers. The initial reconstruction error is upper bounded by a threshold σ. In addition, a maximum bitwidth of each group is defined as $I_{\max}$. Both of these two parameters determine the initial bitwidth $I_{l,g}$.

Algorithm 2.1: Structured sketching of weights

```
   Input: w_{1:L}, G_{1:L}, I_max, σ
   Output: {{α_{l,g}, B_{l,g}, I_{l,g}}_{g=1}^{G_l}}_{l=1}^{L}
1  for l ← 1 to L do
2      for g ← 1 to G_l do
3          Fetch and vectorize w_{l,g} from w_l;
4          Initialize ε = w_{l,g}, i = 0;
5          B_{l,g} = [ ];
6          while ||ε ⊘ w_{l,g}||_2^2 > σ and i < I_max do
7              i = i + 1;
8              β_i = sign(ε);
9              B_{l,g} = [B_{l,g}, β_i];
               /* Find the optimal point spanned by B_{l,g}      */
10             α_{l,g} = (B_{l,g}^T B_{l,g})^{-1} B_{l,g}^T w_{l,g};
               /* Update the residual reconstruction error  */
11             ε = w_{l,g} − B_{l,g} α_{l,g};
12         I_{l,g} = i;
```

We discuss the choice of group size n, and the maximum bitwidth $I_{\max}$ in Section 2.6.2.

Theorem 2.1. *The column vectors in $\boldsymbol{B}_{l,g}$ are linear independent.*

Proof. The instruction $\boldsymbol{\alpha}_{l,g} = (\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{B}_{l,g})^{-1}\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{w}_{l,g}$ ensures $\boldsymbol{\alpha}_{l,g}$ is the optimal point in $\mathrm{span}(\boldsymbol{B}_{l,g})$ regarding the least square reconstruction error $\boldsymbol{\epsilon}$. Thus, $\boldsymbol{\epsilon}$ is orthogonal to $\mathrm{span}(\boldsymbol{B}_{l,g})$. The new basis is computed from the next iteration by $\boldsymbol{\beta}_i = \mathrm{sign}(\boldsymbol{\epsilon})$. Since $\mathrm{sign}(\boldsymbol{\epsilon}) \cdot \boldsymbol{\epsilon} > 0, \forall \boldsymbol{\epsilon} \neq \boldsymbol{0}$, we have $\boldsymbol{\beta}_i \notin \mathrm{span}(\boldsymbol{B}_{l,g})$. Thus, the iteratively generated column vectors in $\boldsymbol{B}_{l,g}$ are linear independent. This also means the square matrix of $\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{B}_{l,g}$ is invertible. □

2.4.4.2 Implementation of Pruning Step

As discussed in Section 2.4.2, α_i's are pruned iteratively in mini-batches. During each Pruning Step, for example, 30% of α_i's are iteratively pruned in one epoch. Due to the high complexity of sorting all $f_{\alpha,i}$, sorting is firstly executed in each layer, and the top-k% $f_{\alpha_l,i}$ of the l-th layer are selected to resort again for pruning. Recall that l stands for the layer index. k is generally small, e.g., 1 or 0.5, which ensures that the pruned α_i's in one iteration do not always come from a single layer. There are n_l weights in each group, and G_l groups in the l-th layer. The sorting

[1] Circled operation in Algorithm 2.1 means elementwise operations.

Algorithm 2.2: Pruning in α domain

Input: $T, M_T, k, \{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, training dataset

Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$

```
1  Compute M_0 with Eq. (2.27);
2  Compute the pruning number per iteration M_p = round((M_0 - M_T)/T);
3  for t ← 1 to T do
4      for l ← 1 to L do
5          Update ŵ^t_{l,g} = B^t_{l,g} α^t_{l,g};
6          Forward propagate;
7      Compute the loss ℓ^t;
8      for l ← L to 1 do
9          Backward propagate gradient ∂ℓ^t/∂ŵ^t_{l,g};
10         Compute ∂ℓ^t/∂α^t_{l,g} with Eq. (2.16);
11         Update momentums of AMSGrad in α domain;
12         for α^t_{l,i} in α^t_l do
13             Compute f^t_{α_{l,i}} with Eq. (2.15);
14         Sort and select Top-k% f^t_{α_{l,i}} in ascending order;
15     Resort the selected {f^t_{α_{l,i}}}^L_{l=1} in ascending order;
16     Remove Top-M_p α^t_{l,i} and their binary bases;
17     Update {{α^{t+1}_{l,g}, B^{t+1}_{l,g}, I^{t+1}_{l,g}}^{G_l}_{g=1}}^L_{l=1};
```

complexity mainly depends on the sorting in the most critical layer that has the largest card($\boldsymbol{\alpha}_l$).

The Pruning Step is elaborated in Algorithm 2.2. Here, assume that there are altogether T pruning (training) iterations in each execution of Pruning Step; the total number of α_i's across all layers is M_0 before pruning, i.e.,

$$M_0 = \sum_l \sum_g \text{card}(\boldsymbol{\alpha}_{l,g}) \tag{2.27}$$

and the desired total number of α_i's after pruning is M_T.

2.4.4.3 Implementation of Optimization Step

Optimization Step is also executed in batch training. Since $\boldsymbol{\alpha}_g$ is floating-point value, the complexity of optimizing $\boldsymbol{\alpha}_g$ is the same as the conventional optimization (see Algorithm 2.3). Assume that there are altogether P training iterations. It is worth noting that both the bitwidth $I_{l,g}$ and the binary bases $\boldsymbol{B}_{l,g}$ do not change in this step; only the coordinates $\boldsymbol{\alpha}_{l,g}$ are updated over P iterations.

Optimizing $\boldsymbol{B}_g$ with speedup is presented in Algorithm 2.4. Assume

Algorithm 2.3: Optimizing $\boldsymbol{\alpha}_g$

Input: P, $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, training dataset

Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$

```
1  for p ← 1 to P do
2      for l ← 1 to L do
3          Update ŵ^p_{l,g} = B_{l,g} α^p_{l,g};
4          Forward propagate;
5      Compute the loss ℓ^p;
6      for l ← L to 1 do
7          Backward propagate gradient ∂ℓ^p/∂ŵ^p_{l,g};
8          Compute ∂ℓ^p/∂α^p_{l,g} with Eq. (2.16);
9          Update momentums of AMSGrad in α domain;
10         for g ← 1 to G_l do
11             Update α^{p+1}_{l,g} with Eq. (2.23);
```

Algorithm 2.4: Optimizing $\boldsymbol{B}_g$ with speedup

Input: Q, $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, training dataset

Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$

```
1  for q ← 1 to Q do
2      for l ← 1 to L do
3          Update ŵ^q_{l,g} = B^q_{l,g} α^q_{l,g};
4          Forward propagate;
5      Compute the loss ℓ^q ;
6      for l ← L to 1 do
7          Backward propagate gradient ∂ℓ^q/∂ŵ^q_{l,g};
8          Update momentums of AMSGrad;
9          for g ← 1 to G_l do
10             Compute all values of Eq. (2.22);
11             for j ← 1 to n_l do
12                 Update B^{q+1}_{l,g,j} with Eq. (2.21);
13             Update α^{q+1}_{l,g} with Eq. (2.26);
```

that there are altogether Q training iterations. It is worth noting that the bitwidth $I_{l,g}$ does not change in this step; only the binary bases $\boldsymbol{B}_{l,g}$ and the coordinates $\boldsymbol{\alpha}_{l,g}$ are updated over Q iterations.

The extra complexity related to the original AMSGrad mainly comes

from two parts, Eq. (2.21) and Eq. (2.26). Eq. (2.21) is also the most resource-hungry step of the whole pipeline, since it requires an exhaustive search. For each group, Eq. (2.21) takes both time and storage complexities of $O(n \cdot 2^{I_g})$, and in general $n >> I_g \geq 1$. Since $\boldsymbol{H}^q$ is a diagonal matrix, most of the matrix-matrix multiplications in Eq. (2.26) is avoided through matrix-vector multiplication and matrix-diagonalmatrix multiplication. Thus, the time complexity trims down to $O(nI_g + nI_g^2 + I_g^3 + nI_g + n + n + nI_g + I_g^2) \doteq O(n(I_g^2 + 3I_g + 2))$.

2.4.4.4 Implementation of the Pipeline

The entire pipeline of ALQ is demonstrated in Algorithm 2.5. For Initialization Step, the pretrained full precision weights $\boldsymbol{w}_{1:L}$ are required. Then, we need to specify the structure used in each layer, i.e., the structure of grouping $G_{1:L}$. In addition, a maximum bitwidth $I_{\max}$ and a threshold σ for the residual reconstruction error also need to be determined (see more details in Section 2.4.4.1). After initialization, we might need to retrain the model with several epochs of Algorithm 2.4 to recover the accuracy degradation caused by the initialization.

Then, we need to determine the number of outer iterations R, i.e., how many times the Pruning Step is executed. A pruning schedule $M^{1:R}$ is also required. M^r determines the total number of remaining α_i's (across all layers) after the r-th Pruning Step, which is also taken as the input M_T in Algorithm 2.2. For example, we can build this schedule by pruning 30% of α_i's during each execution of Pruning Step, as,

$$M^{r+1} = M^r \times (1 - 0.3) \tag{2.28}$$

with $r \in 0, 1, 2, ..., R-1$. M^0 represents the total number of α_i's (across all layers) after initialization.

For Pruning Step, other individual inputs include the total number of iterations T, and the selected percentages k for sorting (see Algorithm 2.2). For Optimization Step, the individual inputs includes the total number of iterations Q in optimizing $\boldsymbol{B}_g$ (see Algorithm 2.4), and the total number of iterations P in optimizing $\boldsymbol{\alpha}_g$ (see Algorithm 2.3).

2.5 Activation Quantization

To leverage bitwise operations for speedup, the inputs of each layer (i.e., the activation output of the last layer) also need to be quantized into the multi-bit form. We quantize activations with the same binary basis (i.e., $\{-1, +1\}$) as the aforementioned weight quantization.

Our activation quantization follows the idea proposed in [CWV+18], i.e., a parameterized clipping for fixed-point activation quantization, but it is adapted to the multi-bit form. Specially, we replace ReLU with a

Algorithm 2.5: Adaptive Loss-aware Quantization for multi-bit networks

Input: Pretrained full precision weights $\boldsymbol{w}_{1:L}$, structures $G_{1:L}$, $I_{\max}$, σ, T, pruning schedule $M^{1:R}$, k, P, Q, R, training dataset

Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$

```
/* Initialization Step: */
```

1 Initialize $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$ with Algorithm 2.1;

2 **for** $r \leftarrow 1$ **to** R **do**

```
/* Pruning Step: */
```

3 Assign M^r to the input M_T of Algorithm 2.2;

4 Prune in $\boldsymbol{\alpha}$ domain with Algorithm 2.2;

```
/* Optimization Step: */
```

5 Optimize binary bases with Algorithm 2.4;

6 Optimize coordinates with Algorithm 2.3;

step activation function. The vectorized activation $\boldsymbol{x}$ of the l-th layer is quantized as,

$$\boldsymbol{x} \doteq \hat{\boldsymbol{x}} = x_{\text{ref}} + \boldsymbol{D}\boldsymbol{\gamma} = \boldsymbol{D}'\boldsymbol{\gamma}' \tag{2.29}$$

where $\boldsymbol{D} \in \{-1, +1\}^{N_x \times I_x}$, and $\boldsymbol{\gamma} \in \mathbb{R}_+^{I_x \times 1}$. $\boldsymbol{\gamma}'$ is a column vector formed by $[x_{\text{ref}}, \boldsymbol{\gamma}^{\text{T}}]^{\text{T}}$; $\boldsymbol{D}'$ is a matrix formed by $[\mathbf{1}^{N_x \times 1}, \boldsymbol{D}]$. N_x is the dimension of $\boldsymbol{x}$, and I_x is the quantization bitwidth for activations. x_{ref} is the introduced layerwise (positive floating-point) reference to fit the output range of ReLU. During inference, x_{ref} is convoluted with the weights of the next layer and added to the bias. Hence the introduction of x_{ref} does not lead to extra computations. The output of the last layer is not quantized, as it does not involve computations anymore. For other settings, we mainly follow the ones used in [ZYYH18]. $\boldsymbol{\gamma}$ and x_{ref} are updated during the forward propagation with a running average to minimize the squared reconstruction error as,

$$\boldsymbol{\gamma}'_{\text{new}} = (\boldsymbol{D}'^{\text{T}}\boldsymbol{D}')^{-1}\boldsymbol{D}'^{\text{T}}\boldsymbol{x} \tag{2.30}$$

$$\boldsymbol{\gamma}' = 0.9\boldsymbol{\gamma}' + (1 - 0.9)\boldsymbol{\gamma}'_{\text{new}} \tag{2.31}$$

The (quantized) weights are also further fine-tuned with our optimizer to resume the accuracy drop. Here, we only set a global bitwidth for all layers in activation quantization.

2.6 Experiments

In this section, we implement ALQ with Pytorch [PGC+17], and evaluate its performance on MNIST [LC10], CIFAR10 [KNH09], and ImageNet [RDS+15] using LeNet5 [LBB+98], VGGNet [HYK17, RORF16],

and ResNet18/34 [HZRS16], respectively. The Top-1 test accuracy is reported, when the validation dataset has the highest accuracy during training. We first conduct the experiments on Initialization Step (Section 2.6.2), Pruning Step (Section 2.6.4) and Optimization Step (Section 2.6.3) individually to study their impacts. Then, we benchmark ALQ on different datasets and compare ALQ with different state-of-the-art network compression methods.

2.6.1 Benchmarking Details

LeNet5 on MNIST. The MNIST dataset [LC10] consists of 28×28 gray scale images from 10 digit classes. We use 50000 samples in the training set for training, the rest 10000 for validation, and the 10000 samples in the test set for testing. We use a mini-batch with size of 128. We use the default hyperparameters proposed in [Pyt19a] to train LeNet5 for 100 epochs as the baseline of full precision version. The network architecture is presented as, 20C5 - MP2 - 50C5 - MP2 - 500FC - 10SVM.

VGGNet on CIFAR10. The CIFAR-10 dataset [KNH09] consists of 60000 32×32 color images in 10 object classes. We use 45000 samples in the training set for training, the rest 5000 for validation, and the 10000 samples in the test set for testing. We use a mini-batch with size of 128. We use the default Adam optimizer provided by Pytorch to train full precision parameters for 200 epochs as the baseline of the full precision version. The initial learning rate is 0.01, and it decays with 0.2 every 30 epochs. The network architecture is presented as, 2×128C3 - MP2 - 2×256C3 - MP2 - 2×512C3 - MP2 - 2×1024FC - 10SVM.

ResNet18/34 on ImageNet. The ImageNet dataset [RDS+15] consists of 1.28 million high-resolution images for classifying in 1000 object classes. The validation set contains 50k images, which are used to report the accuracy level. We use mini-batch with size of 256. The used ResNet18/34 is from [HZRS16]. We use the ResNet18/34 provided by Pytorch as the baseline of full precision version. The network architecture is the same as "resnet18/resnet34" in [Pyt19b].

2.6.2 Experiments on Initialization

As mentioned in Section 2.4.4.1, we propose a structured sketching for Initialization Step. Some important parameters in Algorithm 2.1 are discussed as below.

2.6.2.1 Group Size n

Researchers propose different structures e.g., layerwise, channelwise, to partition weights, and then quantize the weights in one structured group with the same bitwidth. To explore the redundancy among weights, we

conduct experiments on the different structures of grouping. Certainly, the weights in one layer can be arbitrarily selected to gather a group. However, due to the extra indexing cost, the weights are often sliced along the tensor dimensions and uniformly grouped.

According to [GYZC17], the squared reconstruction error of a single group decays with Eq. (2.32), where $\lambda \geq 0$.

$$||\epsilon||_2^2 \leq ||\boldsymbol{w}_g||_2^2 (1 - \frac{1}{n - \lambda})^{I_g} \tag{2.32}$$

If full precision values are stored in floating-point, i.e., 32-bit, the storage compression ratio in one layer can be written as,

$$r_s = \frac{N \times 32}{I \times N + I \times 32 \times \frac{N}{n}} \tag{2.33}$$

where N is the total number of weights in one layer; n is the number of weights in each group, i.e., $n = N/G$; I is the average bitwidth, $I = \frac{1}{G}\sum_{g=1}^{G} I_g$.

We analyse the trade-off between the reconstruction error and the storage compression ratio of different group size n. We choose the pretrained AlexNet [KSH12] and VGGNet [SZ15], and plot the curves of the average (per weight) reconstruction error related to the storage compression ratio of each layer under different sliced structures. We also randomly shuffle the weights in each layer, then partition them into groups with different sizes. We select one example plot which comes from the last conv layer ($256 \times 256 \times 3 \times 3$) of AlexNet [KSH12] (see Figure 2.2). The pretrained full precision weights are provided by Pytorch [PGC+17].

We found that there is not a significant difference between random groups and sliced groups along tensor dimensions. Only the group size influences the trade-off. We think the reason is that one layer always contains thousands of groups, such that the points presented by these groups are roughly scattered in the n-dim space. Furthermore, regarding the deployment on a 32-bit general microprocessor, the group size should be larger than 32 for efficient computation. In short, a group size from 32 to 512 achieves relatively good trade-off between the weight reconstruction error and the storage compression ratio. Accordingly, for conv layers, grouping in channelwise ($\boldsymbol{w}_{c,:,:,:}$), kernelwise ($\boldsymbol{w}_{c,d,:,:}$), and pointwise ($\boldsymbol{w}_{c,:,h,w}$) appears to be appropriate. Channelwise $\boldsymbol{w}_{c,:}$ and subchannelwise $\boldsymbol{w}_{c,d:d+n}$ grouping are suited for `fc` layers. For example, if each channel is sliced into 2 groups with the same size, we denote it as subchannelwise(2). In addition, the most frequently used structures in this chapter are pointwise (conv layers) and (sub)channelwise (`fc` layers), which align with the bit-packing approach in [PTT18], and could result in a more efficient deployment. Since many network architectures choose

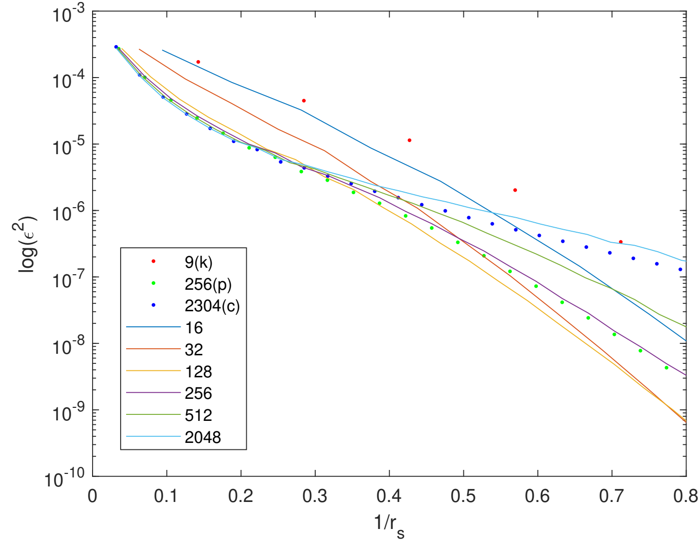

Figure 2.2: The curves about the logarithmic L2-norm of the average reconstruction error $\log(\|\epsilon\|_2^2)$ related to the reciprocal of the storage compression ratio $1/r_s$. The pretrained full precision weights are from the last conv layer of AlexNet. The legend demonstrates the corresponding group sizes. 'k' stands for kernelwise; 'p' stands for pointwise; 'c' stands for channelwise.

an integer multiple of 32 as the number of output channels in each layer, pointwise and (sub)channelwise are also efficient for the current storage format in 32-bit microprocessors.

2.6.2.2 Maximum Bitwidth $I_{\max}$

The initial I_g is decided by a predefined initial reconstruction precision or a maximum bitwidth. We notice that the accuracy degradation caused by the initialization can be fully recovered after several optimization epochs of Algorithm 2.4, if the maximum bitwidth is 8. For example, ResNet18 on ImageNet after such an initialization can be retrained to a Top-1/5 accuracy of 70.3%/89.4%, even higher than its full precision counterpart (69.8%/89.1%). For smaller networks, e.g., VGGNet on CIFAR10, a maximum bitwidth of 6 is already sufficient.

2.6.3 Convergence Analysis of Optimization Step

In this section, we conduct the ablation studies on our Optimization Step in Section 2.4.3. We show the advantages of our optimizer in terms of convergence. We mainly studied the convergence performance of Algorithm 2.4 (i.e., optimizing $\boldsymbol{B}_g$ with speedup) for two reasons, (*i*) it involves the domain constraints of binarization and takes the

majority of computation complexity; (*ii*) it conducts a similar alternative process as prior works [XYL+18, ZYYH18]. Recall that our optimizer in Algorithm 2.4 (*i*) has no gradient approximation and (*ii*) directly minimizes the loss. We developed the following two baselines for comparison.

- *STE with rec. error:* This baseline quantizes the maintained full precision weights by minimizing the reconstruction error (rather than the loss) during forward and approximates gradients via STE during backward. This approach is adopted in some of the best-performing quantization schemes such as LQ-Net [ZYYH18].
- *STE with loss-aware:* This baseline approximates gradients via STE but performs a loss-aware projection updating (adapted from our ALQ). It can be considered as a multi-bit extension of prior loss-aware quantizers for binary and ternary networks [HYK17, HK18]. See Section 2.6.3.1 below for more details.

2.6.3.1 The Optimizer of "STE with Loss-Aware"

In this section, we provide the details of the proposed *STE with loss-aware* optimizer. The training scheme of *STE with loss-aware* is similar to Algorithm 2.4, except that it maintains the full precision weights $\boldsymbol{w}_g$. See the pseudocode of *STE with loss-aware* in Algorithm 2.6.

For the layer l, the quantized weights $\hat{\boldsymbol{w}}_g$ is used during forward propagation. During backward propagation, the loss gradients to the full precision weights $\partial\ell/\partial\boldsymbol{w}_g$ are directly approximated with $\partial\ell/\partial\hat{\boldsymbol{w}}_g$, i.e., via STE in the q-th training iteration as,

$$\frac{\partial \ell^q}{\partial \boldsymbol{w}_g^q} = \frac{\partial \ell^q}{\partial \hat{\boldsymbol{w}}_g^q} \tag{2.34}$$

Then the first and second momentums in AMSGrad are updated with $\partial\ell^q/\partial\boldsymbol{w}_g^q$. Accordingly, the loss increment around $\boldsymbol{w}_g^q$ is modeled as,

$$f_{\text{ste}}^q = (\boldsymbol{g}^q)^{\mathrm{T}}(\boldsymbol{w}_g - \boldsymbol{w}_g^q) + \frac{1}{2}(\boldsymbol{w}_g - \boldsymbol{w}_g^q)^{\mathrm{T}}\boldsymbol{H}^q(\boldsymbol{w}_g - \boldsymbol{w}_g^q) \tag{2.35}$$

Since $\boldsymbol{w}_g$ is full precision, $\boldsymbol{w}_g^{q+1}$ can be directly obtained through the above AMSGrad step without projection updating,

$$\boldsymbol{w}_g^{q+1} = \boldsymbol{w}_g^q - (\boldsymbol{H}^q)^{-1}\boldsymbol{g}^q = \boldsymbol{w}_g^q - a^q\boldsymbol{m}^q/\sqrt{\hat{\boldsymbol{v}}^q} \tag{2.36}$$

Similarly, the loss increment caused by $\boldsymbol{B}_g$ (see Eq. (2.19) and Eq. (2.20)) is formulated as,

$$f_{\text{ste},\boldsymbol{B}}^q = (\boldsymbol{g}^q)^{\mathrm{T}}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \boldsymbol{w}_g^q) + \frac{1}{2}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \boldsymbol{w}_g^q)^{\mathrm{T}}\boldsymbol{H}^q(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \boldsymbol{w}_g^q) \tag{2.37}$$

Algorithm 2.6: STE with loss-aware

Input: Q, $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, training dataset
Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$

1 **for** $q \leftarrow 1$ **to** Q **do**
2   **for** $l \leftarrow 1$ **to** L **do**
3   Update $\hat{\boldsymbol{w}}_{l,g}^{q} = \boldsymbol{B}_{l,g}^{q}\boldsymbol{\alpha}_{l,g}^{q}$;
4   Forward propagate;
5   Compute the loss ℓ^q ;
6   **for** $l \leftarrow L$ **to** 1 **do**
7   Backward propagate gradient $\partial\ell^q/\partial\hat{\boldsymbol{w}}_{l,g}^{q}$;
8   Directly approximate $\partial\ell^q/\partial\boldsymbol{w}_{l,g}^{q}$ with $\partial\ell^q/\partial\hat{\boldsymbol{w}}_{l,g}^{q}$;
9   Update momentums of AMSGrad;
10   **for** $g \leftarrow 1$ **to** G_l **do**
11   Update $\boldsymbol{w}_{l,g}^{q+1}$ with Eq. (2.36);
12   Compute all values of Eq. (2.22);
13   **for** $j \leftarrow 1$ **to** n_l **do**
14   Update $\boldsymbol{B}_{l,g,j}^{q+1}$ with Eq. (2.38);
15   Update $\boldsymbol{\alpha}_{l,g}^{q+1}$ with Eq. (2.39);

Thus, the j-th row in $\boldsymbol{B}_{g}^{q+1}$ is updated by,

$$\boldsymbol{B}_{g,j}^{q+1} = \underset{\boldsymbol{B}_{g,j}}{\operatorname{argmin}} \|\boldsymbol{B}_{g,j}\boldsymbol{\alpha}_{g}^{q} - (w_{g,j}^{q} - g_{j}^{q}/H_{jj}^{q})\| \tag{2.38}$$

In addition, the speedup of Eq. (2.26) is changed accordingly as,

$$\boldsymbol{\alpha}_{g}^{q+1} = -((\boldsymbol{B}_{g}^{q+1})^{\mathrm{T}}\boldsymbol{H}^{q}\boldsymbol{B}_{g}^{q+1} + \lambda\mathbf{I})^{-1} \times ((\boldsymbol{B}_{g}^{q+1})^{\mathrm{T}}(\boldsymbol{g}^{q} - \boldsymbol{H}^{q}\boldsymbol{w}_{g}^{q})) \tag{2.39}$$

So far, the quantized weights are updated in a loss-aware manner as,

$$\hat{\boldsymbol{w}}_{g}^{q+1} = \boldsymbol{B}_{g}^{q+1}\boldsymbol{\alpha}_{g}^{q+1} \tag{2.40}$$

2.6.3.2 Ablation Results

Settings. To show the convergence performance of our Optimization Step, we compare Algorithm 2.4 with the above two baselines *STE with rec. error* and *STE with loss-aware* mentioned above. The three optimizers are used to train the networks quantized with a uniform bitwidth. We use AMSGrad[2] as the optimization framework for all optimizers and adopt a learning rate of 0.001.

[2]AMSGrad can also optimize full precision parameters.

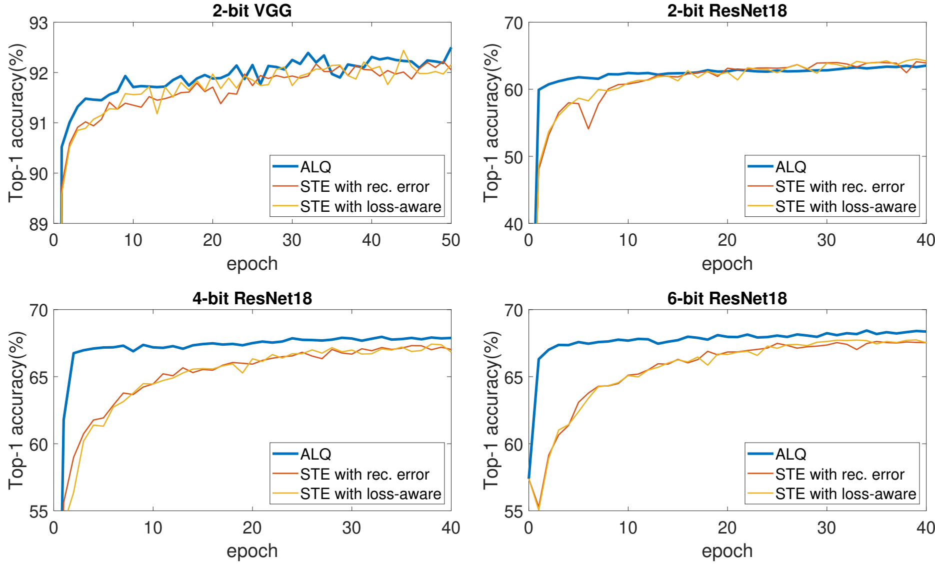

Figure 2.3: Validation accuracy trained with ALQ and other STE-based baselines along the training epochs.

Method	I_W	Top-1
Baseline VGGNet (uniform)	1	91.8%
ALQ VGGNet	**0.66**	**92.0%**
Baseline ResNet18 (uniform)	2	66.2%
ALQ ResNet18	**2.00**	**68.9%**

Table 2.1: Comparison between uniform bitwidth and adaptive bitwidth in ALQ.

Results. Figure 2.3 shows the Top-1 validation accuracy of different optimizers, with increasing epochs on uniform bitwidth MBNs. ALQ exhibits not only a more stable and faster convergence, but also a higher accuracy. The exception is 2-bit ResNet18. ALQ converges faster, but the validation accuracy trained with STE gradually exceeds ALQ after about 20 epochs. For training a large network with ≤ 2 bitwidth, the positive effect brought from the high precision trace may compensate certain negative effects caused by gradient approximation. In this case, keeping full precision parameters will help calibrate some aggressive steps of quantization, resulting in a slow oscillating convergence to a better local optimum. This also encourages us to add several epochs of STE based optimization (e.g., *STE with loss-aware*) after low bitwidth quantization to further regain the accuracy.

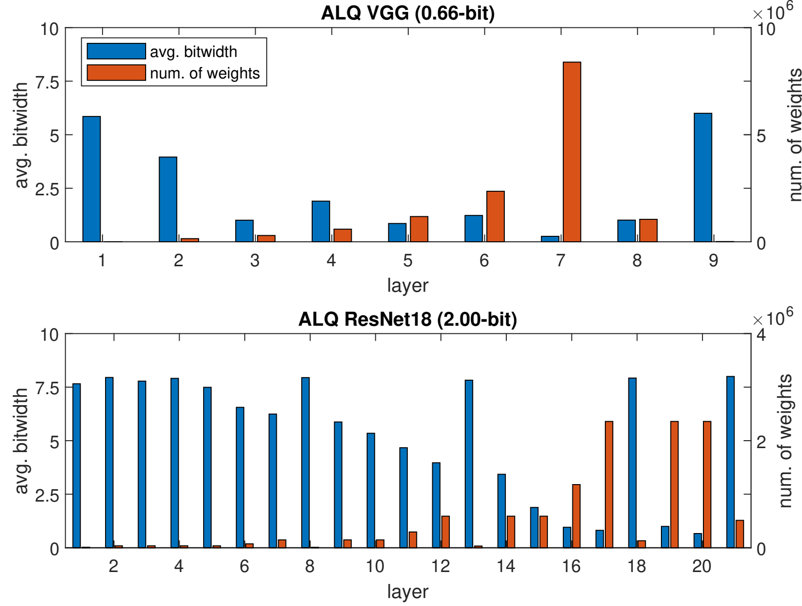

Figure 2.4: Distribution of the average bitwidth and the number of weights across layers.

2.6.4 Ablation Studies on Adaptive Bitwidth

Settings. This experiment demonstrates the performance of incrementally trained adaptive bitwidth in ALQ, i.e., our Pruning Step in Section 2.4.2. Uniform bitwidth quantization (an equal bitwidth allocation across all groups in all layers) is taken as the baseline. The baseline is trained with the same number of epochs as the sum of all epochs during the bitwidth reduction. Both ALQ and the baseline are trained with the same learning rate decay schedule.

Results. Table 2.1 shows that there is a large Top-1 accuracy gap between an adaptive bitwidth trained with ALQ and a uniform bitwidth. In addition to the overall average bitwidth, we also plot the distribution of the average bitwidth and the number of weights across layers (both models in Table 2.1) in Figure 2.4. Generally, the first several layers and the last layer are more sensitive to the loss, thus require a higher bitwidth. The shortcut layers in ResNet architecture (e.g., the 8-th, 13rd, 18-th layers in ResNet18) also need a higher bitwidth. We think this is due to the fact that the shortcut pass helps the information forward/backward propagate through the blocks. Since the average of adaptive bitwidth can have a decimal part, ALQ can achieve a compression ratio with a much higher resolution than a uniform bitwidth, which not only controls a more precise trade-off between storage and accuracy, but also benefits our incremental bitwidth reduction scheme.

It is worth noting that both the Optimization Step and the Pruning Step in ALQ follow the same metric, i.e., the loss increment modeled by

a quadratic function, allowing them to work in synergy. We replace the step of optimizing $\boldsymbol{B}_g$ in ALQ with an STE step (with the reconstruction forward, see in Section 2.6.3), and keep other steps unchanged in the pipeline. When the VGGNet model is reduced to an average bitwidth of 0.66-bit, the simple combination of an STE step with our Pruning Step can only reach 90.7% Top-1 accuracy, which is significantly worse than ALQ's 92.0%.

2.6.5 Comparison with State-of-the-Art Methods

2.6.5.1 Unstructured Pruning on MNIST

Settings. Since ALQ can be considered a structured pruning scheme (i.e., pruning in $\boldsymbol{\alpha}$ domain), we first compare ALQ with two widely used unstructured pruning schemes: Deep Compression (DC) [HMD16] and ADMM-Pruning (ADMM) [ZYZ⁺18], i.e., pruning in the original $\boldsymbol{w}$ domain. For a fair comparison, we implement a modified LeNet5 model as in [HMD16, ZYZ⁺18] on MNIST dataset [LC10] and compare the Top-1 prediction accuracy and the compression ratio.

The structures of each layer chosen for ALQ are kernelwise, kernelwise, subchannelwise(2), channelwise, respectively. After each pruning, the network is retrained to recover the accuracy degradation with 20 epochs of optimizing $\boldsymbol{B}_g$ and 10 epochs of optimizing $\boldsymbol{\alpha}_g$. The pruning ratio is 80%, and 4 times of Pruning Step are executed after initialization in the reported experiment in Table 2.2. After the last Pruning Step, we conduct 50 epochs of Optimizing Step to further increase the final accuracy (also applied in the following experiments of VGGNet and ResNet18/34).

ALQ can fast converge in the training. However, we observed that even after the convergence, the accuracy still continues increasing slowly along the training, which is similar to the behavior of STE-based optimizer. During the Optimization Step after each Pruning Step, as long as the training loss is almost converged with a few epochs, we can further proceed the next Pruning Step. We found that the final accuracy level is approximately the same whether we add plenty of epochs each time to slowly recover the accuracy to the original level or not. Thus, we choose a fixed modest number of retraining epochs after each Pruning Step to save the overall training time. In fact, this benefits from the feature of ALQ, which leverages the true gradient w.r.t. the loss to result in a fast and stable convergence. The final added 50 training epochs aim to further slowly regain the final accuracy level, where we use a gradually decayed learning rate, e.g., 10^{-4} decays with 0.98 in each epoch.

Note that the storage consumption only counts the weights, since the weights take the most majority of the storage (even after quantization) in comparison to others, e.g., bias, activation quantizer, batch normalization,

Method	Weights (CR)	Top-1
FP	1720KB (1×)	99.19%
DC [HMD16]	44.0KB (39×)	**99.26%**
ADMM [ZYZ+18]	24.2KB (71×)	99.20%
ALQ	**22.7KB** (**76**×)	99.12%

Table 2.2: Comparison with state-of-the-art unstructured pruning methods (LeNet5 on MNIST). "FP" denotes the full precision baseline. "CR" denotes the compression ratio related to full precision.

etc. The storage consumption of weights in ALQ includes the look-up-table for the resulting I_g in each group.

Results. ALQ shows the highest compression ratio (**76**×) while keeping acceptable Top-1 accuracy compared to the two other pruning methods (see Table 2.2). FP stands for full precision, and the weights in the original full precision LeNet5 consume 1720KB [HMD16]. CR denotes the compression ratio of static weight storage.

Note that both DC [HMD16] and ADMM [ZYZ+18] rely on sparse tensors, which need special libraries or hardwares for efficient execution [LKD+17]. Their operands (the shared quantized values) are still floating-point. Hence they hardly utilize bitwise operations for speedup. In contrast, ALQ achieves a higher compression ratio without sparse tensors, which is more suited for general off-the-shelf platforms.

The average bitwidth of ALQ is below 1.0-bit (1.0-bit corresponds to a compression ratio slightly below 32), indicating some groups are fully removed. In fact, this process leads to a new network architecture containing less output channels of each layer, and thus the corresponding input channels of the next layers can be safely removed. The original configuration $20-50-500-10$ is now $18-45-231-10$.

2.6.5.2 Binary Networks on CIFAR10

Settings. In this experiment, we compare the performance of ALQ with state-of-the-art binary networks [CBD15, RORF16, HYK17]. A binary network is an MBN with the lowest bitwidth, i.e., single-bit. Thus, the storage consumption of a binary network can be regarded as the lower bound of a (uniform) quantized network. We implement a small version of VGGNet from [SZ15] on CIFAR10 dataset [KNH09], as in many state-of-the-art binary networks [CBD15, HYK17, RORF16].

The structures of each layer chosen for ALQ are channelwise, pointwise, pointwise, pointwise, pointwise, pointwise, subchannelwise(16), subchannelwise(2), subchannelwise(2) respectively. After each pruning, the network is retrained to recover the accuracy degradation with 20 epochs of optimizing $\boldsymbol{B}_g$ and 10 epochs of optimizing $\boldsymbol{\alpha}_g$. The pruning

Method	I_W	Weights (CR)	Top-1
FP	32	56.09MB (1×)	92.8%
BC [CBD15]	1	1.75MB (32×)	90.1%
BWN [RORF16]*	1	1.82MB (31×)	90.1%
LAB [HYK17]	1	1.77MB (32×)	89.5%
AQ [KL18]	0.27	1.60MB (35×)	90.9%
ALQ	**0.66**	**1.29MB (43×)**	**92.0%**
ALQ	**0.40**	**0.82MB (68×)**	**90.9%**

*: both first and last layers are unquantized.

Table 2.3: Comparison with state-of-the-art binary networks (VGGNet on CIFAR10). "FP" denotes the full precision baseline. "CR" denotes the compression ratio related to full precision. I_w denotes the average bitwidth of weights.

ratio is 40%, and 5 or 6 times of Pruning Step are executed after initialization in the reported experiment (Table 2.3).

Results. Table 2.3 shows the performance comparison to popular binary networks. I_W stands for the quantization bitwidth for weights. Since ALQ has an adaptive quantization bitwidth, the reported bitwidth of ALQ is an average bitwidth of all weights.

ALQ allows to compress the network to under 1-bit, which remarkably reduces the storage and computation. ALQ achieves the smallest weight storage and the highest accuracy compared to all weights binarization methods BC [CBD15], BWN [RORF16], LAB [HYK17]. Similar to results on LeNet5, ALQ generates a new network architecture with fewer output channels per layer, which further reduces our models in Table 2.3 to 1.01MB (0.66-bit) or even 0.62MB (0.40-bit). The computation and the run-time memory can also decrease.

Furthermore, we also compare with AQ [KL18], the state-of-the-art adaptive fixed-point quantizer. It assigns a different bitwidth for each parameter based on its sensitivity, and also realizes a pruning for 0-bit parameters. Our ALQ not only consumes less storage, but also acquires a higher accuracy than AQ [KL18]. Besides, the non-standard quantization bitwidth in AQ cannot efficiently run on general hardware due to the irregularity [KL18], which is not the case for ALQ.

In order to demonstrate the affects from different steps in ALQ, we plot the training loss curve of quantizing VGGNet on CIFAR10 with ALQ. Different steps in ALQ are marked with different colors, see Figure 2.5. The results show that (*i*) Initialization Step does not bring any performance drop; (*ii*) Optimization Step can fast converge in a few epochs and may recover the performance drop from Pruning Step as long as the average bitwidth is not extremely low.

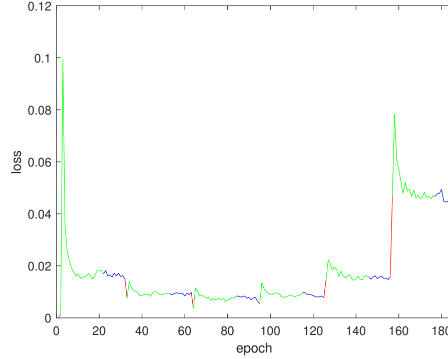

Figure 2.5: The training loss curves of different steps in ALQ (VGGNet on CIFAR10). 'Magenta' stands for Initialization Step; 'Green' stands for optimizing $\boldsymbol{B}_g$ with speedup; 'Blue' stands for optimizing $\boldsymbol{\alpha}_g$; 'Red' stands for Pruning Step. Please see this figure in color.

2.6.5.3 MBNs on ImageNet

Settings. We quantize both the weights and the activations of ResNet18/34 [HZRS16] with a low bitwidth ($\leq$ 2-bit) on ImageNet dataset [RDS+15], and compare our results with state-of-the-art multi-bit networks. The results for the full precision version are provided by Pytorch [PGC+17]. We choose ResNet18, as it is a popular model on ImageNet used in the previous quantization schemes. ResNet34 is a deeper network used more in recent quantization papers.

The structures of each layer chosen for ALQ are all pointwise except for the first layer (kernelwise) and the last layer (subchannelwise(2)). After each pruning, the network is retrained to recover the accuracy degradation with 10 epochs of optimizing $\boldsymbol{B}_g$ and 5 epochs of optimizing $\boldsymbol{\alpha}_g$. The pruning ratio is 15%.

Results. Table 2.4 shows that ALQ obtains the highest accuracy with the smallest network size on ResNet18/34, in comparison with other weight and weight+activation quantization approaches. I_W and I_A are the quantization bitwidth for weights and activations respectively.

Several schemes (marked with *) are not able to quantize the first and last layers, since quantizing both layers as other layers will cause a huge accuracy degradation [WSL+18, MM18]. It is worth noting that the first and last layers with floating-point values occupy 2.09MB storage in ResNet18/34, which is still a significant storage consumption on such a low-bit network. We can simply observe this enormous difference between TWN [LZL16] and LQ-Net [ZYYH18] in Table 2.4 for example. The evolved floating-point computations in both layers can hardly be accelerated with bitwise operations either.

For reported ALQ models in Table 2.4, as several layers have already been pruned to an average bitwidth below 1.0-bit (e.g., see in Figure 2.4),

Method	I_W/I_A	Weights	Top-1
	ResNet18		
FP [PGC+17]	32/32	46.72MB	69.8%
TWN [LZL16]	2/32	2.97MB	61.8%
LR [SLF18]	2/32	4.84MB	63.5%
LQ [ZYYH18]*	2/32	4.91MB	68.0%
QIL [JSL+19]*	2/32	4.88MB	68.1%
INQ [ZYG+17]	3/32	4.38MB	68.1%
ABC [LZP17]	5/32	7.41MB	68.3%
ALQ	**2.00/32**	**3.44MB**	**68.9%**
ALQ^e	**2.00/32**	**3.44MB**	**70.0%**
BWN [RORF16]*	1/32	3.50MB	60.8%
LR [SLF18]*	1/32	3.48MB	59.9%
DSQ [GLJ+19]*	1/32	3.48MB	63.7%
ALQ	**1.01/32**	**1.77MB**	**65.6%**
ALQ^e	**1.01/32**	**1.77MB**	**67.7%**
LQ [ZYYH18]*	2/2	4.91MB	64.9%
PACT [CWV+18]*	2/2	4.88MB	64.4%
QIL [JSL+19]*	2/2	4.88MB	65.7%
DSQ [GLJ+19]*	2/2	4.88MB	65.2%
GroupNet [ZST+19]*	4/1	7.67MB	66.3%
RQ [LRB+19]	4/4	5.93MB	62.5%
ABC [LZP17]	5/5	7.41MB	65.0%
ALQ	**2.00/2**	**3.44MB**	**66.4%**
SYQ [FFBL18]*	1/8	3.48MB	62.9%
LQ [ZYYH18]*	1/2	3.50MB	62.6%
PACT [CWV+18]*	1/2	3.48MB	62.9%
ALQ	**1.01/2**	**1.77MB**	**63.2%**
	ResNet34		
FP [PGC+17]	32/32	87.12MB	73.3%
ALQ^e	**2.00/32**	**6.37MB**	**73.6%**
ALQ^e	**1.00/32**	**3.29MB**	**72.5%**
LQ [ZYYH18]*	2/2	7.47MB	69.8%
QIL [JSL+19]*	2/2	7.40MB	70.6%
DSQ [GLJ+19]*	2/2	7.40MB	70.0%
GroupNet [ZST+19]*	5/1	12.71MB	70.5%
ABC [LZP17]	5/5	13.80MB	68.4%
ALQ	**2.00/2**	**6.37MB**	**71.0%**
TBN [WSL+18]*	1/2	4.78MB	58.2%
LQ [ZYYH18]*	1/2	4.78MB	66.6%
ALQ	**1.00/2**	**3.29MB**	**67.4%**

*: both first and last layers are unquantized.
^{e}: adding extra epochs of *STE with loss-aware* in the end.

Table 2.4: Comparison with state-of-the-art quantized networks (ResNet18/34 on ImageNet). "FP" denotes the full precision baseline. "CR" denotes the compression ratio related to full precision. I_w denotes the average bitwidth of weights. I_A denotes the bitwidth of activations.

we add extra 50 epochs of our *STE with loss-aware* in the end as discussed in Section 2.6.3. The learning rate is 10^{-4}, and gradually decays with 0.98 per epoch. The final accuracy is further boosted by around 1% ~ 2%, see the results marked with ^{e}. With such an extremely low bitwidth, maintained full precision weights help to calibrate some aggressive steps of quantization, which slowly converges to a local optimum with a higher accuracy for a large network. Recall that maintaining full precision parameters means STE is required to approximate the gradients, since the true-gradients only relate to the quantized parameters used in the forward propagation. However, for the quantization bitwidth higher than two (> 2.0-bit), the quantizer can take smooth steps, and the gradient approximation due to STE damages the training inevitably. Thus in this case, the true-gradient optimizer, i.e., Algorithm 2.4, can converge to a better local optimum, faster and more stable.

ALQ can quantize ResNet18/34 with 2.00-bit (across all layers) *without any accuracy loss*. To the best of our knowledge, this is the first time that the 2-bit weight-quantized ResNet18/34 can achieve the accuracy level of its full precision version, even if some prior schemes keep the first and last layers unquantized. These results further demonstrate the high-performance of the pipeline in ALQ.

2.7 Summary

In this chapter, we propose ALQ, an adaptive loss-aware trained quantizer for multi-bit networks. ALQ enables efficient inference on edge devices. ALQ tries to reduce the redundancy on the quantization bitwidth to achieve both storage efficiency and computation efficiency. Unlike prior quantized networks that (*i*) often assign an empirical global bitwidth across layers, (*ii*) train the quantizer by minimizing the reconstruction error to the full precision weights, ALQ (*i*) allocates an adaptive bitwidth to different weights w.r.t. the loss, (*ii*) optimizes the multi-bit quantizer by minimizing the loss as well. The adaptive bitwidth assignment and the direct optimization objective allow ALQ to find and remove more redundant bitwidth, thus achieving a better trade-off between the resource constraints and the model accuracy. The main contributions are summarized as follows,

- ALQ introduces a multi-bit network with adaptive quantization bitwidth across different groups of weights. Such an adaptive multi-bit network not only achieves a high compression ratio on static weight storage by only assigning a high bitwidth to loss-critical weights, but also replaces the expensive floating-point operations with a single set of cheaper operations from `xnor`, `popcount` and accumulations.

- ALQ trains the multi-bit quantized weights by directly minimizing the loss function. This loss-aware quantization results in a faster convergence rate as well as a higher final accuracy than state-of-the-art STE-based quantization training that minimizes the reconstruction error.
- Via entirely pruned groups (i.e., 0-bit weights in some groups), ALQ enables extremely low-bit networks with an average bitwidth below 1-bit yet with *dense tensor form*. It breaks the traditional lower bound of the quantized network, i.e., binary network, thus providing more visions and possibilities for the network compression. Experiments on CIFAR10 show that ALQ can compress VGGNet to an average bitwidth of 0.4-bit, while yielding a higher accuracy than other binary networks [RORF16, CBD15].
- ALQ is the first loss-aware quantization scheme for multi-bit networks and eliminates the need for approximating gradients and retaining full precision weights. ALQ is also able to quantize the first and last layers without incurring a notable accuracy loss.

This chapter studied how to compress the network for efficient inference given the fixed on-device resource constraints. In the next chapter, we will further study how to adapt the network on edge devices when the resource constraint is varied along the lifetime. Although we may deploy multiple ALQ-quantized multi-bit networks with different average bitwidth to execute under different resource budgets, this naive solution can only result in a subpar performance, as it requires several times more storage consumption in comparison to a single (multi-bit) network. However, the solution proposed in the next chapter can meet the varying resource constraints without incurring extra storage overhead.

3

Adaptation on Edge Devices

In Chapter 2, we explored how DNNs can be compressed while respecting resource constraints. However, the resource constraints on the target edge devices may change dynamically during runtime. To maximize model accuracy during on-device inference, in this chapter we deploy a DNN that can adapt to the different resource constraints on the edge device.

Main Resource Constraints. The different resource constraints during on-device inference may be due to for example the available battery power or the allowed inference time. Similar to Chapter 2, we mainly adopt two widely used proxies to quantify the (varying) resource consumption, (*i*) *the storage of weights*, which affects the amount of memory fetching and static memory consumption, and (*ii*) *the number of operations for inference*, which is relevant to the computing energy and the inference latency.

Principles. Faced with the varying resource constraints on edge devices, existing synthesis methods require either deploying multiple individual networks with different resource demands or sampling sub-networks along structured dimensions, which leads to poor performance. However, we propose to sample sub-networks from the backbone network through row-based unstructured sparsity, and propose a novel compressed sparse row (CSR) format for efficient sparse inference. Our synthesis methods reduce redundancy among multiple sub-networks through weight sharing and architecture sharing, resulting in storage efficiency and re-configuration efficiency.

The contents of this chapter are established mainly based on the paper "DRESS: Dynamic REal-time Sparse Subnets" that is published on Efficient Deep Learning for Computer Vision CVPRWorkshop (ECV), 2022. This work is collaborated with the colleagues at Meta Reality Labs Research.

This work was done when Zhongnan Qu was a research intern at Meta.

3.1 Introduction

Extensive synthesis works [HMD16, RORF16, SMNP21, OSKY21] have proposed to first compress a pretrained model according to the given resource constraints, and then compile the compressed model to deploy on target edge devices. However, the time constraints of many practical embedded systems may dynamically change at run-time. For example, when detecting hand positions on a workbench in real-time, the allowed inference time varies during the entire manipulation. In comparison to general movement, engineers will slow down the hand movement if performing some critical tasks, e.g., grasping objects, which gives DNNs a longer execution time when requiring higher perceptive precision. Some similar scenarios also include autonomous vehicles' reaction time on city roads and highways due to different operating speeds. On the other hand, the available resources on the target edge device may also vary along the lifetime, e.g., the battery energy, the allocatable RAM. All considerations mentioned above indicate that the deployed inference model should maintain a dynamic capacity, such that the model can be adapted and executed under different resource constraints.

Challenges. Making DNNs adaptable on resource-constrained devices is even more challenging. Existing synthesis methods either fail to compile DNNs that can adapt to varying resource constraints, or result in subpar performance. Traditional compression techniques, e.g., pruning, quantization, only result in a static inference model. Although the compressed model is mapped onto target devices, it can not meet various resource requirements. As an alternative, we may compile for example multiple networks with different sparsity levels, which however need several times more storage consumption in comparison to a single sparse network. Recent works [YYX+19, CGW+20] show that sub-networks from a pretrained backbone network can reach a decent performance compared to the sub-networks trained individually from scratch. Nevertheless, they only sample sub-network architectures along hand-crafted structured dimensions, e.g., width, kernel size, which leads to sub-optimal results. Switching among multiple compiled architectures on edge devices may also cause extra re-configuration overhead.

In this chapter, we propose a novel synthesis technique, **D**ynamic **RE**al-time **S**parse **S**ubnets (DRESS). DRESS samples sub-networks from the backbone network through row-based unstructured sparsity, while ensuring that nonzero weights of the higher sparsity networks are reused by the lower sparsity networks. This way, the overall memory consumption is bounded by the network with the lowest sparsity and does not depend on the number of networks, resulting in *memory efficiency*; all sparse sub-networks leverage the same architecture as the backbone network, leading to *re-configuration efficiency*. The sub-network with a higher sparsity (i.e., fewer nonzero weights) needs a smaller amount

of on-device memory fetching and fewer FLOPs, thus shall be adopted to inference under more severe resource constraints, e.g., lower energy budget, limited inference time. Specifically, we (*i*) sample weights w.r.t. their magnitudes in a row-based unstructured manner; (*ii*) train all sampled sparse sub-networks with weighted loss in parallel; (*iii*) further fine-tune batch normalization for each sub-network individually.

3.2 Related Work

3.2.1 Network Compression & Deployment

Network compression focuses on trimming down the DNN model size with negligible performance degradation. Commonly used compression techniques can be divided into three categories, (*i*) designing efficient network architectures manually [HZC+17, SHZ+18] or automatically using neural architecture search [CGW+20, YH19a, YJL+20, MKH21]; (*ii*) quantizing weight values into lower bitwidth to use cheaper operations and reduce the storage consumption [RORF16, YLDM19, SMNP21]; (*iii*) structured [LWS+20, LMW+20, WWSH20]/unstructured [HMD16, RFC20, EGM+21, PIVA21, OSKY21, AAH+20] pruning unimportant weights as zeros to reduce the number of operations and the number of nonzero weights. The compressed model is further optimized by some compilation libraries in order to speed up inference on target edge platforms, e.g., CMSIS-NN for Arm Cortex-M CPUs [LSC18], XNNPACK for Arm64 and ArmV7 CPUs [Goo19], Vela for Ethos-U NPU [Vel20]. Note that the compiled model often only supports a static computation graph due to the limited resources on edge devices [LSC18, Goo19, Vel20]. In this chapter, we focus on unstructured pruning among others, since (*i*) it often yields a high compression ratio [RFC20]; (*ii*) the networks with different unstructured sparsity may share the same network architecture, i.e., the same compiled computation graph. Furthermore, some recent libraries e.g., XNNPACK include fast kernels for sparse matrix-dense matrix multiplication, which enables sparse DNN acceleration on edge platforms [EDGS20, LLM+20].

3.2.2 Dynamic Networks

Dynamic networks aim at a better trade-off between inference accuracy and average inference efficiency, by adapting network structures or network parameters according to the inputs during inference [HHS+21]. Among them, some works propose allocating less computation on those canonical data samples, through skipping layers [HCL+18], pruning unimportant channels [LWW+21, WWSH20], selecting a subset of salience pixels [VT20]. Although these sample-wise dynamic networks may achieve a smaller inference cost averaged over different samples, they

cannot adapt the model to fit different resource budgets. In addition, to achieve data-dependent adaptiveness, they often bring additional computation burden, e.g., hard attention, gater, etc. [HHS+21]

3.2.3 Anytime Networks (Sub-networks)

Anytime networks refer to the network whose sub-networks can be executed separately with less resource consumption while achieving a satisfactory performance. DRESS falls into the same scope of anytime networks. MSDNet [HCL+18] densely connects multiple convolutional layers in both depth direction and scale direction, such that the computation can be saved by early-exiting from a certain layer. [HDHB17] introduces an adaptive weighted loss to optimize the network with various depths. Slimmable networks [YYX+19, YH19b] propose to train a single model which supports multiple width multipliers (i.e., number of channels) in each layer. [CGW+20] suggests to search network architectures with different kernel size, depth, and width, in a single pretrained once-for-all network. Subflow [LN20] executes only a sub-graph of the full DNN by activating partial neurons given the varying time constraints. State-of-the-art anytime networks always sample sub-networks from the backbone network along hand-crafted structured dimensions, e.g., depth, width, kernel size, neuron. As zero weights have no effects on the calculation, anytime networks actually perform structured pruning on the backbone network, which could yield a subpar performance in comparison to unstructured sampling. In addition, resulted sub-networks often have different network architectures, e.g., different kernel sizes. When adopting these sub-networks on edge devices, the re-configuration of the computation graph may bring extra overhead. On the other hand, SP-Net [GZRD20] suggests adjusting the quantization bitwidth on demand, which however requires specialized integer arithmetic units for efficient computing.

3.2.4 Weight Sharing

Sub-networks rely on weight reusing (sharing). Except for sub-networks, weight sharing among different networks is also widely used in other settings. Multi-task learning [SK18] reuses partial weights of networks performing diverse tasks to reduce memory consumption. However, these methods are inapplicable in our scenarios, which target a single task with varying resource constraints. Neural architecture search (NAS) applied in [CGW+20, YH19a, YJL+20, CZX21] maintains a single set of shared weights (also known as *supernet*) when searching different architectures to reduce the training effort. Note that NAS is orthogonal to our method since the searched optimal architecture can be used as our backbone network.

3.3 Dynamic Real-time Sparse Subnets

3.3.1 Problem Definition

We aim at sampling multiple subnets from a backbone network. The backbone network is a conventional DNN consisting of L convolutional (conv) layers or fully connected (fc) layers. These subnets have different resource demands and thus can be adapted to different resource availabilities. Since the subnets have the same architecture as the backbone network, they can share a single compiled architecture to achieve re-configuration efficiency; the nonzero weights of the subnet with a higher sparsity are reused by the subnet with a lower sparsity to achieve memory efficiency. This way, we only need to store a table for the lowest sparsity network, including its nonzero weights sorted w.r.t. importance and corresponding indices. Accordingly, the other networks can be build from the top important weights through a pre-defined sparsity together with the compiled architecture. Assume that we sample K sparse subnets, then the preliminary problem is defined as,

$$\min_{\boldsymbol{w},\boldsymbol{m}_k} \quad \ell(\boldsymbol{w} \odot \boldsymbol{m}_k) \quad \forall k \in 1, ..., K \tag{3.1}$$

$$\text{s.t.} \quad ||\boldsymbol{m}_k||_0 = (1 - s_k) \cdot I \quad \forall k \in 1, ..., K \tag{3.2}$$

$$\boldsymbol{m}_i \odot \boldsymbol{m}_j = \boldsymbol{m}_j \quad \forall 1 \le i < j \le K \tag{3.3}$$

where $\boldsymbol{w}$ stands for the weights of the (dense) backbone network; $\boldsymbol{m}_k$ stands for the binary mask of the k-th subnet; s_k stands for the pre-defined sparsity level. $\ell(.)$ denotes the loss function, $||.||_0$ denotes the L0-norm, $\odot$ denotes the element-wise multiplication. Note that $\boldsymbol{w} \in \mathbb{R}^I$, $\boldsymbol{m}_k \in \{0, 1\}^I$, where I is the total number of weights. Clearly, we have $0 < s_1 < s_2 < ... < s_K \le 1$, i.e., the first subnet bounds the overall static storage consumption. $\boldsymbol{w}_k$ is denoted as nonzero weights of the k-th sparse subnet, i.e., $\boldsymbol{w}_k = \boldsymbol{w} \odot \boldsymbol{m}_k$.

In the following sections, we detail how to solve Eq. (3.1)-Eq. (3.3) in our DRESS synthesis approach. DRESS consists of three training stages as discussed below. The overall pipeline is shown in Algorithm 3.1.

- Dense Pre-Training: The backbone network is trained from scratch with a traditional optimizer to provide a good initialization for the following sparse training.
- DRESS Training: The multiple sparse subnets are sampled from the backbone network (Section 3.3.2, Section 3.3.4) and are jointly trained in parallel with weighted loss (Section 3.3.3).
- Post-Training on Batch Normalization: Batch normalization (BN) layers are further optimized individually for each subnet to better reveal the statistical information (Section 3.3.5).

Algorithm 3.1: Dynamic REal-time Sparse Subnets

Input: Initial random weights $\boldsymbol{w}$, training dataset $\mathcal{D}_{\text{tr}}$, validation dataset $\mathcal{D}_{\text{val}}$, overall sparsity $\{s_k\}_{k=1}^{K}$, normalized loss weights $\{\pi_k\}_{k=1}^{K}$

Output: Optimized weights $\boldsymbol{w}$, binary masks $\{\boldsymbol{m}_k\}_{k=1}^{K}$

```
/* Dense pre-training */
```
1 Train dense network $\boldsymbol{w}$ with traditional optimizer;
```
/* DRESS training */
```
2 Allocate layer-wise sparsity $\{s_{k,l}\}_{l=1}^{L}$ for each s_k;
3 Initiate $\boldsymbol{w}^0 = \boldsymbol{w}$;
4 **for** $q \leftarrow 1$ **to** Q **do**
    // The q-th training iteration
5   Fetch mini-batch from $\mathcal{D}_{\text{tr}}$;
6   Initialize backbone-net gradient $\boldsymbol{g}(\boldsymbol{w}^{q-1}) = \boldsymbol{0}$;
7   **for** $k \leftarrow 1$ **to** K **do**
8   Sample a subnet with sparsity $\{s_{k,l}\}_{l=1}^{L}$ and get its mask $\boldsymbol{m}_k$;
9   Get sparse subnet $\boldsymbol{w}_k^{q-1} = \boldsymbol{w}^{q-1} \odot \boldsymbol{m}_k$;
10   Back-propagate subnet gradient $\boldsymbol{g}(\boldsymbol{w}_k^{q-1}) = \pi_k \cdot \frac{\partial \ell(\boldsymbol{w}_k^{q-1})}{\partial \boldsymbol{w}_k^{q-1}}$;
11   Accumulate backbone-net gradient $\boldsymbol{g}(\boldsymbol{w}^{q-1}) = \boldsymbol{g}(\boldsymbol{w}^{q-1}) + \boldsymbol{g}(\boldsymbol{w}_k^{q-1}) \odot \boldsymbol{m}_k$;
12   Compute optimization step $\Delta \boldsymbol{w}^q$ with $\boldsymbol{g}(\boldsymbol{w}^{q-1})$;
13   Update $\boldsymbol{w}^q = \boldsymbol{w}^{q-1} + \Delta \boldsymbol{w}^q$;
14   **if** *Higher average epoch accuracy on* $\mathcal{D}_{val}$ **then**
15   Save $\boldsymbol{w} = \boldsymbol{w}^q$ and $\{\boldsymbol{m}_k\}_{k=1}^{K}$;
16   **else**
17   Re-allocate layer-wise sparsity $\{s_{k,l}\}_{l=1}^{L}$ for each s_k;
```
/* Post-training on batch normalization (BN) */
```
18 **for** $k \leftarrow 1$ **to** K **do**
19   Load $\boldsymbol{w}$ and $\boldsymbol{m}_k$;
20   Fine-tune BN layers of subnet $\boldsymbol{w} \odot \boldsymbol{m}_k$;

3.3.2 How to Sample Sparse Subnets

Unlike traditional anytime networks that sample subnets along structured dimensions, DRESS samples subnets weight-wise which extremely enlarges the sampling space. Recall that we introduce K binary masks $\boldsymbol{m}_{1:K}$ to indicate if the weight is selected in each subnet. The naive approach could be iterative sampling K subnets where each iteration exhaustively searches the best-performed subnet inside the current subnet. Yet this naive approach can be either conducted rarely or infeasible due to the high complexity.

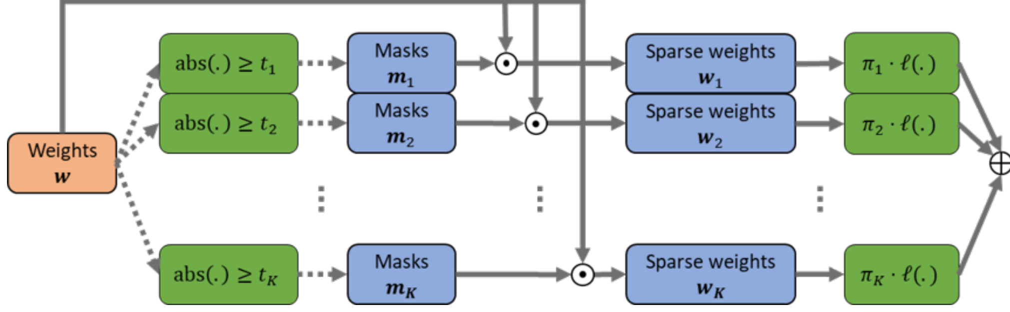

Figure 3.1: The computation graph used in parallel training multiple subnets. The orange block stand for the leaf variable to be optimized; the blue block stand for the intermediate variable; the green block stand for the computation unit.

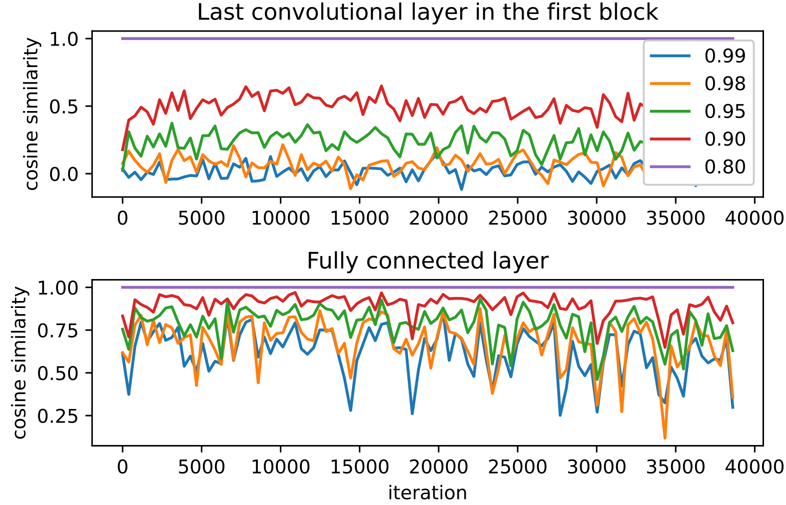

Figure 3.2: The cosine similarity between the loss gradients of 5 subnets (with sparsity 0.8,0.9,0.95,0.98,0.99) and that of the lowest sparsity subnet (with sparsity 0.8) along the training iterations. We show two typical layers in ResNet20, the last `conv` layer of the first block and the `fc` layer

To reduce the complexity, we propose to greedily sample the subnet based on the importance of weights. Following the prior pruning works [HMD16, FC19, RFC20], the importance is measured by the weight magnitudes. Given an overall sparsity level s_k, the $(1-s_k) \cdot I$ weights with the largest magnitudes will be sampled and are used to build the subnet. However, it is still infeasible to conduct such a global sorting across all layers in each training iteration. Instead, the weights are only sorted and sampled inside each layer according to a layer-wise sparsity $s_{k,l}$, where l denotes the layer index. The global sorting, also the (re-)allocation of layer-wise sparsity, is conducted only if the average accuracy of subnets does not improve anymore, see in Algorithm 3.1. During (re-)allocation,

the weights of all `conv` and `fc` layers with the largest magnitudes will be selected in sequence until reaching the overall sparsity s_k, and the layer-wise sparsity can be then calculated accordingly.

Note that the (dense) backbone network is maintained and continuously updated when training sampled subnets. In comparison to traditional pruning, where only the nonzero weights at fixed locations are fine-tuned, our sparse subnets are re-sampled from the backbone network in each training iteration. This flexible mechanism is crucial to acquire multiple high-performed subnets, see the ablation results in Section 3.4.2.2.

3.3.3 How to Optimize Subnets

With sampled binary masks, we can now build and train the subnets. Our concept for optimizing subnets is based on the key insight: *in comparison to iterative training of subnets in progressively decreased/increased sparsity, parallel training allows multiple subnets to be sampled and optimized jointly thus yields higher performance.*

Experimental results in Section 3.4.2.3 show that parallel training multiple subnets always yields a higher prediction accuracy than iterative training. As a possible explanation, the optimizer may be stuck into a bad local optimum around the previous subnet during iterative training, whereas parallel training searches multiple subnets jointly. We thus adopt parallel training in DRESS as in Algorithm 3.1.

In parallel training, Eq. (3.1) can be re-written as,

$$\min_{\boldsymbol{w}, \boldsymbol{m}_k} \sum_{k=1}^{K} \pi_k \cdot \ell(\boldsymbol{w} \odot \boldsymbol{m}_k) \tag{3.4}$$

where π_k is the normalized scale ($\sum_{k=1}^{K} \pi_k = 1$) used to weight K loss items, which will be discussed later. In fact, this process determines a threshold t_k for the k-th sparsity level, the mask value $m_{k,i} = 1$ if $\text{abs}(w_i) \geq t_k$, otherwise 0, $\forall i \in 1, ..., I$. t_k is set to the value such that $(1 - s_k)$ of weights have a larger absolute value than t_k. Clearly, we have $t_1 < t_2 < ... < t_K$ due to the constraints of Eq. (3.2)-Eq. (3.3). In each training iteration, we sample K sparse subnets $\boldsymbol{w}_{1:K}$ from the backbone network $\boldsymbol{w}$. Each subnet's loss function is weighted by π_k and summed together. This weighted sum is to be minimized and thus is used to compute the gradients of $\boldsymbol{w}$, see the optimization graph in Figure 3.1.

When parallel training multiple subnets, the gradients of the backbone network is accumulated by the (weighted) loss gradients back-propagated through all K sparse subnets, as,

$$\boldsymbol{g}(\boldsymbol{w}) = \sum_{k=1}^{K} \pi_k \frac{\partial \ell(\boldsymbol{w}_k)}{\partial \boldsymbol{w}_k} \odot \boldsymbol{m}_k \tag{3.5}$$

We parallelly train 5 subnets of ResNet20 [HZRS16] with sparsity $s_{1:5} = 0.8, 0.9, 0.95, 0.98, 0.99$ on CIFAR10, and let the 5 loss items weighted equally, i.e., $\pi_{1:5} = 0.2$. We plot the cosine similarity between the loss gradients of 5 subnets (i.e., $(\partial\ell(\boldsymbol{w}_k)/\partial\boldsymbol{w}_k)\odot\boldsymbol{m}_k$ with $k = 1, ..., 5$) and that of the lowest sparsity subnet (i.e., $(\partial\ell(\boldsymbol{w}_1)/\partial\boldsymbol{w}_1)\odot\boldsymbol{m}_1$) along with the training iterations, see in Figure 3.2. It shows that the loss gradients of different subnets are always positively correlated with each other. The results also verify that multiple subnets are jointly trained towards the optimal point in the loss landscape. Because of Eq. (3.3), the nonzero weights in higher sparsity subnets (e.g., $\boldsymbol{w}_5$) are also selected by other subnets, which means these weights are optimized with a larger step size than other weights. To balance the step size, the subnet with a higher sparsity (fewer trainable weights) shall be assigned to a smaller weight π_k on its loss. We propose to weight the loss items by the ratio of trainable weights (i.e., $1 - s_k$) together with a correction factor γ.

$$\alpha_k = (1 - s_k)^\gamma \tag{3.6}$$

$$\pi_k = \alpha_k / \sum_{k=1}^{K} \alpha_k \tag{3.7}$$

The normalized weights π_k provide control over the significance of subnets in parallel training. $\gamma = 0$ means weighting loss items equally. Experimentally, we find that $\gamma \in [0.5, 1]$ often yields a satisfactory performance, see in Section 3.4.2.4.

3.3.4 How to Store Subnets

To realize efficient storage and computation, current compilation libraries often encode sparse tensors in compressed sparse row (CSR) format (or some similar formats e.g., Block-CSR) [Goo19, EDGS20, LLM+20]. An example CSR format of sparse tensor for a conv layer is depicted in Figure 3.3. When adopting traditional CSR format to store subnets generated by DRESS, we need to store (*i*) the subnet with the lowest sparsity including the row indices, the column indices, and the nonzero values, (*ii*) K threshold values $t_{1:K}$. However, when selecting the k-th subnet for inference, all nonzero weights need to be fetched and compared with t_k. Although we may build specialized indexing for every subnet individually, it in turn results in more memory cost depending on the number of subnets.

To achieve an efficient inference on different sparse subnets while without extra memory overhead, we adopt a *row-based unstructured* sparsity (a.k.a. *N:M fine-grained structure sparsity* [ZMZ+21, HCI+21, SZS+21]), where different rows leverage the same sparsity level. We denote N as the row size, also the number of weights in each row. Especially, for sparsity s_k, all rows have exactly $(1-s_k)\cdot N$ nonzero weights.

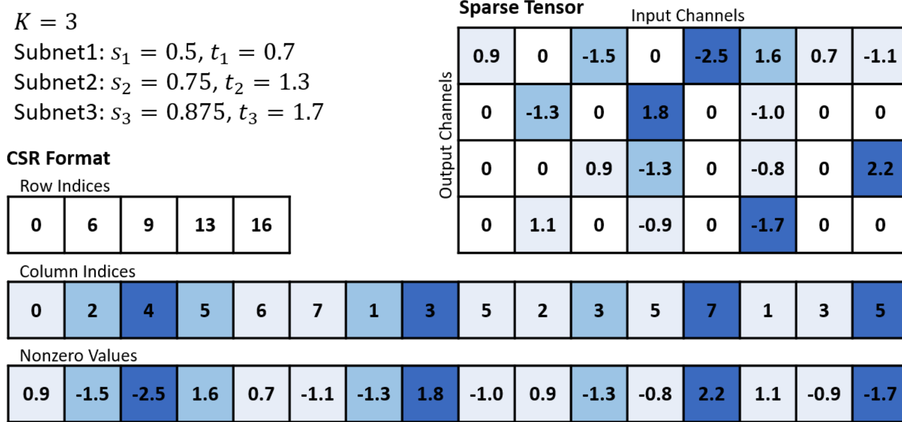

Figure 3.3: Traditional CSR format of unstructured sparse tensor. The example weight tensor is from a 1×1 **conv** layer with 8 input channels and 4 output channels. The sparse tensor has a row dimension along each output channel, i.e., each **conv** filter. There are 3 sparse subnets with sparsity 0.5, 0.75, and 0.875. Each subnet corresponds to a threshold value of 0.7, 1.3, and 1.7, respectively.

In comparison to conventional unstructured sparsity, this kind of sparsity can also be accelerated with sparse tensor cores of Nvidia A100 GPUs [MLP+21] for both training and inference, and thus becomes prevailing recently. *To our best knowledge, this is the first work that builds multiple subnetworks via fine-grained structure of weight sharing.* The column indices are stored according to the descending order of the importance (also weight magnitudes) in a two-dimensional table. The nonzero weights are stored in another table with the same order as the column indices. This DRESS CSR format needs to store (*i*) the subnet with the lowest sparsity including the table of the column indices and the table of nonzero weights, (*ii*) K integers $\{(1 - s_k) \cdot N\}_{k=1}^{K}$. It has overall a similar memory cost as traditional CSR format. When adopting the k-th subnet, we fetch the first $(1 - s_k) \cdot N$ columns from both tables as shown in Figure 3.4. The row indices can be built with $(1 - s_k) \cdot N$. Note that all fetched subnets follow the CSR format and utilize the same compiled network architecture, which allows us to leverage available libraries to achieve a fast inference without reconfiguration overhead.

To obtain DRESS CSR format, the sampling process needs to be adjusted accordingly. Especially, for layer l, we first pre-define a row size N_l and reshape the weight tensor into rows. For example, each **conv** filter corresponds to one row in accordance with the compilation libraries [Goo19, EDGS20]. When sampling a subnet in layer l, we conduct unstructured sampling in each row individually, while each row has the same sparsity $s_{k,l}$ as in Figure 3.4. Note that when N_l equals the total number of weights, i.e., a single row in DRESS CSR, it turns

Column Indices

4	5	2	7
3	1	5	7
7	3	2	5
5	1	3	4

Nonzero Values

-2.5	1.6	-1.5	-1.1
1.8	-1.3	-1.0	-0.6
2.2	-1.3	0.9	-0.8
-1.7	1.1	-0.9	0.3

Sparse Tensor (Input Channels × Output Channels)

0	0	-1.5	0	-2.5	1.6	0	-1.1
0	-1.3	0	1.8	0	-1.0	0	-0.6
0	0	0.9	-1.3	0	-0.8	0	2.2
0	1.1	0	-0.9	0.3	-1.7	0	0

Figure 3.4: DRESS CSR format of row-based unstructured sparse tensor. The example weight tensor has the same size as Figure 3.3. Each row has 8 weights in total, also the row size $N = 8$. There are 3 sparse subnets with sparsity 0.5, 0.75, and 0.875, as Figure 3.3. Each subnet has 4, 2, and 1 nonzero weights per row, respectively.

Algorithm 3.2: Row-based unstructured sampling

Input: Weight tensor $\boldsymbol{w} \in \mathbb{R}^{G \times N}$, row size N, sparsity $\{s_k\}_{k=1}^{K}$
Output: Binary masks $\{\boldsymbol{m}_k\}_{k=1}^{K}$

1 **for** $k \leftarrow 1$ **to** K **do**
2   Initiate binary mask $\boldsymbol{m}_k = 0^{G \times N}$;
3   Get the number of nonzero weights per row $N_k^{\text{nz}} = N \cdot (1 - s_k)$;
4 **for** $g \leftarrow 1$ **to** G **do**
5   Sort the weight magnitudes of row $\boldsymbol{w}_{g,:}$ in descending order;
6   **for** $k \leftarrow 1$ **to** K **do**
7   Set the mask values of $\boldsymbol{m}_{k,g,:}$ as 1 for Top-N_k^{nz} indices;

back into the original unstructured sampling discussed in Section 3.3.2. In this case, unstructured sampling is conducted in the entire tensor. Although the resulted sparse tensor can still be stored in the traditional CSR format as Figure 3.3, it can not perform an efficient inference due to extra comparison computation discussed above.

We present our algorithm of row-based unstructured sampling in Algorithm 3.2. We focus on sampling in a weight tensor $\boldsymbol{w}$ with a predefined row size N. Note that N must be divisible by the total number of weights in $\boldsymbol{w}$. The weight tensor $\boldsymbol{w}$ is then reshaped into the form of $\mathbb{R}^{G \times N}$, i.e., N weights per row and G rows in total. Given K sparsity levels $s_{1:K}$, K binary masks with the form of $\{0,1\}^{G \times N}$ are generated. Binary masks can be reshaped into the original form of the weight tensor accordingly.

3.3.5 How to Further Boost Subnets

Batch normalization (BN) layers are critical for the stable training of state-of-the-art DNNs. Previous synthesis works [YYX+19, YH19b] find that subnets with different width may cause an accumulated error on batch

statistics, and propose to switch BN layers for different subnets. Multiple subnets in DRESS share a single architecture, and thus are capable of being optimized in synergy with a shared BN layer. However, post-training BN layers for each subnet can better calibrate the running statistics, which in turn increases the accuracy. As BN layers often only require a rather smaller amount of memory and computation in comparison to conv/fc layers, we propose to further fine-tune BN layers for each subnet individually after parallel training, as shown in Algorithm 3.1.

3.4 Evaluation

With our design-flow mentioned above, we are now synthesizing the algorithm to map onto resource-constrained edge platforms. To better understand the effectiveness of our algorithm, we first evaluate our algorithm on widely used vision benchmarks in this section. Then, we compile and deploy the generated subnets on an edge platform in Section 3.5 to see the actual performance of the entire synthesis.

3.4.1 Benchmarking Details

We implement our algorithm with Pytorch [PGC+17], and evaluate on image classification and object detection/instance segmentation tasks. As prior works [HCL+18, YYX+19, YH19b, RFC20, PIVA21, ZMZ+21, SZS+21], for image classification, we benchmark VGGNet [SZ15] and ResNet20 [HZRS16] on CIFAR10 [KNH09], and benchmark ResNet50 [HZRS16] and MobileNetV1/V2 [HZC+17, SHZ+18] on ImageNet [RDS+15]; for object detection, we benchmark Faster-RCNN with ResNet50-FPN on COCO [LMB+15]; for instance segmentation, we benchmark Mask-RCNN with ResNet50-FPN on COCO [LMB+15]. We use Nesterov SGD optimizer with the cosine schedule for learning rate decay. We report the Top-1 test accuracy for the subnets of the epoch when the validation dataset achieves the highest average accuracy over all subnets. For all pre-processing and random initialization, we apply the tools provided in Pytorch.

In our experiments, the row-based unstructured sampling is conducted in all conv/fc layers, except for the depthwise conv layers in MobileNetV1/V2. We found that sparse depthwise conv layers lead to substantially lower accuracy. As depthwise conv layers only consume a rather small amount of memory and computation [EGM+21, Cho21], different subnets share the same dense depthwise conv layers in DRESS. In addition, we keep BN layers dense as in [YYX+19, YH19b]. We set the overall sparsity levels $s_{1:5} = 0.95, 0.98, 0.99, 0.995, 0.998$ for VGGNet, $s_{1:5} = 0.8, 0.9, 0.95, 0.98, 0.99$ for ResNet20, $s_{1:4} = 0.5, 0.8, 0.9, 0.95$ for ResNet50 and MobileNetV1/V2. The sparsity levels are averaged over

all conv/fc layers.

3.4.1.1 VGGNet/ResNet20 on CIFAR10

CIFAR10 [KNH09] is an image classification dataset, which consists of 32×32 color images in 10 object classes. We use the original training dataset with 50000 samples for training, and randomly select 2000 samples in the original test dataset (10000 samples in total) for validation, and the rest 8000 samples for testing. We train on 1 Nvidia V100 GPU with a batch size of 128.

VGGNet. The used VGGNet is widely adopted in many previous compression works [CBD15, HYK17, RORF16], which is a modified version of the original VGG [SZ15]. The used VGGNet architecture is presented as, 2×128C3 - MP2 - 2×256C3 - MP2 - 2×512C3 - MP2 - 2×1024FC - 10SVM/100SVM. The initial learning rate is set as 0.1; the momentum is set as 0.9; the weight decay is set as 0.0005; the number of training epochs is set as 100. Note that we use the same training hyperparameters for all three stages in Algorithm 3.1. This also holds true for the following experiments.

ResNet20. The network architecture is the same as ResNet-20 in the original paper [HZRS16]. The initial learning rate is set as 0.1; the momentum is set as 0.9; the weight decay is set as 0.0005; the number of training epochs is set as 100.

3.4.1.2 ResNet50/MobileNetV1/MobileNetV2 on ImageNet

ImageNet [RDS+15] is a large-scale image classification dataset, which consists of high-resolution color images in 1000 object classes. We use the original training dataset with 1.28 million samples for training, and randomly select 10000 samples in the original validation dataset (50000 samples in total) for validation, and the rest 40000 samples for testing. We train on 4 Nvidia V100 GPUs with a batch size of 1024.

ResNet50. We use pytorch-style ResNet50, which is slightly different than the original Resnet-50 [HZRS16]. The down-sampling (stride=2) is conducted in 3×3 conv layer instead of 1×1 conv layer. The network architecture is the same as "resnet50" in [Pyt19b]. The initial learning rate is set as 0.5; the momentum is set as 0.9; the weight decay is set as 0.0001; the number of training epochs is set as 100.

MobileNetV1. The network architecture is the same as 1.0× MobileNet-224 in the original paper [HZC+17]. The initial learning rate is set as 0.5; the momentum is set as 0.9; the weight decay is set as 0.00001; the number of training epochs is set as 150.

MobileNetV2. The network architecture is the same as 1.0× MobileNetV2

in the original paper [SHZ+18]. The initial learning rate is set as 0.1; the momentum is set as 0.9; the weight decay is set as 0.00004; the number of training epochs is set as 300.

3.4.1.3 ResNet50-FPN on COCO

MS COCO [LMB+15] is object detection, segmentation, key-point detection, and captioning dataset. We use COCO 2017 dataset, which consists of high-resolution annotated images in 80 object classes. It contains a training dataset with 118000 annotated samples, and a validation dataset with 5000 data samples. We focus on object detection and instance segmentation. We report the standard COCO metrics, average precision (AP), which is averaged over Intersection-over-Union (IoU) thresholds $\in 0.5 : 0.05 : 0.95$. The bounding box level AP and the mask level AP are adopted in object detection and the instance segmentation, respectively. We follow the official reference training scripts provided by Pytorch [Pyt21] to set up our experiments. We distributed train on 8 Nvidia V100 GPUs with a batch size of 16 (2 per GPU). The final AP is reported on the validation dataset after the entire training.

ResNet50-FPN. We adopt Faster-RCNN [RHGS15] in object detection and Mask-RCNN [HGDG17] in instance segmentation. The overall network architecture consists of two parts, the basic network[1] and the head architecture. We use ResNet50 pretrained on ImageNet dataset as the basic network. As suggested by [HGDG17], the feature extractor, feature pyramid network (FPN) [LDG+17], is connected to ResNet50 in lateral. The bounding-box head and the mask head will then use the extracted feature to detect objects and segment instances. Especially, our network architectures are the same as the ones provided in the Pytorch reference training scripts [Pyt21]. As the batch size used in Faster-RCNN training and Mask-RCNN training is relatively small, we freeze the BN layers of ResNet50 as in [HGDG17, GRG+18].

Following [YYX+19], we first pretrain ResNet50 with Algorithm 3.1 on ImageNet, i.e., obtain 4 subnets of ResNet50 with sparsity 0.5, 0.8, 0.9, 0.95 as in Figure 3.9. The lateral FPN and the head architecture are added into ResNet50. We then train the overall network on COCO dataset with Algorithm 3.1 while fixing BN layers for each subnet.

3.4.2 Ablation Studies

We first implement a set of ablation experiments to study the effect of different components/parameters in DRESS. The ablation experiments

[1] To avoid confusion, we use *the basic model* to refer to *the backbone network* mentioned in the Faster-RCNN and Mask-RCNN papers [RHGS15, HGDG17]. The backbone network only stands for the original dense network in DRESS in this chapter.

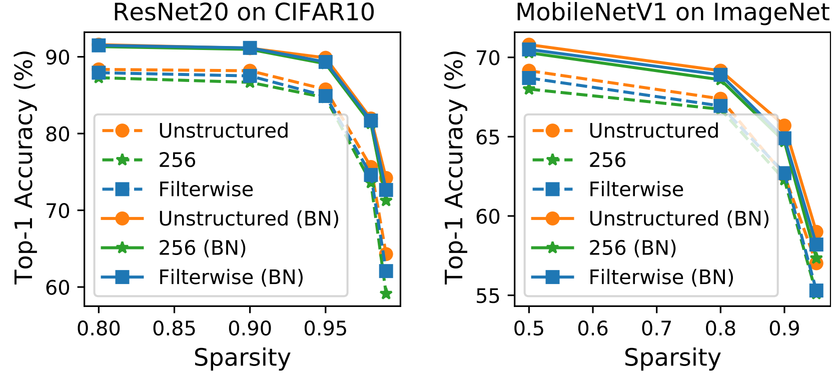

Figure 3.5: Ablation studies on different row sizes. “BN” means further fine-tuning BN layers for each subnet.

are mainly conducted with ResNet20 on CIFAR10 and MobileNetV1 on ImageNet.

3.4.2.1 Row Size N

Settings. In Section 3.3.4, we restrict different rows in a CSR sparse tensor to have the same number of nonzero weights, and subnets are sampled in a row-based unstructured manner. To study the impact of row size N, we select three methodical ways to reshape the weight tensor for row-based sampling, (*i*) unstructured, where unstructured sampling is conducted in the entire weight tensor, i.e., each layer only contains a single row in DRESS CSR format as discussed in Section 3.3.4; (*ii*) filterwise, where unstructured sampling is conducted in each filter for `conv` layers or in each output-neuron for `fc` layers; (*iii*) 256, where each row contains 256 elements in a `conv` filter, or the entire filter if the filter has less than 256 weights. γ is set as 1 in these experiments. The row size N used in CSR format is tightly related to the memory cost to store the column indices of nonzero weights, e.g., 256 means 8-bit for each column index.

Results. The results in Figure 3.5 show that when choosing a relatively large row size e.g., filterwise or 256, our proposed row-based unstructured sampling can yield a similar accuracy as totally unstructured sampling in the entire tensor. Especially, for both ResNet20 and MobileNteV1, the accuracy difference between “Unstructured” and “Filterwise” is less than 0.5% on average. In the following experiments, we mainly adopt filterwise unstructured sampling due to its high accuracy and efficient DRESS CSR format. The dashed curves and the solid curves with the same marker in Figure 3.5 can be viewed as the ablation study of the third training stage, i.e., further fine-tuning BN layers for each subnet (see Section 3.3.5). Particularly, fine-tuning BN layers can calibrate the statistic discrepancy between different subnets and thus consistently improves the performance of subnets.

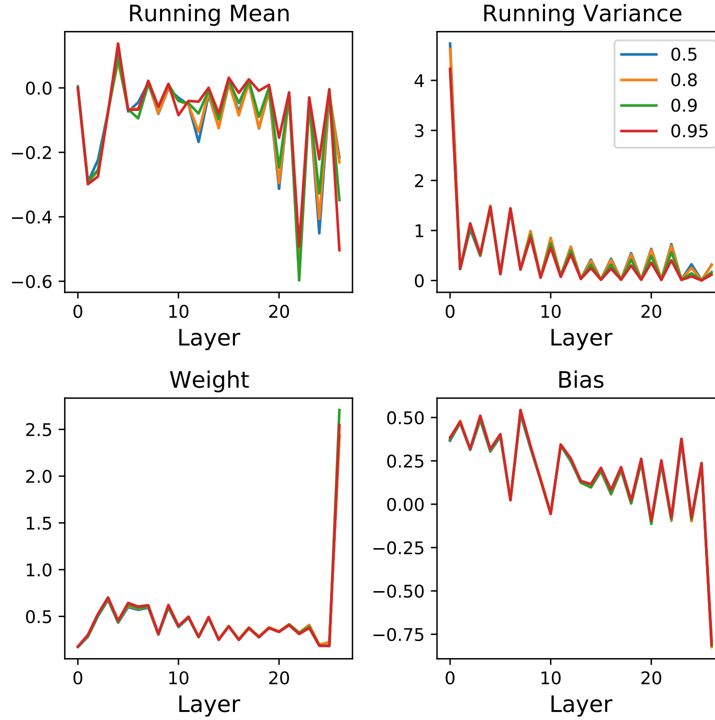

Figure 3.6: The BN statistics of different subnets across layers. The subnets of MobileNetV1 with different sparsity levels are plotted with different colors.

The BN statistic information of 4 subnets of MobileNetV1 is shown in Figure 3.6. For each layer in subnets, we plot the average value of "running mean", "running variance", "weight", and "bias" over all channels after the third training stage. The results show that the BN statistic information is closed among different subnets, which allows multiple subnets to be optimized in synergy in the second training stage (DRESS training) of Algorithm 3.1. On the other hand, the third training stage can calibrate the small discrepancy between different subnets, which in turn improves the accuracy of each subnet.

3.4.2.2 With/Without Sampling

Settings. To explore the efficacy of our sampling process, we compare DRESS with sampling (i.e., Algorithm 3.1) and DRESS without sampling. DRESS without sampling has a similar process as traditional unstructured magnitude pruning [HMD16, RFC20], where K binary masks are built after the dense pre-training and then are fixed, and only the nonzero weights (with mask value equals 1) are sparsely fine-tuned. In other words, the subnets will not be re-sampled in the DRESS training stage of Algorithm 3.1. We set γ as 1 and use filterwise unstructured sampling in these experiments.

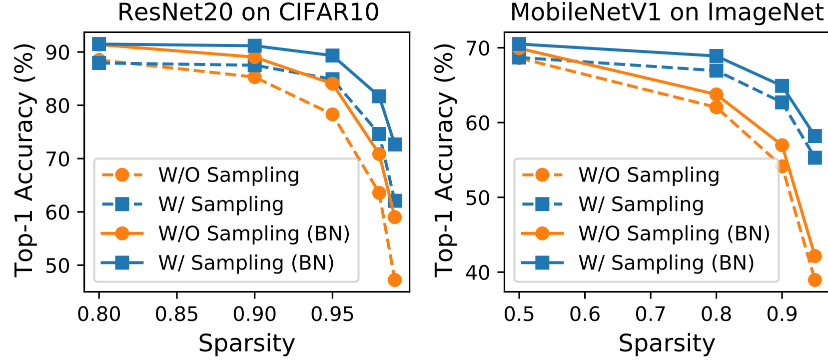

Figure 3.7: Comparison between DRESS with sampling and DRESS without sampling.

Results. As shown in Figure 3.7, our (re-)sampling process improves the accuracy of subnets by a large margin than without sampling, especially under a high sparsity, e.g., increasing by around 7.4% on average (up to 16.1%) on MobileNetV1. Re-sampling provides more flexibility to re-select the sparse subnets that are abandoned before the parallel training.

3.4.2.3 Iterative vs. Parallel

In this part, we compare DRESS (parallel training) with iterative training multiple subnets. We first elaborate the iterative training methods with progressively increased/decreased sparsity mentioned in Section 3.3.3.

Recall that there are K sparsity levels, and $0 < s_1 < s_2 < ... < s_K \leq 1$. In iterative training, each subnet is optimized separately, also altogether K iterations. In each iteration, we mainly adopt the idea of traditional unstructured pruning [RFC20], which is the current best-performed pruning method aiming at the trade-off between the model accuracy and the number of zero's weights. [RFC20] conducts iterative pruning with a pruning scheduler $p^{1:R}$. The network progressively reaches the desired sparsity s until the R-th pruning iteration. We choose $p^{1:5} = 0.5, 0.8, 0.9, 0.95, 1$, i.e., the sparsity is set to $0.5s, 0.8s, 0.9s, 0.95s, s$ in 5 pruning iterations, respectively. During each pruning iteration, the network is pruned with the corresponding sparsity, and the remaining nonzero weights are sparsely fine-tuned with learning rate rewinding.

The pseudocode of training subnets iteratively with increased sparsity is shown in Algorithm 3.3. With progressively increased sparsity (from s_1 to s_K), the first optimized subnet of $\boldsymbol{w} \odot \boldsymbol{m}_1$ already contains all subsequent subnets with higher sparsity due to Eq. (3.3). The first sparse subnet $\boldsymbol{w} \odot \boldsymbol{m}_1$ is trained by unstructured pruning [RFC20] as discussed above. In the following iteration k ($k \in 2, ..., K$), the subnet with sparsity s_k is directly sampled from the previous subnet without any retraining regarding the constraint of Eq. (3.3).

The pseudocode of training multiple subnets iteratively with

Algorithm 3.3: Iterative training with increased sparsity

Input: Initial random weights $\boldsymbol{w}$, training dataset $\mathcal{D}_{\text{tr}}$, validation dataset $\mathcal{D}_{\text{val}}$, sparsity $\{s_k\}_{k=1}^{K}$, pruning scheduler $\{p^r\}_{r=1}^{R}$

Output: Optimized weights $\boldsymbol{w}$, binary masks $\{\boldsymbol{m}_k\}_{k=1}^{K}$

`/* Dense pre-training */`

1 Train dense network $\boldsymbol{w}$ with traditional optimizer;

`/* Traditional pruning, also k=1 */`

2 **for** $r \leftarrow 1$ **to** R **do**

  `// The` r`-th pruning iteration`

3   Prune with sparsity $s_1 \cdot p^r$ and get mask $\boldsymbol{m}_1^r$;

4   Sparsely fine-tune nonzero weights $\boldsymbol{w} \odot \boldsymbol{m}_1^r$ on $\mathcal{D}_{\text{tr}}$;

5 Get mask $\boldsymbol{m}_1 = \boldsymbol{m}_1^R$;

`/* Iterative (training) */`

6 **for** $k \leftarrow 2$ **to** K **do**

7   Get the previous subnet $\boldsymbol{w}_{k-1} = \boldsymbol{w} \odot \boldsymbol{m}_{k-1}$;

8   Sample a subnet from $\boldsymbol{w}_{k-1}$ with sparsity s_k and get mask $\boldsymbol{m}_k$;

  `// Note that no training here.`

decreased sparsity is shown in Algorithm 3.4. For progressively decreased sparsity (from s_K to s_1), the sampling and training process only happen in the complementary part of the previous subnet due to Eq. (3.3). Particularly, in iteration k ($k \in K, ..., 1$), we should (*i*) sample the new subnet from the backbone network with sparsity s_k that contains the subnet of $\boldsymbol{w} \odot \boldsymbol{m}_{k+1}$; (*ii*) freeze the subnet of $\boldsymbol{w} \odot \boldsymbol{m}_{k+1}$ and only update the other weights. We still adopt the iterative pruning when training each subnet, i.e., the sparsity of the k-th subnet gradually approaches the target sparsity s_k. Note that the dense backbone network is maintained and updated during the training. Note that the (re-)sampling process is only conduct in each pruning iteration instead of each training iteration.

Settings. We implement the two iterative training methods of Algorithm 3.3 and Algorithm 3.4. The loss of each subnet is optimized separately in iterative training. Thus for a fair comparison, we do not re-weight loss in the parallel training of DRESS, i.e., $\gamma = 0$. Also in all experiments, we conduct unstructured sampling in the entire tensor, and allow BN layers to be fine-tuned individually for each subnet to avoid other side effects.

Results. The comparison results are plotted in Figure 3.8 Left. Parallel training substantially outperforms iterative training. Iterating over increased sparsity does not provide any space to optimize subnets with higher sparsity. Therefore, the accuracy drops quickly along iterations. Although iterating over decreased sparsity may yield a well-performed high sparsity network, the accuracy does not improve significantly

Algorithm 3.4: Iterative training with decreased sparsity

Input: Initial random weights $\boldsymbol{w}$, training dataset $\mathcal{D}_{\text{tr}}$, validation dataset $\mathcal{D}_{\text{val}}$, sparsity $\{s_k\}_{k=1}^{K}$, pruning scheduler $\{p^r\}_{r=1}^{R}$

Output: Optimized weights $\boldsymbol{w}$, binary masks $\{\boldsymbol{m}_k\}_{k=1}^{K}$

```
/* Dense pre-training                                    */
```
1 Train dense network $\boldsymbol{w}$ with traditional optimizer;
```
/* Iterative training                                    */
```
2 Set $s_{K+1} = 1$ and $\boldsymbol{m}_{K+1} = \boldsymbol{0}$;
3 **for** $k \leftarrow K$ **to** 1 **do**
4   Get the complementary subnet $\boldsymbol{w}^{\text{cs}} = \boldsymbol{w} \odot (1 - \boldsymbol{m}_{k+1})$;
5   **for** $r \leftarrow 1$ **to** R **do**
        // The r-th pruning iteration
6   Sample a subnet from $\boldsymbol{w}^{\text{cs}}$ with sparsity $(1 - (s_{k+1} - s_k)) \cdot p^r$ and get mask $\boldsymbol{m}_k^{\text{cs},r}$;
7   Merge mask $\boldsymbol{m}_k^r = \boldsymbol{m}_{k+1} + \boldsymbol{m}_k^{\text{cs},r}$;
8   Initiate $\boldsymbol{w}^0 = \boldsymbol{w}$;
9   **for** $q \leftarrow 1$ **to** Q **do**
            // The q-th training iteration
10   Fetch mini-batch from $\mathcal{D}_{\text{tr}}$;
11   Get sparse subnet $\boldsymbol{w}_k^{r,q-1} = \boldsymbol{w}^{q-1} \odot \boldsymbol{m}_k^r$;
12   Back-propagate subnet gradient $\boldsymbol{g}(\boldsymbol{w}_k^{r,q-1}) = \frac{\partial \ell(\boldsymbol{w}_k^{r,q-1})}{\partial \boldsymbol{w}_k^{r,q-1}}$;
13   Compute optimization step $\Delta \boldsymbol{w}^q$ with $\boldsymbol{g}(\boldsymbol{w}_k^{r,q-1}) \odot \boldsymbol{m}_k^{\text{cs},r}$;
14   Update $\boldsymbol{w}^q = \boldsymbol{w}^{q-1} + \Delta \boldsymbol{w}^q$;
15   Save $\boldsymbol{w} = \boldsymbol{w}^Q$
16   Save mask $\boldsymbol{m}_k = \boldsymbol{m}_k^R$;

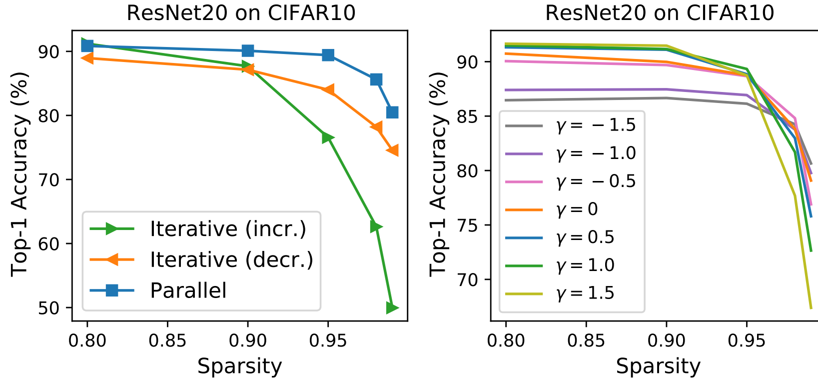

Figure 3.8: Left: Comparing parallel training with iterative training. Right: Ablation studies on the correction factor γ.

afterwards. We argue this is due to the fact that iterative training causes the optimizer to end in a hard to escape region around the previous subnet in the loss landscape. On the contrary, parallel training allows multiple subnets to be sampled and optimized jointly, which may especially benefit highly sparse networks, see Figure 3.8 Left.

3.4.2.4 Correction Factor γ

Settings. The loss weights π_k used in the parallel training may influence the final accuracy of different subnets. In Section 3.3.3, we introduce a correction factor γ to control π_k (see Eq. (3.6) and Eq. (3.7)). We thus conduct a set of experiments with different γ. $\gamma = 0$ means all loss items are weighted equally; $\gamma > 0$ means the loss of the lower sparsity subnets is weighted larger, and vice versa. For example, for ResNet20 with $s_{1:5} = 0.8, 0.9, 0.95, 0.98, 0.99$, when $\gamma = 0.5$, $\pi_{1:5} \approx 0.36, 0.26, 0.18, 0.12, 0.08$; $\gamma = -1.0$, $\pi_{1:5} \approx 0.03, 0.05, 0.11, 0.27, 0.54$.

Results. The results in Figure 3.8 Right show that the high sparsity subnets generally yield a higher final accuracy with a smaller γ. This is intuitive since a smaller γ assigns a larger weight on the high sparsity subnets. However, the downside is that the most powerful subnet (with the lowest sparsity) can not reach its top accuracy. Note that the most powerful subnet is often adopted either under the critical case requiring high accuracy or in the commonly used scenario with standard resource constraints, see in Section 3.1. Also as discussed in Section 3.3.3, low sparsity subnets should be weighted more, since they are implicitly optimized with a smaller step size. Experimentally, we find that $\gamma \in [0.5, 1]$ in parallel training allows us to train a group of subnets where the most powerful subnet can reach a similar accuracy as training in separately. We set $\gamma = 0.5$ in the following experiments.

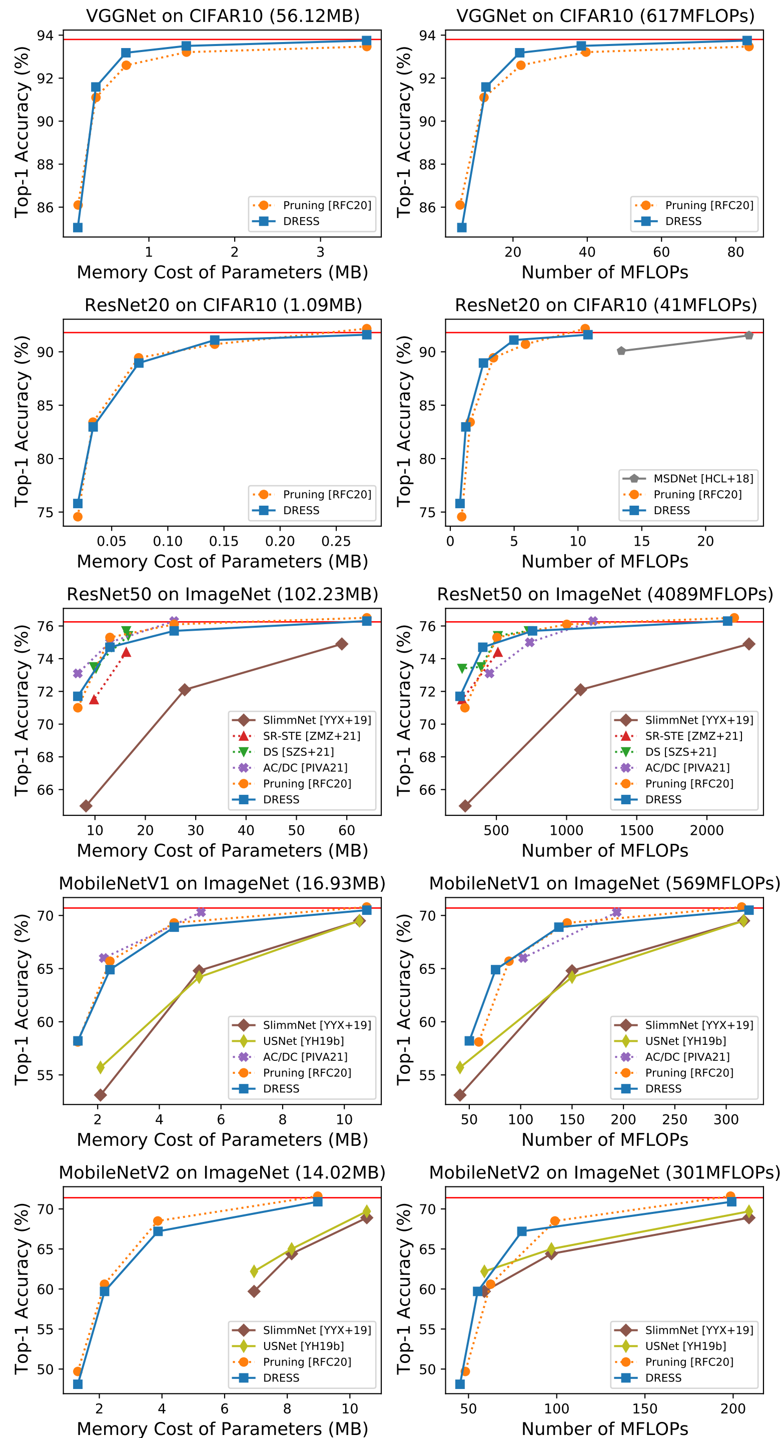

Figure 3.9: Comparing DRESS with other baselines on image classification. The methods that do not involve weight sharing among different networks are plotted with dotted curves. The memory cost and the number of MFLOPs of the original backbone networks are reported in the parentheses in titles; their accuracy is shown as red horizontal lines. The sparsity levels of DRESS and "Pruning" are the same as the ones discussed in Section 3.4.

Model	Average Accuracy		Overall Storage (MB)		Average MFLOPs	
	DRESS	Pruning	DRESS	Pruning	DRESS	Pruning
VGGNet	**91.4%**	91.1%	**3.54**	6.27	**32**	33
ResNet20	**86.1%**	85.9%	**0.28**	0.55	**4**	**4**
ResNet50	74.5%	**74.6%**	**63.97**	109.25	**887**	994
MobileNetV1	65.6%	**65.9%**	**10.72**	18.96	**146**	152
MobileNetV2	61.5%	**62.4%**	**8.98**	16.32	**95**	101

Table 3.1: The average test accuracy over all sub-networks, the theoretical storage (MB) required by all sub-networks and the average number of theoretical MFLOPs over all sub-networks.

3.4.3 Evaluation on Image Classification

Settings. In this section, we benchmark DRESS on public image classification datasets including CIFAR10 and ImageNet with different backbone networks discussed earlier in Section 3.4. We compare the performance of the subnets generated by DRESS with various methods, including (*i*) anytime networks [HCL+18, YYX+19, YH19b], where the sub-networks with different width or depth can be cropped from the backbone network; (*ii*) unstructured pruning [RFC20, PIVA21], where [RFC20] is re-implemented under our settings for a fair comparison; (*iii*) N:M fine-grained structure pruning [ZMZ+21, SZS+21]. We choose two metrics for comparison, the memory cost of parameters and the number of MFLOPs (10^6 FLOPs). Both metrics are widely used proxies of resource consumption. FLOPs dominate in the entire computation burden, thus fewer FLOPs can (but does not necessarily) result in a smaller computation time. The memory cost of parameters not only represents the static storage consumption but also relates to the amount of memory fetching when on-device inference with different (sub-)networks [AAH+20]. Note that memory access often consumes more time and more energy than computation [Hor14]. Assume that each parameter uses 32-bit for floating point values. DRESS, (*ii*), and (*iii*) generate sparse tensors, thus their memory cost also includes the indices of nonzero weights. Following the suggestions of [ABC+16, Goo19], each index of nonzero weights is encoded into 8-bit in DRESS and (*ii*), whereas the binary mask is stored for indexing in [ZMZ+21, SZS+21].

Results. The results are plotted in Figure 3.9. In comparison to other anytime networks, the subnets generated by DRESS require a significantly lower memory cost and fewer FLOPs under the same accuracy level. In addition, the sub-networks of conventional anytime networks [HCL+18, YYX+19, YH19b] have different network architectures, while current compilation libraries (e.g., TensorFlowLite) may not support to adopt a dynamic architecture on-device. The extra re-configuration overhead e.g., storing various compiled architectures could be necessary for on-device

inference. However, this is avoided in DRESS, since different subnets of DRESS leverage the same architecture as the backbone network. Like traditional unstructured pruning, DRESS does not explicitly reduce the number of operations, i.e., the networks with the same sparsity can require different numbers of FLOPs to perform inference as shown in Figure 3.9. Thanks to the weight sharing, the static storage is only determined by the largest network for both DRESS and anytime networks [HCL+18, YYX+19, YH19b]. The methods of (*ii*)-(*iii*) do not involve weight sharing, thus they need more memory to store all networks separately.

We further compare DRESS with the unstructured pruning method [RFC20], in terms of the test accuracy, the theoretical storage, and the theoretical FLOPs, see Table 3.1. DRESS reaches a similar average accuracy and computation complexity while only requiring 50%-60% of storage as pruning.

3.4.4 Evaluation on Object Detection/Instance Segmentation

Settings. To show the versatility of our synthesis technique, we further benchmark DRESS on other vision tasks, object detection and instance segmentation. We compare DRESS with other baselines mentioned in Section 3.4.3 on MS COCO 2017 dataset. We adopt Faster-RCNN with ResNet50-FPN [RHGS15] in object detection and Mask-RCNN [HGDG17] with ResNet50-FPN in instance segmentation.

Results. Since the number of FLOPs for Faster-RCNN and Mask-RCNN depends on the number of proposals in each image [CMS+20], we report the average number of FLOPS for the randomly selected 100 images in COCO 2017 validation dataset. We compute the FLOPS with the tool flop count operators from Detectron2 [WKM+19]. For Faster-RCNN, we report its bounding box AP; for Mask-RCNN, we report its bounding box AP and its mask AP. The results are plotted in Figure 3.10. Similar to the results in Figure 3.9, the subnets generated by DRESS require a significantly lower memory cost and fewer GFLOPs (10^9 FLOPs) than other anytime networks [YYX+19]. In addition, in comparison to the unstructured pruning [RFC20] that does not involve weight sharing, DRESS can also achieve a similar precision level.

3.4.5 Sparsity across Layers

To further explore the different impact from DRESS and traditional pruning, we compare their layerwise sparsity. Recall that the main differences between DRESS and traditional pruning are, (*i*) the nonzero weights of the higher sparsity subnets are reused by the lower sparsity subnets in DRESS, whereas different sparse networks generated by traditional pruning are independent; (*ii*) DRESS maintains an unstructured sparse pattern in a row-based manner (i.e., fine-grained

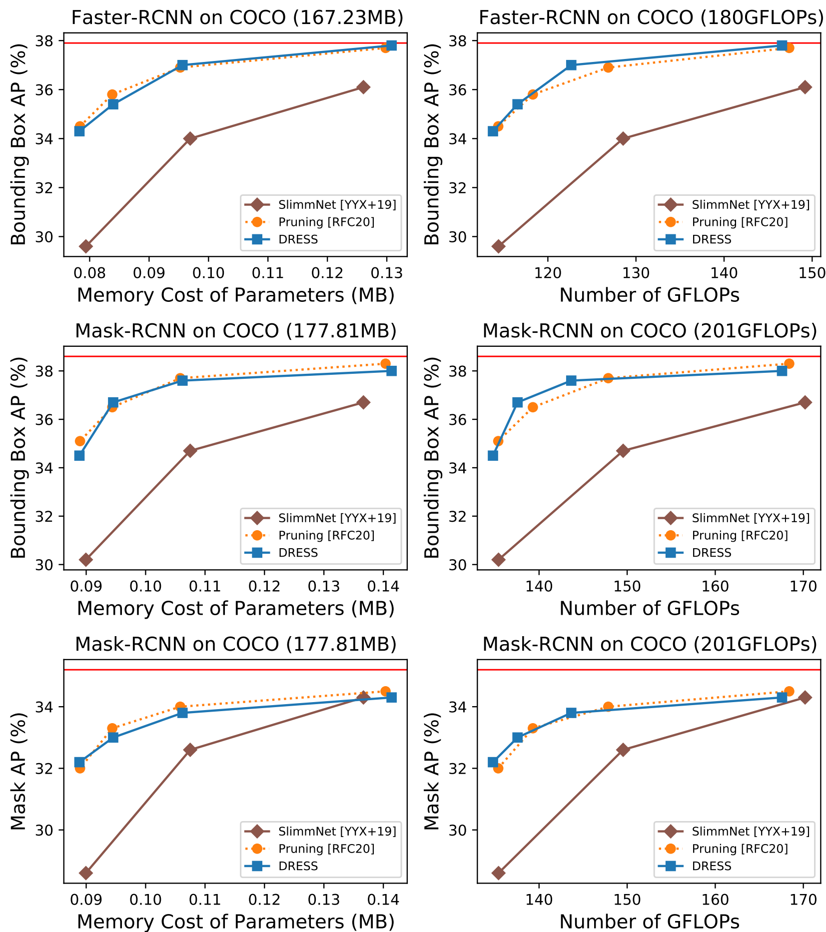

Figure 3.10: Comparing DRESS with other baselines on object detection and instance segmentation. The methods that do not involve weight sharing among different networks are plotted with dotted curves. The memory cost and the number of GFLOPs of the original backbone networks are reported in the parentheses in titles; their average precision is shown as red horizontal lines.

Model	Sparsity	MobileNetV1		MobileNetV2	
Model		Dense		Dense	
Time (ms)	0%	83		52	
Model		DRESS	Pruning	DRESS	Pruning
Time (ms)	50%	**77**	80	**47**	48
	80%	**45**	55	**36**	41
	90%	**31**	35	**29**	32
	95%	**25**	27	**26**	**26**

Table 3.2: The average inference time (ms) on RaspberryPi 4.

structure sparsity [ZMZ+21, HCI+21, SZS+21]), whereas traditional pruning yields an unstructured sparse pattern in the entire tensor.

We plot the layerwise sparsity of the sparse (sub-)networks generated by DRESS and traditional pruning [RFC20], for ResNet20 on CIFAR10 in Figure 3.11 and for MobileNetV2 on ImageNet in Figure 3.12. In general, both methods have a similar layerwise sparsity in each subplot. However, there exists a diversity under a low sparsity level, e.g., MobileNetV2 with sparsity 0.5. Jointly training with weight sharing in DRESS enforces the low sparsity network to be optimized towards a certain region that has been less explored in the individual training of pruning, as the low sparsity network often has relatively looser constraints.

3.5 Deployments

To measure actual performance and compare it to the benchmark evaluation, we used DRESS-generated subnets on a RaspberryPi 4 edge platform (with off-the-shelf Arm Cortex-A72 quad-core CPUs) for on-device inference. The optimized Pytorch model is compiled by TensorFlow Lite [ABC+16] with XNNPACK [Goo19] delegate for deployment. We use multi-threading with 4 threads for acceleration.

The inference latency when adopting different subnets of MobileNetV1 and MobileNetV2 on RaspberryPi 4 is reported in Table 3.2. The original dense models and the sparse models generated from unstructured pruning methods [RFC20] are also added in Table 3.2 for comparison. The reported latency is averaged over 100 randomly selected samples from ImageNet dataset. By using the fast kernels for sparse matrix-dense matrix multiplication provided by XNNPACK, DRESS can dynamically select its subnets to satisfy the various inference latency constraints. Note that the sparse (sub-)networks of DRESS and pruning [RFC20] have a similar number of theoretical FLOPs (see Figure 3.9 and Table 3.1), yet DRESS often yields a lower inference time. This is due to the fact that row-based unstructured sparsity leads to regular computation among different rows, which speeds up inference [ZMZ+21].

Note also that although the inference time decreases when adopting

the subnets with a higher sparsity, the realistic speedup of sparse inference is not proportional to the reduction in theoretical FLOPs. For example, the theoretical FLOPs decrease by a factor of 6.4 when the sparsity of DRESS MobileNetV1 subnets increases from 50% to 95%, while the inference is only accelerated by a factor of 3.1. A similar phenomenon can also be observed in MobileNetV2 and pruned models. We suspect that the reason is that sparse computational cores of XNNPACK have a larger fraction of cache miss at a higher sparsity level, see also in [EDGS20].

3.6 Summary

This chapter develops a novel synthesis approach DRESS that can adapt the sub-networks for on-device inference to maximize the model performance with different resource budgets. DRESS enables efficient adaptation on edge devices under varying resource constraints. Prior synthesis methods either require deploying multiple individual networks, or sample sub-network architectures along structured dimensions leading to subpar performance. However, DRESS utilizes nonzero-weight sharing and architecture sharing to reduce the redundancy among multiple unstructured sub-networks, resulting in both storage efficiency and re-configuration efficiency. The main contributions of DRESS are summarized as follows,

- DRESS can adapt different sub-networks sampled from the backbone network on edge devices. These optimized sub-networks have different sparsity, and thus can infer under various resource constraints, e.g., the inference latency, and the battery energy.

- DRESS samples sub-networks in a row-based unstructured sparsity (a.k.a. fine-grained structure sparsity) and introduces a novel compressed sparse row (CSR) format for storing the sub-networks. This way, multiple sub-networks can be efficiently fetched and executed for on-device inference, by using the fast kernels of sparse tensor computation provided by recent compilation libraries. To our best knowledge, this is the first work that builds multiple sub-networks via a fine-grained structure of weight sharing.

- DRESS enables weight sharing and architecture sharing among multiple sub-networks, resulting in (static) storage efficiency and re-configuration efficiency, respectively.

- Experimental results show DRESS reaches a similar accuracy while only requiring 50%-60% of static storage as unstructured pruned networks, and can result in various distinct inference latency on off-the-shelf edge platforms according to different sparsity levels.

This chapter studied how to adapt the network on edge devices to maximize the inference accuracy under varying resource constraints. In the next chapter, we will study how to conduct learning on edge devices given a few data samples of new tasks. The different sub-networks generated by DRESS are used for the same inference task, thus DRESS is inapplicable to adapt its network given a new task.

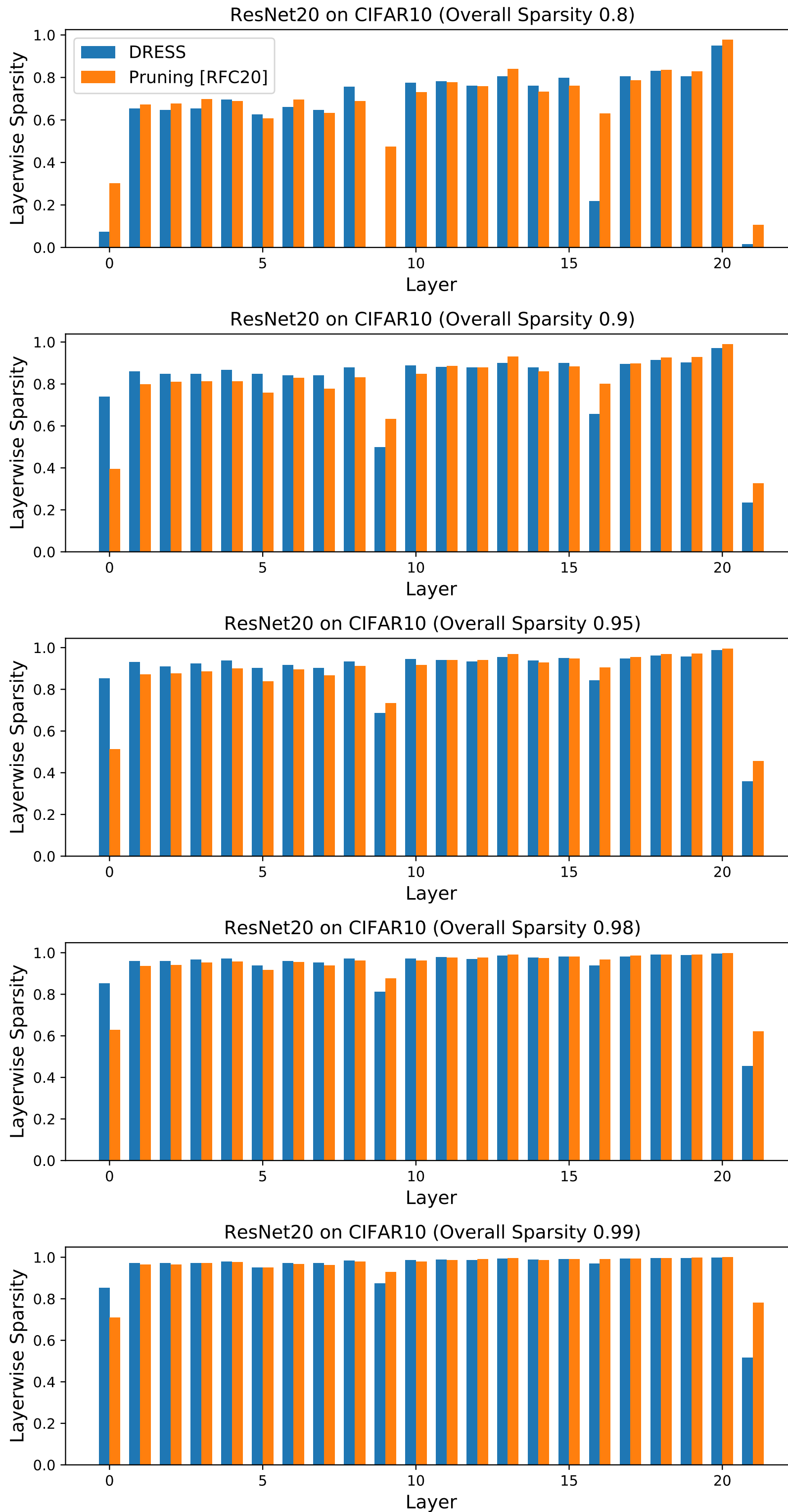

Figure 3.11: Comparing DRESS with traditional pruning on ResNet20 (CIFAR10) in terms of the layerwise sparsity. The (sub-)networks with different overall sparsity levels $(0.8, 0.9, 0.95, 0.98, 0.99)$ are plotted in different subplots.

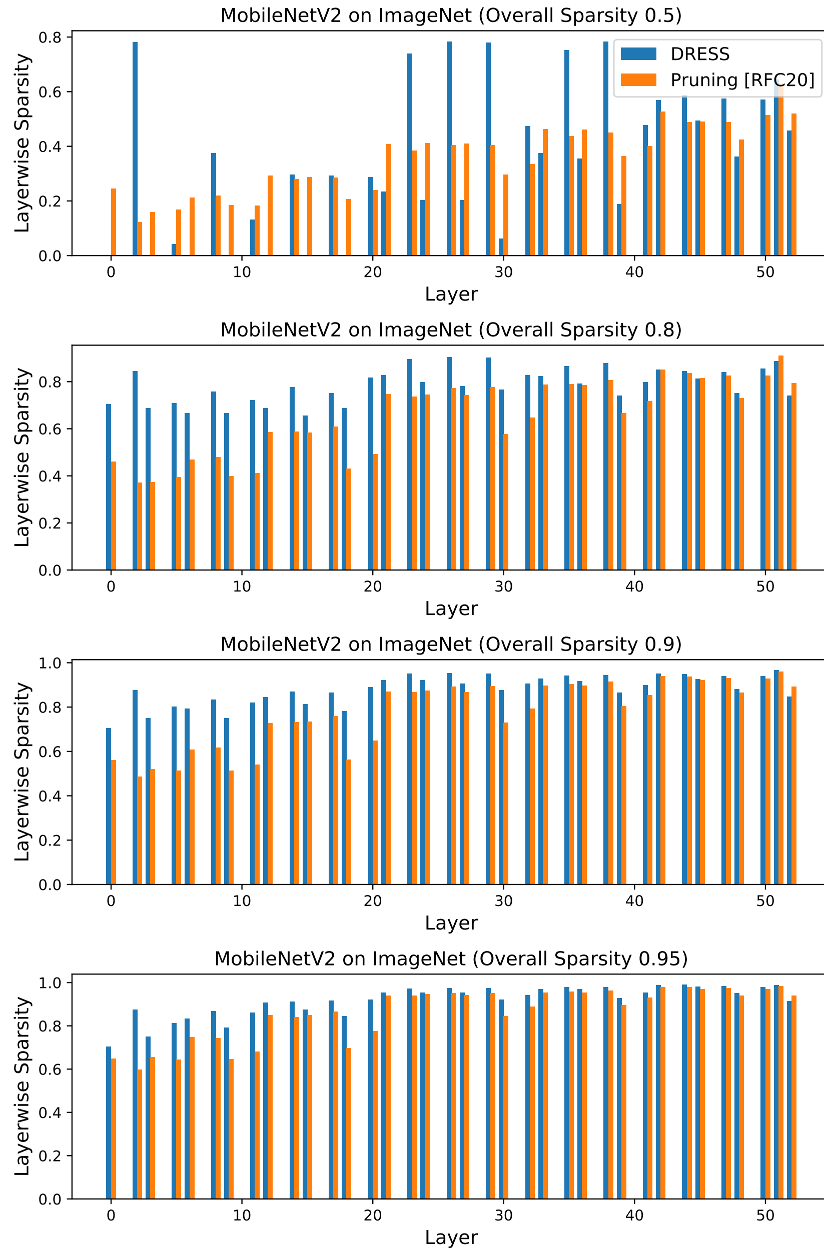

Figure 3.12: Comparing DRESS with traditional pruning on MobileNetV2 (ImageNet) in terms of the layerwise sparsity. The (sub-)networks with different overall sparsity levels $0.5, 0.8, 0.9, 0.95$ are plotted in different subplots.

4

Learning on Edge Devices

In Chapter 2 and Chapter 3, we studied how to compress a pretrained DNN for on-device inference under *fixed* and *varying* resource constraints. However, when facing *unseen* environments, users, or tasks, it is crucial to adapt[1] the pretrained DNN to deliver consistent performance and customized services. Sometimes, data collected by edge devices are private and have a large diversity across users/devices. Hence, *on-device learning* is preferred over uploading the data to cloud servers for adaptation.

Main Resource Constraints. For on-device learning, neither abundant *user data* nor *computing resources* are applicable. On the one hand, the amount of user data collected on a single edge device is rather small due to the limited labor resources. On the other hand, edge devices often have a small amount of available resources from memory and computation.

Principles. Existing memory-efficient training approaches are not able to optimize a DNN given only a few training samples, whereas current meta learning methods require a significant amount of dynamic memory to few-shot learn unseen tasks. Therefore, we introduce a memory-efficient on-device few-shot learning setting, and propose a novel meta learning scheme that can (*i*) fast learn new unseen tasks given a few training samples, resulting in data efficiency, (*ii*) avoid redundant training by distinguishing and learning adaptation-critical weights only, leading to memory efficiency.

The contents of this chapter are established mainly based on the paper "p-Meta: Towards On-device Deep Model Adaptation" that is published on ACM Conference on Knowledge Discovery and Data Mining (SIGKDD), 2022.

[1] In this chapter, the adaptation is referred to as (re-)training on new data samples.

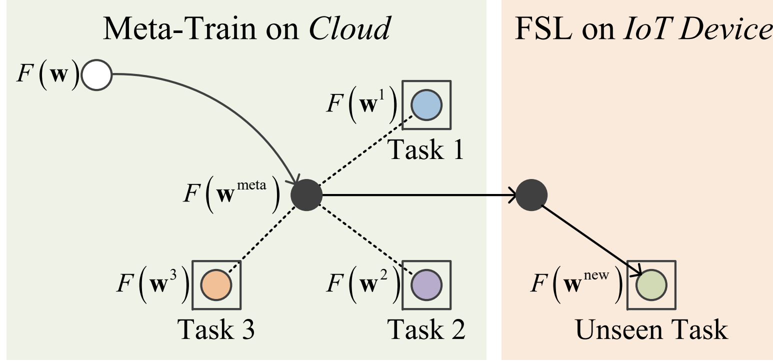

Figure 4.1: Meta learning and few-shot learning (FSL) in the context of on-device learning. The backbone $F(\boldsymbol{w})$ is meta-trained into $F(\boldsymbol{w}^{\text{meta}})$ on the cloud and is deployed to IoT devices to learn unseen tasks as $F(\boldsymbol{w}^{\text{new}})$ via FSL.

4.1 Introduction

The excellent accuracy of contemporary DNNs is attributed to training with high-performance computers on large-scale datasets [GBCB16]. For example, it takes 29 hours to complete a 90-epoch ResNet50 [HZRS16] training on ImageNet (1.28 million training images) [RDS+15] with 8 Nvidia Tesla P100 GPUs [GDG+17]. However, on-device learning/adaptation of a DNN demands both *data efficiency* and *memory efficiency*. A personal voice assistant, for example, may learn to adapt to users' accent and dialect within a few sentences, while a home robot should learn to recognize new object categories with few labelled images to navigate in new environments. Furthermore, such learning is expected to be conducted on low-resource platforms such as smart portable devices, home hubs, and other IoT devices, with only several *KB* to *MB* memory.

For *data-efficient* DNN training, we resort to *meta learning*, a paradigm that learns to fast generalize to unseen tasks [HAMS20]. Of our particular interest is *gradient-based* meta learning [AES19, FAL17, RRBV20, OYKY21] for its wide applicability in classification, regression and reinforcement learning, as well as the availability of gradient-based training frameworks for low-resource devices, e.g., TensorFlow Lite [Ten]. Figure 4.1 explains major terminologies in the context of on-device learning. Given a backbone, its weights are *meta-trained* on *many* tasks, to output a *model* that is expected to fast learn new *unseen* tasks. The process of learning is also known as *few-shot learning*, where the meta-trained model is further retrained by standard stochastic gradient decent (SGD) on *few new* samples only.

However, existing gradient-based meta learning schemes [AES19, FAL17, RRBV20, OYKY21] fail to support *memory-efficient* training. Although *meta training* is conducted in the cloud, *few-shot learning* of the meta-trained model is performed on IoT devices. Consider to retrain a

common backbone ResNet12 in a 5-way (5 new classes) 5-shot (5 samples per class) scenario. One round of SGD consumes 370.44MB peak dynamic memory, since the inputs of all layers must be stored to compute the gradients of these layers' weights in the backward path. In comparison, inference only needs 3.61MB. The necessary dynamic memory is a key bottleneck for on-device learning due to cost and power constraints, even though the meta-trained model only needs to be retrained with a few data.

Prior efficient DNN training solutions mainly focus on parallel and distributed training on data centers [CXZG16, CLP+21, GSS17, GMD+16, RA20]. On-device training has been explored for *vanilla supervised training* [GSZ+20, MBdG+21, LN19], where training and testing are performed on the *same* task. A pioneer study [CGZH20] investigated on-device learning to new tasks via memory-efficient *transfer learning*. Yet transfer learning is prone to overfitting when only a few samples are available [FAL17].

In this paper, we propose **p-Meta**, a new meta learning method for data- and memory-efficient DNN training. The key idea is to enforce *structured partial parameter updates* while ensuring *fast generalization to unseen tasks*. The idea is inspired by recent advances in understanding gradient-based meta learning [OYKY21, RRBV20]. Empirical evidence shows that only the *head* (the last output layer) of a DNN needs to be updated to achieve reasonable few-shot classification accuracy [RRBV20] whereas the *body* (the layers closed to the input) needs to be updated for cross-domain few-shot classification [OYKY21]. These studies imply that certain weights are more important than others when generalizing to unseen tasks. Hence, we propose to automatically identify these *adaptation-critical weights* to minimize the memory demand in few-shot learning.

Particularly, the critical weights are determined in two structured dimensionalities as, (*i*) layer-wise: we meta-train a layer-by-layer learning rate that enables a *static* selection of critical layers for updating; (*ii*) channel-wise: we introduce meta attention modules in each layer to select critical channels *dynamically*, i.e., depending on samples from new tasks. Partial updating of weights means that (structurally) sparse gradients are generated, reducing memory requirements to those for computing nonzero gradients. In addition, the computation demand for calculating zero gradients can be also saved. To further reduce the memory, we utilize *gradient accumulation* in few-shot learning and *group normalization* in the backbone. Although weight importance metrics and SGD with sparse gradients have been explored in vanilla training [RA20, DLH+20, GSZ+20, HMD16], it is unknown *(i)* how to identify adaptation-critical weights and *(ii)* whether meta learning is robust to sparse gradients, where the objective is to fast learn *unseen* tasks.

4.2 Related Work

4.2.1 Meta Learning for Few-Shot Learning

Meta learning is a prevailing solution to few-shot learning [HAMS20], where the meta-trained model can learn an unseen task from a few training samples, i.e., data-efficient training. The majority of meta learning methods can be divided into two categories, (*i*) embedding-based methods [VBL+16, SSZ17, SYZ+18] that learn an embedding for classification tasks to map the query samples onto the classes of labeled support samples, (*ii*) gradient-based methods [AES19, FAL17, RRBV20, OYKY21, VOZK+21] that learn an initial model (and/or optimizer parameters) such that it can be trained with gradient information calculated on the new few samples. Among them, we focus on gradient-based meta learning methods for their applicability in various learning tasks and the availability of gradient-based training frameworks for low-resource devices [Ten].

Particularly, we aim at meta training a DNN that allows fast learning on memory-constrained devices. Most meta learning algorithms [AES19, FAL17, VOZK+21] optimize the backbone network for better generalization yet ignore the workload if the meta-trained backbone is deployed on low-resource platforms for few-shot learning. Manually fixing certain layers during on-device few-shot learning [RRBV20, OYKY21, SLQ+21] may also reduce memory and computation, but to a much lesser extent as shown in our evaluations.

4.2.2 Efficient DNN Training

Existing efficient training schemes are mainly designed for high-throughput GPU training on large-scale datasets. [CBG+20, WCB+18] conduct 8-bit floating point low precision training which requires specialized hardware for efficient execution. A general strategy is to trade memory with computation [CXZG16, GMD+16], which is unfit for IoT device with a limited computation capability. An alternative is to sparsify the computational graphs in backpropagation [RA20]. Yet it relies on massive training iterations on large-scale datasets. Other techniques include layer-wise local training [GSS17] and reversible residual module [GRUG17], but they often incur notable accuracy drops.

There are a few studies on DNN training on low-resource platforms, such as updating the last several layers only [MBdG+21], reducing batch sizes [LN19], and gradient approximation [GSZ+20]. However, they are designed for vanilla supervised training, i.e., train and test on the same task. One recent study proposes to update the bias parameters only for memory-efficient transfer learning [CGZH20], yet transfer learning is prone to overfitting when trained with limited data [FAL17].

4.3 Preliminaries and Challenges

In this section, we first motivate on-device few-shot learning via example applications, then provide the basics on meta learning for few-shot learning and highlight the challenges to enable on-device learning.

4.3.1 Example Application Scenarios

On-device few-shot learning is essential for model adaptation in some intelligent applications, when the new data collected on edge devices tend to relate to personal habits and lifestyle. For instance, activity recognition with smartphone sensors should adapt to countless walking patterns and sensor orientation [GKSL19]. Gaze tracking with smart glasses requires calibration to personal gaze conditions for cognitive context recognition [LHSG20]. Human motion prediction with home robots needs fast learning of unseen poses for seamless human-robot interaction [GWRM18]. We detail two representative applications below and summarize their resource utilization in Table 4.2.

Home Surveillance Customization. Household camera systems are pervasively deployed to detect intruders and monitor pets, where suspicious images are uploaded to a smart gateway for further investigation such as object classification. Due to the *countless object classes* of interest across individuals, the image classification model needs post-deployment customization. Fast model adaptation (e.g., pre-trained on dog breeds such as Komondor, Poodle and Saluki, and re-trained to recognize Malamute) at the smart gateway delivers more targeted surveillance services without leaking images of family members or private locations.

Robot Locomotion Control. Robots that walk and run as humans have been a long-standing challenge in robotics [ITF+21]. Deep reinforcement learning (DRL) advances the development of naturally behaved robots for new applications such as police robotic dogs and unmanned last-mile delivery [HLD+19]. It is important that the robots fast learn their locomotion policies to new goals and environments since there is often a gap between the training and deployment environments. Naive DRL can take millions of data samples to learn meaningful locomotion gaits [ITF+21]. Conversely, on-robot few-shot DRL enables rapid control policy acquisition with few new experience.

4.3.2 Meta Learning for Few-Shot Learning

Meta learning is a prevailing solution to adapt a DNN to unseen tasks with limited training samples, i.e., few-shot learning [HAMS20]. We ground our work on model-agnostic meta learning (MAML) [FAL17], a generic meta learning framework which supports classification, regression and

reinforcement learning. Table 4.1 lists the major notations.

Notation	Description
l	Layer index, $l \in 1, 2, ..., L$
$\boldsymbol{x}_{l-1}$, $\boldsymbol{w}_l$, $\boldsymbol{y}_l$	Input, weight, and intermediate tensors
C_l, H_l, W_l	Output channel number, height and width
$\boldsymbol{x}_L = F(\boldsymbol{w}; \boldsymbol{x}_0)$	A model (backbone) with parameter $\boldsymbol{w}$, its input and output
T^i	Sampled task i from distribution $p(\mathsf{T})$ during meta training
$\mathcal{D}^i = \{\mathcal{S}^i, \mathcal{Q}^i\}$	Dataset with support set $\mathcal{S}^i$ and query set $\mathcal{Q}^i$ for task T^i
T^{new}	New unseen task during on-device few-shot learning
$\mathcal{D}^{\text{new}} = \{\mathcal{S}^{\text{new}}, \mathcal{Q}^{\text{new}}\}$	Dataset for unseen task T^{new} $\mathcal{S}^{\text{new}}$ for few-shot learning and $\mathcal{Q}^{\text{new}}$ for evaluation
$\ell(\boldsymbol{w}; \mathcal{D})$	loss function over model $F(\boldsymbol{w})$ and dataset $\mathcal{D}$
$\boldsymbol{w}^{\text{meta}}$, $\boldsymbol{w}^{\text{new}}$	parameters after meta training and few-shot learning
$\boldsymbol{w}^{i,k}$	model parameters $\boldsymbol{w}^i$ at step k on task i in inner loop
$\boldsymbol{\alpha}$, β	inner and outer step size
$\boldsymbol{g}(\cdot)$	the loss gradients w.r.t. the given tensor
$\sigma(\cdot)$, $\sigma'(\cdot)$	Non-linear function and its derivative
$m(\cdot)$	memory consumption of the given tensor in words

Table 4.1: Summary of major notations.

Given the dataset $\mathcal{D} = \{\mathcal{S}, \mathcal{Q}\}$ of an unseen few-shot task, where $\mathcal{S}$ (support set) and $\mathcal{Q}$ (query set) are for training and testing, MAML trains a model $F(\boldsymbol{w})$ with weights $\boldsymbol{w}$ such that it yields high accuracy on $\mathcal{Q}$ even when $\mathcal{S}$ only contains a few samples. This is enabled by simulating the few-shot learning experiences over abundant few-shot tasks sampled from a task distribution $p(\mathsf{T})$. Specifically, it meta-trains a backbone F over few-shot tasks $\mathsf{T}^i \sim p(\mathsf{T})$, where each T^i has dataset $\mathcal{D}^i = \{\mathcal{S}^i, \mathcal{Q}^i\}$, and then generates $F(\boldsymbol{w}^{\text{meta}})$, an initialization for the unseen few-shot task T^{new} with dataset $\mathcal{D}^{\text{new}} = \{\mathcal{S}^{\text{new}}, \mathcal{Q}^{\text{new}}\}$. Training from $F(\boldsymbol{w}^{\text{meta}})$ over $\mathcal{S}^{\text{new}}$ is expected to achieve a high test accuracy on $\mathcal{Q}^{\text{new}}$.

MAML achieves fast learning via two-tier optimization. In the *inner loop*, a task T^i and its dataset $\mathcal{D}^i$ are sampled. The weights $\boldsymbol{w}$ are updated to $\boldsymbol{w}^i$ on support dataset $\mathcal{S}^i$ via K gradient descent steps, where K is usually small, compared to vanilla training:

$$\boldsymbol{w}^{i,k} = \boldsymbol{w}^{i,k-1} - \alpha \nabla_{\boldsymbol{w}} \, \ell\left(\boldsymbol{w}^{i,k-1}; \mathcal{S}^i\right) \quad \text{for } k = 1, ..., K \tag{4.1}$$

where $\boldsymbol{w}^{i,k}$ are the weights at step k in the inner loop, and α is the inner step size. Note that $\boldsymbol{w}^{i,0} = \boldsymbol{w}$ and $\boldsymbol{w}^i = \boldsymbol{w}^{i,K}$. $\ell(\boldsymbol{w}; \mathcal{D})$ is the loss function on dataset $\mathcal{D}$. In the *outer loop*, the weights are optimized to minimize the sum of loss at $\boldsymbol{w}^i$ on query dataset $\mathcal{Q}^i$ across tasks. The gradients to update weights in the outer loop are calculated w.r.t. the starting point $\boldsymbol{w}$ of the inner loop.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \beta \nabla_{\boldsymbol{w}} \sum_i \ell\left(\boldsymbol{w}^i; \mathcal{Q}^i\right) \tag{4.2}$$

where β is the outer step size.

Benchmark	4Conv MiniImageNet	ResNet12 MiniImageNet	MLP MuJoCo
Model Static Storage (MB)	0.13	32.0	0.05
Sample Static Storage (MB)	0.53	0.53	0.016(0.00008)
Inference Peak Memory (MB)	0.90	3.61	0.08(0.0004)
Training Peak Memory (MB)	48.33	370.44	3.72
Inference GFLOPs	0.72	62.08	0.05
Training GFLOPs	1.96	185.42	0.15

Table 4.2: Memory and total computation (GFLOPs = 10^9FLOPs) of inference and training in example few-shot learning. For image classification ("4Conv on MiniImageNet" and "ResNet12 on MiniImageNet"), we use batch size = 25, i.e., 5-way 5-shot. For robot locomotion ("MLP" on MuJoCo), we use rollouts = 20, horizon = 200; each sample corresponds to a rollouted episode, and the case for an observation is reported in brackets. The calculation is based on Section 4.5.

The meta-trained weights $\boldsymbol{w}^{\text{meta}}$ are then used as initialization for few-shot learning into $\boldsymbol{w}^{\text{new}}$ by K gradient descent steps over $\mathcal{S}^{\text{new}}$. Finally we assess the accuracy of $F(\boldsymbol{w}^{\text{new}})$ on $\boldsymbol{Q}^{\text{new}}$.

4.3.3 Memory Bottleneck of On-Device Learning

As mentioned above, the meta-trained model $F(\boldsymbol{w}^{\text{meta}})$ can learn unseen tasks via K gradient descent steps. Each step is the same as the inner loop of meta-training Eq. (4.1), but on dataset $\mathcal{S}^{\text{new}}$.

$$\boldsymbol{w}^{\text{new},k} = \boldsymbol{w}^{\text{new},k-1} - \alpha \nabla_{\boldsymbol{w}^{\text{new}}} \ell\left(\boldsymbol{w}^{\text{new},k-1}; \mathcal{S}^{\text{new}}\right) \tag{4.3}$$

where $\boldsymbol{w}^{\text{new},0} = \boldsymbol{w}^{\text{meta}}$. For brevity, we omit the superscripts of model adaption in Eq. (4.3) and use $\boldsymbol{g}(\cdot)$ as the loss gradients w.r.t. the given tensor. Hence, without ambiguity, we simplify the notations of Eq. (4.3) as follows:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \boldsymbol{g}(\boldsymbol{w}) \tag{4.4}$$

Let us now understand where the main memory cost for iterating Eq. (4.4) comes from. For the sake of clarity, we focus on a feed forward DNNs that consist of L convolutional (`conv`) layers or fully-connected (`fc`) layers. A typical layer (see Figure 4.2) consists of two operations: (*i*) a linear operation with trainable parameters, e.g., convolution or affine; (*ii*) a parameter-free non-linear operation, where we consider max-pooling or ReLU-styled (ReLU, LeakyReLU) activation functions in this paper. Note that the non-linear operation unit may not exist in some layers; some layers may also have more than one non-linear units (e.g., both max-pooling and ReLU activation function), and all corresponding intermediate tensors should be stored.

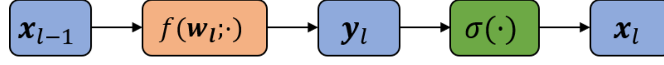

Figure 4.2: A typical layer l in DNNs. $\boldsymbol{x}_{l-1}$ is the input tensor; $\boldsymbol{x}_l$ is the output tensor, also the input tensor of layer $l+1$; $\boldsymbol{y}_l$ is the intermediate tensor; $\boldsymbol{w}_l$ is the weight tensor.

Take a network consisting of `conv` layers only as an example. The memory requirements for storing the activations $\boldsymbol{x}_l \in \mathbb{R}^{C_l \times H_l \times W_l}$ as well as the convolution weights $\boldsymbol{w}_l \in \mathbb{R}^{C_l \times C_{l-1} \times S_l \times S_l}$ of layer l in words can be determined as

$$m(\boldsymbol{x}_l) = C_l H_l W_l \ , \quad m(\boldsymbol{w}_l) = C_l C_{l-1} S_l^2$$

where C_{l-1}, C_l, H_l, and W_l stand for input channel number, output channel number, height and width of layer l, respectively; S_l stands for the kernel size. The detailed memory and computation demand analysis as provided in Section 4.5 reveals that the by far largest memory requirement is neither attributed to determining the activations $\boldsymbol{x}_l$ in the forward path nor to determining the gradients of the activations $\boldsymbol{g}(\boldsymbol{x}_l)$ in the backward path. Instead, the memory bottleneck lies in the computation of the weight gradients $\boldsymbol{g}(\boldsymbol{w}_l)$, which requires the availability of the activations $\boldsymbol{x}_{l-1}$ from the forward path. Following Eq. (4.20) in Section 4.5, the necessary memory in words is

$$\sum_{1 \le l \le L} m(\boldsymbol{x}_{l-1}) \tag{4.5}$$

Table 4.2 summarizes the peak memory and the total computation of the commonly used few-shot learning backbone models [FAL17, ORL18]. The requirements are based on the detailed analysis in Section 4.5. We can draw two intermediate conclusions.

- The total computation of training is approximately 2.7× to 3× larger compared to inference. Yet the peak memory of training is far larger, 47× to 103× over inference.

- To enable training on memory-constrained IoT devices, we need to find some way of getting rid of the major dynamic memory contribution in Eq. (4.5).

4.4 p-Meta

This section presents p-Meta, a new meta learning scheme that enables memory-efficient few-shot learning on unseen tasks. p-Meta is a novel meta training algorithm that not only learns the weights of the initialized backbone but also learns to identify adaptation-critical weights for memory-efficient few-shot learning.

4.4.1 p-Meta Overview

We first provide an overview of p-Meta and introduce its main concepts, namely selecting critical gradients, using a hierarchical approach to determine adaption-critical layers and channels, and using a mixture of static and dynamic selection mechanisms.

We impose *structured sparsity* on the *gradients* $\boldsymbol{g}(\boldsymbol{w}_l)$ such that the corresponding tensor dimensions of $\boldsymbol{x}_l$ do not need to be saved. There are other options to reduce the dominant memory demand in Eq. (4.5). They are inapplicable for the reasons below.

- One may trade-off computation and memory by recomputing activations $\boldsymbol{x}_{l-1}$ when needed for determining $\boldsymbol{w}_l$, see for example [CXZG16, GMD+16]. Due to the limited processing abilities of IoT devices, we exclude this option.

- It is also possible to prune activations $\boldsymbol{x}_{l-1}$. Yet based on our experimental results in Table 4.7, imposing sparsity on $\boldsymbol{x}_{l-1}$ hugely degrades few-shot learning accuracy as this causes error accumulation along the propagation, see also [RA20].

- Note that unstructured sparsity, as proposed in [GHZ+21, VOZK+21], does not in general lead to memory savings, since there is a very small probability that all weight gradients for which an element of $\boldsymbol{x}_{l-1}$ is necessary have been pruned. Furthermore, their weight selection is fixed after meta training, whereas p-Meta allows dynamic weight selection when few-shot learning on different tasks. Such runtime weight selection is essential for few-shot model training.

We impose sparsity on the gradients in a hierarchical manner.

- Selecting Adaption-Critical Layers: We first impose layer-by-layer sparsity on $\boldsymbol{g}(\boldsymbol{w}_l)$. It is motivated by previous results showing that manual freezing of certain layers does no harm to few-shot learning accuracy [RRBV20, OYKY21]. Layer-wise sparsity reduces the number of layers whose weights need to be updated. We determine the adaptation-critical layers from the meta-trained *layer-wise sparse learning rates*.

- Selecting Adaption-Critical Channels: In addition to imposing layer-wise sparsity of weight gradients, We further reduce the memory demand by imposing sparsity on $\boldsymbol{g}(\boldsymbol{w}_l)$ within each layer. Noting that calculating $\boldsymbol{g}(\boldsymbol{w}_l)$ needs both the input channels $\boldsymbol{x}_{l-1}$ and the output channels $\boldsymbol{g}(\boldsymbol{y}_l)$, we enforce sparsity on both of them. Input channel sparsity decreases memory and computation overhead, whereas output channel sparsity improves few-shot

learning accuracy and reduces computation. We design a novel *meta attention mechanism* to *dynamically* determine adaptation-critical channels. They take as inputs $\boldsymbol{x}_{l-1}$ and $\boldsymbol{g}(\boldsymbol{y}_l)$ and determine adaptation-critical channels during few-shot learning, based on the given data samples from new unseen tasks. Dynamic channel-wise learning rates as determined by meta attention yield a significant higher accuracy than static channel-wise learning rate (see Section 4.6.5).

Memory Reduction. The reduced memory demand due to our hierarchical approach can be seen in Eq. (4.20) in Section 4.5:

$$\sum_{1 \leq l \leq L} \hat{\alpha}_l \mu_l^{\text{fw}} m(\boldsymbol{x}_{l-1}) \tag{4.6}$$

where $\hat{\alpha}_l \in \{0, 1\}$ is the mask from the static selection of critical layers and $0 \leq \mu_l^{\text{fw}} \leq 1$ denotes the relative amount of dynamically chosen input channels.

Next, we explain how p-Meta selects adaptation-critical layers (Section 4.4.2) and channels within layers (Section 4.4.3) as well as the deployment optimizations (Section 4.4.4) for memory-efficient training.

4.4.2 Selecting Adaption-Critical Layers by Learning Sparse Inner Step Sizes

This subsection introduces how p-Meta meta-learns adaptation-critical layers to reduce the number of updated layers during few-shot learning. Particularly, instead of manual configuration as in [OYKY21, RRBV20], we propose to automate the layer selection process. During meta training, we identify adaptation-critical layers by learning layer-wise sparse inner step sizes (Section 4.4.2.1). Only these critical layers with nonzero step sizes will be updated during on-device learning to new tasks (Section 4.4.2.2).

4.4.2.1 Learning Sparse Inner Step Sizes in Meta Training

Prior work [AES19] suggests that instead of a global fixed inner step size α, learning the inner step sizes $\boldsymbol{\alpha}$ for each layer and each gradient descent step improves the generalization of meta learning, where $\boldsymbol{\alpha} = \alpha_{1:L}^{1:K} \geq \mathbf{0}$. We utilize such learned inner step sizes to infer layer importance for adaptation. We learn the inner step sizes $\boldsymbol{\alpha}$ in the outer loop of meta-training while fixing them in the inner loop.

Learning Layer-wise Inner Step Sizes. We change the inner loop of Eq. (4.1) to incorporate the per-layer inner step sizes:

$$\boldsymbol{w}_l^{i,k} = \boldsymbol{w}_l^{i,k-1} - \alpha_l^k \nabla_{\boldsymbol{w}_l} \ell \left(\boldsymbol{w}_{1:L}^{i,k-1}; \mathcal{S}^i \right) \tag{4.7}$$

where $\boldsymbol{w}_l^{i,k}$ is the weights of layer l at step k optimized on task i (dataset $\mathcal{S}^i$). In the outer loop, weights $\boldsymbol{w}$ are still optimized as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \beta \nabla_{\boldsymbol{w}} \sum_i \ell\left(\boldsymbol{w}^i; \mathcal{Q}^i\right) \tag{4.8}$$

where $\boldsymbol{w}^i = \boldsymbol{w}^{i,K} = \boldsymbol{w}_{1:L}^{i,K}$, which is a function of $\boldsymbol{\alpha}$. The inner step sizes $\boldsymbol{\alpha}$ are then optimized as

$$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \beta \nabla_{\boldsymbol{\alpha}} \sum_i \ell\left(\boldsymbol{w}^i; \mathcal{Q}^i\right) \tag{4.9}$$

Imposing Sparsity on Inner Step Sizes. To facilitate layer selection, we enforce sparsity in $\boldsymbol{\alpha}$, i.e., encouraging a subset of layers to be selected for updating. Specifically, we add a Lasso regularization term in the loss function of Eq. (4.9) when optimizing $\boldsymbol{\alpha}$. Hence, the final optimization of $\boldsymbol{\alpha}$ in the outer loop is formulated as

$$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \beta \nabla_{\boldsymbol{\alpha}} (\sum_i \ell\left(\boldsymbol{w}^i; \mathcal{Q}^i\right) + \lambda \sum_{l,k} m(\boldsymbol{x}_{l-1}) \cdot |\alpha_l^k|) \tag{4.10}$$

where λ is a positive scalar to control the ratio between two terms in the loss function. We empirically set $\lambda = 0.001$. $|\alpha_l^k|$ is re-weighted by $m(\boldsymbol{x}_{l-1})$, which denotes the necessary memory in Eq. (4.5) if only updating the weights in layer l.

4.4.2.2 Exploiting Sparse Inner Step Sizes for on-device learning

We now explain how to apply the learned $\boldsymbol{\alpha}$ to save memory during on-device learning. After deploying the meta-trained model to IoT devices for few-shot learning, at updating step k, for layers with $\alpha_l^k = 0$, the activations (i.e., their inputs) $\boldsymbol{x}_{l-1}$ need not be stored, see Eq. (4.19) and Eq. (4.20) in Section 4.5. In addition, we do not need to calculate the corresponding weight gradients $\boldsymbol{g}(\boldsymbol{w}_l)$, which saves computation, see Eq. (4.21) in Section 4.5.

4.4.3 Selecting Adaption-Critical Channels within Layers via Sparse Meta Attention

This subsection explains how p-Meta learns a novel meta attention mechanism in each layer to dynamically select adaptation-critical channels for further memory saving in few-shot learning. Despite the widespread adoption of channel-wise attention for inference [HSS18, CDL+20], we make the first attempt to use attention for memory-efficient training (few-shot learning in our case). For each layer, its meta attention outputs a dynamic channel-wise sparse attention score based on the

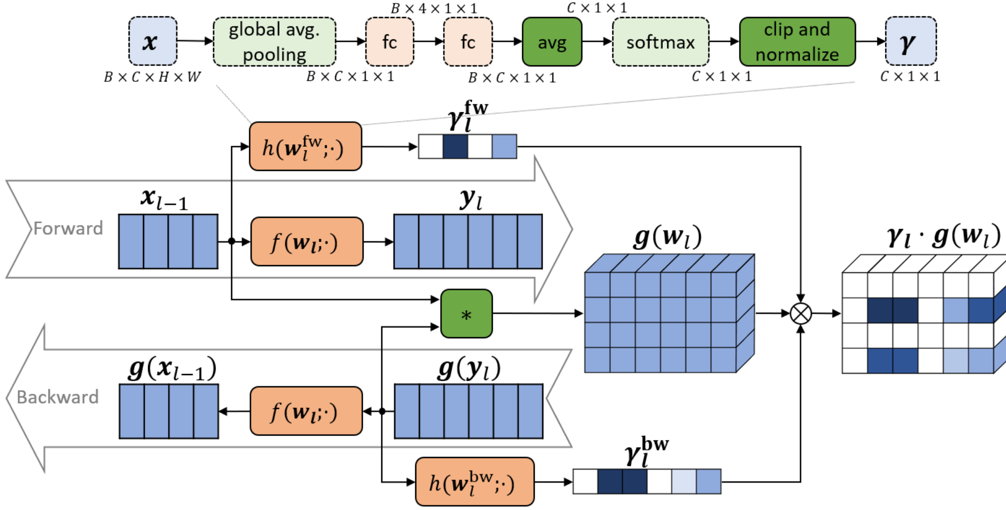

Figure 4.3: Meta attention of layer l during meta-training. The blue blocks correspond to tensors; the orange blocks correspond to computation units with parameters, and the green ones without. Each column of a tensor corresponds to one channel. The input tensor $\boldsymbol{x}_{l-1}$ has 4 channels; the output tensor $\boldsymbol{y}_l$ has 6 channels. The other dimensions (e.g., height, width and batch) are omitted here. The green block with $*$ stands for the operations involved to compute $\boldsymbol{g}(\boldsymbol{w}_l)$. In order to compute the gradients of the parameters in meta attention, i.e., $\boldsymbol{w}_l^{\text{fw}}$ and $\boldsymbol{w}_l^{\text{bw}}$, the full dense gradients $\boldsymbol{g}(\boldsymbol{w}_l)$ are computed during meta-training, and then are masked by γ_l. An example meta attention module for a conv layer is shown in the upper part. B denotes the batch size. The newly added blocks related to the inference attention in [CDL+20] are marked with solid lines.

samples from new tasks. The sparse attention score is used to re-weight (also sparsify) the weight gradients. Therefore, by calculating only the nonzero gradients of critical weights within a layer, we can save both memory and computation. We first present our meta attention mechanism during meta training (Section 4.4.3.1) and then show its usage for on-device model training (Section 4.4.3.2).

4.4.3.1 Learning Sparse Meta Attention in Meta Training

Since mainstream backbones in meta learning use small kernel sizes (1 or 3), we design the meta attention mechanism channel-wise. Figure 4.3 illustrates the attention design during meta-training.

Learning Meta Attention. The attention mechanism is as follows.

- We assign an attention score to the weight gradients of layer l in the inner loop of meta training. The attention scores are expected to indicate which weights/channels are important and thus should be updated in layer l.

- The attention score is obtained from two attention modules: one taking $\boldsymbol{x}_{l-1}$ as input in the forward pass, and the other taking $\boldsymbol{g}(\boldsymbol{y}_l)$ as input during the backward pass. We use $\boldsymbol{x}_{l-1}$ and $\boldsymbol{g}(\boldsymbol{y}_l)$ to calculate the attention scores because they are used to compute the weight gradients $\boldsymbol{g}(\boldsymbol{w}_l)$.

Concretely, we define the forward and backward attention scores for a conv layer as,

$$\gamma_l^{\text{fw}} = h(\boldsymbol{w}_l^{\text{fw}}; \boldsymbol{x}_{l-1}) \in \mathbb{R}^{C_{l-1}\times 1\times 1} \tag{4.11}$$

$$\gamma_l^{\text{bw}} = h(\boldsymbol{w}_l^{\text{bw}}; \boldsymbol{g}(\boldsymbol{y}_l)) \in \mathbb{R}^{C_l\times 1\times 1} \tag{4.12}$$

where $h(\cdot;\cdot)$ stands for the meta attention module, and $\boldsymbol{w}_l^{\text{fw}}$ and $\boldsymbol{w}_l^{\text{bw}}$ are the parameters of the meta attention modules. The overall (sparse) attention scores $\gamma_l \in \mathbb{R}^{C_l\times C_{l-1}\times 1\times 1}$ and is computed as,

$$\gamma_{l,ba11} = \gamma_{l,a11}^{\text{fw}} \cdot \gamma_{l,b11}^{\text{bw}} \tag{4.13}$$

In the inner loop, for layer l, step k and task i, γ_l is (broadcasting) multiplied with the dense weight gradients to get the sparse ones,

$$\gamma_l^{i,k} \odot \nabla_{\boldsymbol{w}_l} \ell\left(\boldsymbol{w}_{1:L}^{i,k-1}; \mathcal{S}^i\right) \tag{4.14}$$

The weights are then updated by,

$$\boldsymbol{w}_l^{i,k} = \boldsymbol{w}_l^{i,k-1} - \alpha_l^k(\gamma_l^{i,k} \odot \nabla_{\boldsymbol{w}_l} \ell\left(\boldsymbol{w}_{1:L}^{i,k-1}; \mathcal{S}^i\right)) \tag{4.15}$$

Let all attention parameters be $\boldsymbol{w}^{\text{atten}} = \{\boldsymbol{w}_l^{\text{fw}}, \boldsymbol{w}_l^{\text{bw}}\}_{l=1}^{L}$. The attention parameters $\boldsymbol{w}^{\text{atten}}$ are optimized in the outer loop as,

$$\boldsymbol{w}^{\text{atten}} \leftarrow \boldsymbol{w}^{\text{atten}} - \beta \nabla_{\boldsymbol{w}^{\text{atten}}} \sum_i \ell\left(\boldsymbol{w}^i; \mathcal{Q}^i\right) \tag{4.16}$$

Note that we use a dense forward path and a dense backward path in both meta-training and on-device learning, as shown in Figure 4.3. That is, the attention scores γ_l^{fw} and γ_l^{bw} are only calculated locally and will not affect $\boldsymbol{y}_l$ during forward and $\boldsymbol{g}(\boldsymbol{x}_{l-1})$ during backward. Based on our experimental results in Table 4.7, using either sparse $\boldsymbol{x}_{l-1}$ during forward or sparse $\boldsymbol{g}(\boldsymbol{y}_l)$ during backward will cause a dramatic performance degradation.

Meta Attention Module Design. Figure 4.3 (upper part) shows an example meta attention module. We adapt the inference attention modules used in [HSS18, CDL+20], yet with the following modifications.

- Unlike inference attention that applies to a single sample, training may calculate the averaged loss gradients based on a batch of samples. Since $\boldsymbol{g}(\boldsymbol{w}_l)$ does not have a batch dimension, the input to softmax function is first averaged over the batch data, see in Figure 4.3.

Algorithm 4.1: Clip and normalization

Input: softmax output (normalized) $\boldsymbol{\pi} \in \mathbb{R}^C$, clip ratio ρ
Output: sparse $\boldsymbol{\gamma}$
1 Sort $\boldsymbol{\pi}$ in ascending order and get sorted indices $d_{1:C}$;
2 Find the smallest c such that $\sum_{i=1}^{c} \pi_{d_i} \geq \rho$;
3 Set $\pi_{d_1:d_c}$ as 0 ; `// if` $\rho = 0$`, do nothing`
4 Normalize $\boldsymbol{\gamma} = \boldsymbol{\pi} / \sum \boldsymbol{\pi}$;
5 Re-scale $\boldsymbol{\gamma} = \boldsymbol{\gamma} \cdot C$; `// keeping step sizes' magnitude`

Algorithm 4.2: p-Meta

Input: meta-training task distribution $p(\mathsf{T})$, backbone F with initial weights $\boldsymbol{w}$, meta attention parameters $\boldsymbol{w}^{\text{atten}}$, inner step sizes $\boldsymbol{\alpha}$, outer step sizes β
Output: meta-trained weights $\boldsymbol{w}^{\text{meta}}$, meta attention parameters $\boldsymbol{w}^{\text{atten}}$, sparse inner step sizes $\boldsymbol{\alpha}$
1 **while** *not done* **do**
2   Sample a batch of I tasks $\mathsf{T}^i \sim p(\mathsf{T})$;
3   **for** $i \leftarrow 1$ **to** I **do**
4   Update $\boldsymbol{w}^i$ in K gradient descent steps with (4.15);
5   Update $\boldsymbol{w}$ with (4.8);
6   Update inner step sizes $\boldsymbol{\alpha}$ with (4.10);
7   Update attention parameters $\boldsymbol{w}^{\text{atten}}$ with Eq. (4.16);

- We enforce sparsity on the meta attention scores such that they can be utilized to save memory and computation in few-shot learning. The original attention in [HSS18, CDL+20] outputs normalized scales in $[0, 1]$ from softmax. We clip the output with a clip ratio $\rho \in [0, 1]$ to create zeros in $\boldsymbol{\gamma}$. This way, our meta attention modules yield batch-averaged sparse attention scores γ_l^{fw} and γ_l^{bw}. Algorithm 4.1 shows this clipping and re-normalization process. Note that Algorithm 4.1 is not differentiable. Hence we use the straight-through-estimator for its backward propagation in meta training.

Algorithm 4.2 shows the overall process of p-Meta during meta training.

4.4.3.2 Exploiting Meta Attention for on-device learning

We now explain how to apply the meta attention to save memory during on-device few-shot learning. Note that the parameters in the meta attention modules are fixed during few-shot learning. Assume that at step k, layer l has a nonzero step size α_l^k. In the forward pass, we only store a sparse tensor $\gamma_l^{\text{fw}} \cdot \boldsymbol{x}_{l-1}$, i.e., its channels are stored only if they

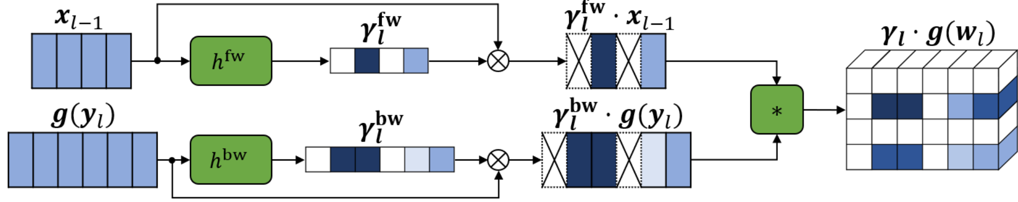

Figure 4.4: Meta attention of layer l during on-device few-shot learning. Note that "Forward" part and "Backward" part are the same as Figure 4.3, which are omitted for simplicity. Meta attention modules are not optimized during few-shot learning, thus are expressed as parameter-free functions h^{fw} and h^{bw}. The input $\boldsymbol{x}_{l-1}$ stored during forward path is a sparse re-weighted tensor.

correspond to nonzero entries in γ_l^{fw}. This reduces memory consumption as shown in Eq. (4.20) in Section 4.5. Similarly, in the backward pass, we get a channel-wise sparse tensor $\gamma_l^{\mathrm{bw}} \cdot \boldsymbol{g}(\boldsymbol{y}_l)$. Since both sparse tensors are used to calculate the corresponding nonzero gradients in $\boldsymbol{g}(\boldsymbol{w}_l)$, the computation cost is also reduced, see Eq. (4.21) in Section 4.5. We plot the meta attention during on-device learning in Figure 4.4.

4.4.4 Deployment Optimization

To further reduce the memory during few-shot learning, we propose gradient accumulation during backpropagation and replace batch normalization in the backbone with group normalization.

4.4.4.1 Gradient Accumulation

In standard few-shot learning, all the new samples (e.g., 25 for 5-way 5-shot) are fed into the model as one batch. To reduce the peak memory due to large batch sizes, we conduct few-shot learning with gradient accumulation (GA).

GA is a technique that (*i*) breaks a large batch into smaller partial batches; (*ii*) sequentially forward/backward propagates each partial batches through the model; (*iii*) accumulates the loss gradients of each partial batch and get the final averaged gradients of the full batch. Note that GA does not increase computation, which is desired for low-resource platforms with constrained memory and limited parallelism. Accordingly, our meta attention module should be also modified. Particularly, the input to the softmax is averaged over all samples in the batch (see Figure 4.3), i.e., γ_l^{fw} and γ_l^{bw} are the batch-averaged scores. We evaluate the impact of different sample batch sizes in GA in Section 4.6.7.

4.4.4.2 Group Normalization

Mainstream backbones in meta learning typically adopt batch normalization layers. Batch normalization layers compute the statistical information in each batch, which is dependent on the sample batch size. When using GA with different sample batch sizes, the inaccurate batch statistics can degrade the training performance (see Section 4.6.6). As a remedy, we use group normalization [WH18], which does not rely on batch statistics (i.e., independent of the sample batch size). We also apply meta attention on group normalization layers when updating their weights. The only difference w.r.t. `conv` and `fc` layers is that the stored input tensor (also the one used for the meta attention) is not $\boldsymbol{x}_{l-1}$, but its normalized version.

4.5 Theoretical Analysis on Memory and Computation

In this section, we derive the memory requirement and computation workload for inference and training. We further analyze the reduced consumption of memory and computation due to p-Meta.

Recall that we focus on a feed forward DNN that consists of L convolutional (`conv`) layers or fully-connected (`fc`) layers. Note that our analysis focuses on 2D `conv` layers but can apply to other `conv` layer types as well. We assume the ReLU activation function for all layers, denoted as $\sigma(\cdot)$. For simplicity, we omit the bias, normalization layers, pooling or strides. We use the notation $m(\boldsymbol{x})$ to denote the memory demand in words to store tensor $\boldsymbol{x}$. The wordlength is denoted as T.

For representing indexed summations we use the Einstein notation. If index variables appear in a term on the right hand side of an equation and are not otherwise defined (free indices), it implies summation of that term over the range of the free indices. If indices of involved tensor elements are out of range, the values of these elements are assumed to be 0.

4.5.1 Single Layer

We start with a single layer and accumulate the memory and computation for networks with several layers afterwards. Assume the input tensor of a layer is $\boldsymbol{x}$, the weight tensor is $\boldsymbol{w}$, the result after the linear transformation is $\boldsymbol{y}$, and the layer output after the non-linear operator is $\boldsymbol{z}$ which is also the input to the next layer.

For convolutional layers, we have $\boldsymbol{x} \in \mathbb{R}^{C_I \times H_I \times W_I}$ and elements x_{cij}, where C_I, H_I, and W_I denote the number of input channels, height and width, respectively. In a similar way, we have $\boldsymbol{z} \in \mathbb{R}^{C_O \times H_O \times W_O}$ with elements x_{fij} where C_O, H_O, and W_O denote the number of output channels, height and width, respectively. Moreover, $\boldsymbol{w} \in \mathbb{R}^{C_O \times C_I \times S \times S}$ with

elements w_{fcmn}. Therefore,

$$m(\boldsymbol{x}) = C_I H_I W_I \ , \ \ m(\boldsymbol{y}) = m(\boldsymbol{z}) = C_O H_O W_O \ , \ \ m(\boldsymbol{w}) = C_O C_I S^2$$

For fully connected layers we have $\boldsymbol{x} \in \mathbb{R}^{C_I}$, $\boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^{C_O}$, and $\boldsymbol{w} \in \mathbb{R}^{C_O \times C_I}$ with memory demand

$$m(\boldsymbol{x}) = C_I \ , \ \ m(\boldsymbol{y}) = m(\boldsymbol{z}) = C_O \ , \ \ m(\boldsymbol{w}) = C_O C_I$$

4.5.1.1 Fully Connected Layer

For inference we derive the relations $y_f = w_{fc} x_c$ and $z_f = \sigma(y_f)$ for all admissible indices $f \in [1, C_O]$. The necessary dynamic memory has a size of about $m(\boldsymbol{x}) + m(\boldsymbol{y})$ words and we need about $m(\boldsymbol{w})$ FLOPs.

For training, we suppose that $\frac{\partial \ell}{\partial z_i}$ is already provided from the next layer. We find $\frac{\partial \ell}{\partial y_i} = \sigma'(y_i) \cdot \frac{\partial \ell}{\partial z_i}$ with $\sigma'(y_i) = \begin{cases} 1 & \text{if } y_i > 0 \\ 0 & \text{if } y_i < 0 \end{cases}$ which leads to $\frac{\partial \ell}{\partial x_i} = w_{ji} \cdot \frac{\partial \ell}{\partial y_j}$. The necessary dynamic memory is about $m(\boldsymbol{x}) + m(\boldsymbol{y}) \cdot (1 + \frac{1}{T})$ words, where the last term comes from storing $\sigma'(y_i)$ single bits from the forward path. We need about $m(\boldsymbol{w})$ FLOPs.

According to the approach described in the paper we are only interested in the partial derivatives $\frac{\partial \ell}{\partial w_{fc}}$ if $\alpha > 0$ for this layer, and if scales $\gamma_f^{\text{bw}} > 0$ and $\gamma_c^{\text{fw}} > 0$ for indices f, c. To simplify the notation, let us define the critical ratios

$$\mu^{\text{fw}} = \frac{\text{number of nonzero elements of } \gamma_c^{\text{fw}}}{C_I} \tag{4.17}$$

$$\mu^{\text{bw}} = \frac{\text{number of nonzero elements of } \gamma_f^{\text{bw}}}{C_O} \tag{4.18}$$

which are 1 if all channels are determined to be critical for weight adaptation, and 0 if none of them.

We find $\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial w_{fc}} \gamma_c^{\text{fw}} = (\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial y_f}) \cdot (\gamma_c^{\text{fw}} x_c)$. Therefore, we need $\mu^{\text{fw}} \mu^{\text{bw}} m(\boldsymbol{w}) + \mu^{\text{fw}} m(\boldsymbol{x})$ words dynamic memory if $\alpha > 0$ where the latter term considers the information needed from the forward path. We require about $\mu^{\text{fw}} \mu^{\text{bw}} m(\boldsymbol{w})$ FLOPs if $\alpha > 0$.

4.5.1.2 Convolutional Layer

The memory analysis for a convolutional layer is very similar, just replacing matrix multiplication by convolution. For inference we find $y_{fij} = w_{fcmn} x_{c,i+m-1,j+n-1}$ and $z_{fij} = \sigma(y_{fij})$ for all admissible indices f, i, j. The necessary dynamic memory has a size of about $\max\{m(\boldsymbol{x}), m(\boldsymbol{y})\}$ words when using memory sharing between input and output tensors. We need about $H_O W_O \cdot m(\boldsymbol{w})$ FLOPs.

For training, we again suppose that $\frac{\partial \ell}{\partial z_{fij}}$ is provided from the next layer. We find $\frac{\partial \ell}{\partial y_{fij}} = \sigma'(y_{fij}) \cdot \frac{\partial \ell}{\partial z_{fij}}$ and get $\frac{\partial \ell}{\partial x_{cij}} = w_{fcmn} \cdot \frac{\partial \ell}{\partial y_{f,i+m-1,j+n-1}}$. The necessary memory is about $\max\{m(\boldsymbol{x}), m(\boldsymbol{y})\} + \frac{m(\boldsymbol{y})}{T}$ words, where the last term comes from storing $\sigma'(y_{fij})$ single bits from the forward path. We need about $H_I W_I \cdot m(\boldsymbol{w})$ multiply and accumulate operations.

For determining the weight gradients we find $\frac{\partial \ell}{\partial w_{fcmn}} = \frac{\partial \ell}{\partial y_{fij}} \cdot x_{c,i+m-1,j+n-1}$. When considering the scales for filtering, we yield $\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial w_{fcmn}} \gamma_c^{\text{fw}} = (\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial y_{fij}}) \cdot (\gamma_c^{\text{fw}} x_{c,i+m-1,j+n-1})$. As a result, we need $\mu^{\text{fw}} \mu^{\text{bw}} m(\boldsymbol{w}) + \mu^{\text{fw}} m(\boldsymbol{x})$ words of dynamic memory if $\alpha > 0$ where the latter term considers the information needed from the forward path. We require about $\mu^{\text{fw}} \mu^{\text{bw}} H_O W_O m(\boldsymbol{w})$ FLOPs if $\alpha > 0$.

Finally, let us determine the required memory and computation to determine the scales γ_c^{fw} and γ_f^{bw}. According to Figure 4.3, we find as an upper bound for the memory $B \cdot (C_I + C_O)$ and $(C_I H_I W_I + 2C_I^2 + C_O H_O W_O + 2C_O^2)$ FLOPs.

4.5.2 All Layers

The above relations are valid for a single layer. The following relations hold for the overall network. In order to simplify the notation, we consider a network that consists of convolution layers only. Extensions to mixed layers can simply be done.

We suppose L layers with sizes C_l, H_l, W_l and S_l for the number of output channels, output width, output height and kernel size, respectively. We assume that the step-sizes α_l for some iteration of the training are given. The memory requirement in words is

$$m(\boldsymbol{x}_l) = C_l H_l W_l \ , \quad m(\boldsymbol{w}_l) = C_l C_{l-1} S_l^2$$

and the word-length is again denoted as T. We define as $\hat{\alpha}_l = \begin{cases} 1 & \text{if } \alpha_l > 0 \\ 0 & \text{if } \alpha_l = 0 \end{cases}$ the mask that determines whether the weight adaptation for this layer is necessary or not.

Let us first look at the forward path. The necessary dynamic memory is about $\max_{0 \le l \le L}\{m(\boldsymbol{x}_l)\}$ words. The number of FLOPs is $\sum_{1 \le l \le L} H_l W_l m(\boldsymbol{w}_l)$.

The backward path needs only to be evaluated until we reach the first layer where we require the computation of the gradients. We define $l_{min} = \min\{l \,|\, \hat{\alpha}_l = 1\}$. For the calculation of the partial derivatives of the activations we need dynamic memory of $\max_{l_{min} \le l \le L}\{m(\boldsymbol{x}_l)\} + \frac{1}{T} \sum_{l_{min} \le l \le L} m(\boldsymbol{x}_l)$ words where the last term is due to storing the derivatives of the ReLU operations. We need about $\sum_{l_{min}+1 \le l \le L} H_{l-1} W_{l-1} m(\boldsymbol{w}_l)$ FLOPs.

The second contribution of the backward path is for computing the weight gradients. The memory and computation demand of the scales will be neglected as they are much smaller than

other contributions. We can determine the necessary dynamic memory as $\max_{1\leq l\leq L}\{\hat{\alpha}_l\mu_l^{\text{fw}}\mu_l^{\text{bw}}m(\boldsymbol{w}_l)\} + \sum_{1\leq l\leq L}\hat{\alpha}_l\mu_l^{\text{fw}}m(\boldsymbol{x}_{l-1})$, and we need $\sum_{1\leq l\leq L}\hat{\alpha}_l\mu_l^{\text{fw}}\mu_l^{\text{bw}}H_lW_lm(\boldsymbol{w}_l)$ FLOPs.

Considering all necessary dynamic memory with memory reuse for a gradient-based training step, we get an estimation of memory in words

$$\max_{0\leq l\leq L}\{m(\boldsymbol{x}_l)\} + \sum_{1\leq l\leq L}\hat{\alpha}_l m(\boldsymbol{w}_l) + \sum_{1\leq l\leq L}\hat{\alpha}_l\mu_l^{\text{fw}}m(\boldsymbol{x}_{l-1}) + \frac{1}{T}\sum_{l_{min}\leq l\leq L}m(\boldsymbol{x}_l) \quad (4.19)$$

if we accumulate the weight gradients before doing an SGD step and re-use some memory during back-propagation. More elaborate memory re-use can be used to slightly sharpen the bounds without a major improvement. For conventional training, each parameter is in 32-bit floating point format, i.e., one word corresponds to 32-bit. As discussed in Section 4.3.3, we only consider max-pooling and ReLU-styled activation as the σ function. The wordlength T in Eq. (4.19) is set as 16 for max-pooling , and 32 for ReLU-styled activation. One can see that under the typical assumptions for network parameters, the above memory requirement in words is dominated by

$$\sum_{1\leq l\leq L}\hat{\alpha}_l\mu_l^{\text{fw}}m(\boldsymbol{x}_{l-1}) \quad (4.20)$$

The necessary storage between the forward and backward path is reduced proportionally to μ_l^{fw} with factor $m(\boldsymbol{x}_{l-1})$.

Finally, the amount of FLOPs can be estimated as

$$\sum_{1\leq l\leq L}H_lW_lm(\boldsymbol{w}_l)(1+\hat{\alpha}_l\mu_l^{\text{fw}}\mu_l^{\text{bw}}) + \sum_{l_{min}\leq l\leq L}H_{l-1}W_{l-1}m(\boldsymbol{w}_l) \quad (4.21)$$

while neglecting lower order terms. Here it is important to note that all terms are of similar order. The approach used in the paper does not determine a trade-off between computation and memory, but reduces the amount of FLOPs. This reduction is less than the reduction in required dynamic memory.

4.6 Experiments

This section presents the evaluations of p-Meta on standard few-shot image classification and reinforcement learning benchmarks.

4.6.1 General Experimental Settings

Compared Methods. We test the meta learning algorithms below.

- MAML [FAL17]: the original model-agnostic meta learning.

- ANIL [RRBV20]: update the last layer only in few-shot learning.
- BOIL [OYKY21]: update the body except the last layer.
- MAML++ [AES19]: learn a per-step per-layer step sizes $\boldsymbol{\alpha}$.
- p-Meta (4.4.2): can be regarded as a sparse version of MAML++, since it learns a sparse $\boldsymbol{\alpha}$ with our methods in Section 4.4.2.
- p-Meta (4.4.2+4.4.3): the full version of our methods which include the meta attention modules in Section 4.4.3.

For fair comparison, all the algorithms are re-implemented with the deployment optimization in Section 4.4.4.

Implementation. The experiments are conducted with tools provided by TorchMeta [DWS+19, Del18]. Particularly, the backbone is meta-trained with full sample batch size (e.g., 25 for 5-way 5-shot) on meta training dataset. After each meta training epoch, the model is tested (i.e., few-shot learned) on meta validation dataset. The model with the highest validation performance is used to report the final few-shot learning results on meta test dataset. We follow the same process as TorchMeta [DWS+19, Del18] to build the dataset. During few-shot learning, we adopt a sample batch size of 1 to verify the model performance under the most strict memory constraints.

In p-Meta, meta attention is applied to all `conv`, `fc`, and group normalization layers, expect the last output layer, because (*i*) we find modifying the last layer's gradients may decrease accuracy; (*ii*) the final output is often rather small in size, resulting in little memory saving even if imposing sparsity on the last layer. Without further notations, we set $\rho = 0.3$ in forward attention, and $\rho = 0$ in backward attention across all layers, as the sparsity of γ_l^{bw} almost has no effect on the memory saving.

Metrics. We compare the peak memory and FLOPs of different algorithms. Note that the reported peak memory and FLOPs for p-Meta also include the consumption from meta attention, although they are rather small related to the backward propagation.

4.6.2 Benchmarking Details

4.6.2.1 4Conv/ResNet12 on MiniImageNet/TieredImageNet/CUB

MiniImageNet [VBL+16] is an image classification dataset from ImageNet dataset [RDS+15], which consists of 84×84 color images in 100 classes. Following the splitting in [VBL+16], 64 classes are used for meta-training, 16 classes are used for meta-validation, and the rest 20 classes are used as unseen tasks for meta-testing (i.e., few-shot learning). We train on 1 Nvidia V100 GPU. We experiment in both 5-way 1-shot and 5-way 5-shot

settings. The task batch size is set to 4 in general, except for ResNet12 under 5-way 5-shot settings where we use 2.

TieredImageNet [RTR+18] is an image classification dataset from ImageNet dataset [RDS+15], which consists of 84×84 color images in 34 categories (608 classes). Following the splitting in [RTR+18], 20 categories (351 classes) are used for meta-training, 6 categories (97 classes) are used for meta-validation, and the rest 8 categories (160 classes) are used as unseen tasks for meta-testing (i.e., few-shot learning).

CUB [WBM+10] is an image classification dataset, which consists of 84×84 color images of bird species in 200 classes. Following the splitting in [DWS+19], 100 classes are used for meta-training, 50 classes are used for meta-validation, and the rest 50 classes are used as unseen tasks for meta-testing (i.e., few-shot learning).

4Conv. The "4Conv" [FAL17] backbone has 4 `conv` blocks. Each `conv` block includes a `conv` layer with 32 channels, a group normalization layer (as discussed in Section 4.4.4.2), a ReLU activation, and a max-pooling with stride 2.

ResNet12. The "ResNet12" [ORL18] backbone has 4 residual blocks with $\{64, 128, 256, 512\}$ channels in each block respectively. Each residual block consists of 3 `conv` blocks followed by max-pooling with stride 2. Each `conv` layer is followed by a group normalization layer and a LeakyReLU activation with slope 0.1. Refer to [ORL18] for more detailed structure.

4.6.2.2 MLP on MuJoCo

MuJoCo is an advanced simulator for multi-body dynamics with contact. For all experiments, we mainly adopt the experimental setup in [FAL17, Del18]. We run the MuJoCo environment as well as the policy model training on 8 CPUs.

MLP. We use a neural network as the policy model. The neural network is a MLP with two hidden `fc` layers of size 100 and the ReLU activation.

4.6.3 Experiments on Image Classification

Settings. We test on standard few-shot image classification tasks (both in-domain and cross-domain). We adopt two common backbones "4Conv" [FAL17] and "ResNet12" [ORL18]. The batch normalization layers are replaced with group normalization layers, as discussed in Section 4.4.4.2. We train the model on MiniImageNet [VBL+16] (both meta training and meta validation dataset) with 100 meta epochs. In each meta epoch, 1000 random tasks are drawn from the task distribution.

The model is updated with 5 gradient steps (i.e., $K = 5$) in both inner loop of meta-training and few-shot learning. We use Adam optimizer with cosine learning rate scheduling as [AES19] for all outer

5-Way 1-Shot		Accuracy			GFLOPs	Memory
Benchmarks		Mini	Tiered	CUB	Mini	Mini
4Conv	MAML [FAL17]	46.2%	51.4%	39.7%	0.39	2.06
	ANIL [RRBV20]	46.4%	51.5%	39.2%	0.14	0.92
	BOIL [OYKY21]	44.7%	51.3%	42.3%	0.39	2.05
	MAML++ [AES19]	48.2%	53.2%	**43.2%**	0.39	2.06
	p-Meta (4.4.2)	47.1%	52.3%	41.8%	0.16	1.00
	p-Meta (4.4.2+4.4.3)	**48.8%**	**53.9%**	42.6%	0.15	0.99
ResNet12	MAML [FAL17]	51.7%	57.4%	41.3%	37.08	54.69
	ANIL [RRBV20]	50.3%	56.7%	40.6%	12.42	3.62
	BOIL [OYKY21]	42.7%	47.7%	44.2%	37.08	54.69
	MAML++ [AES19]	53.1%	58.6%	45.1%	37.08	54.69
	p-Meta (4.4.2)	51.8%	58.3%	40.6%	25.84	17.66
	p-Meta (4.4.2+4.4.3)	**53.6%**	**59.4%**	**45.4%**	24.02	16.01

Table 4.3: 5-Way 1-shot few-shot image classification results on 4Conv and ResNet12. All methods are meta-trained on MiniImageNet, and are few-shot learned on the reported datasets: MiniImageNet, TieredImageNet, and CUB (denoted by Mini, Tiered, and CUB in the table). The total computation (# GFLOPs) and the peak memory (MB) during few-shot learning are reported based on the theoretical analysis in Section 4.5.

5-Way 5-Shot		Accuracy			GFLOPs	Memory
Benchmarks		Mini	Tiered	CUB	Mini	Mini
4Conv	MAML [FAL17]	61.4%	66.5%	55.6%	1.96	2.06
	ANIL [RRBV20]	60.6%	64.5%	54.2%	0.72	0.92
	BOIL [OYKY21]	60.5%	65.3%	58.3%	1.96	2.05
	MAML++ [AES19]	63.7%	**68.5%**	59.1%	1.96	2.06
	p-Meta (4.4.2)	62.9%	68.3%	59.3%	1.34	1.09
	p-Meta (4.4.2+4.4.3)	**65.0%**	**68.5%**	**60.2%**	1.11	1.04
ResNet12	MAML [FAL17]	64.7%	69.6%	53.8%	185.42	54.69
	ANIL [RRBV20]	62.3%	68.7%	54.0%	62.08	3.62
	BOIL [OYKY21]	53.6%	59.8%	53.7%	185.42	54.69
	MAML++ [AES19]	68.6%	**73.4%**	63.9%	185.42	54.69
	p-Meta (4.4.2)	68.8%	72.6%	65.9%	124.15	18.95
	p-Meta (4.4.2+4.4.3)	**69.7%**	73.3%	**66.6%**	116.79	17.17

Table 4.4: 5-Way 5-shot few-shot image classification results on 4Conv and ResNet12. All methods are meta-trained on MiniImageNet, and are few-shot learned on the reported datasets: MiniImageNet, TieredImageNet, and CUB (denoted by Mini, Tiered, and CUB in the table). The total computation (# GFLOPs) and the peak memory (MB) during few-shot learning are reported based on the theoretical analysis in Section 4.5.

loop updating. The (initial) inner step size α is set to 0.01. The meta-trained model is then tested on three datasets MiniImageNet [VBL+16], TieredImageNet [RTR+18], and CUB [WBM+10] to verify both *in-domain*

20 Rollouts	Half-Cheetah Velocity			2D Navigation		
Benchmarks	Return	GFLOPs	Memory	Return	GFLOPs	Memory
MAML [FAL17]	-82.2	0.15	0.24	-13.3	0.12	0.21
ANIL [RRBV20]	-78.8	0.06	0.09	-13.8	0.04	0.08
BOIL [OYKY21]	-76.4	0.15	0.23	-12.4	0.12	0.21
MAML++ [AES19]	-69.6	0.15	0.24	-17.6	0.12	0.21
p-Meta (4.4.2)	-65.5	0.11	0.12	**-11.2**	0.09	0.09
p-Meta (4.4.2+4.4.3)	**-64.0**	0.11	0.11	-11.8	0.09	0.09

Table 4.5: Few-shot reinforcement learning results on 2D navigation and robot locomotion tasks (larger return means better). A MLP with two hidden layers of size 100 is used as the policy model. The total computation (# GFLOPs) and the peak memory (MB) during few-shot learning are reported based on the theoretical analysis in Section 4.5.

and *cross-domain* performance.

Results. Table 4.3 and Table 4.4 show the accuracy of few-shot learned models for 5-way 1-shot and 5-way 5-shot scenarios respectively. The reported accuracy is averaged over 5000 new unseen tasks randomly drawn from the meta test dataset We also report the average number of GFLOPs and the average peak memory per task according to Section 4.5. Clearly, p-Meta almost always yields the highest accuracy in all settings. Note that the comparison between "p-Meta (4.4.2)" and "MAML++" can be considered as the ablation studies on learning sparse layer-wise inner step sizes proposed in Section 4.4.2. Thanks to the imposed sparsity on α, "p-Meta (4.4.2)" significantly reduces the peak memory (2.5× saving on average and up to 3.1×) and the computation burden (1.7× saving on average and up to 2.4×) over "MAML++". Note that the imposed sparsity also cause a moderate accuracy drop. However, with the meta attention, "p-Meta (4.4.2+4.4.3)" not only notably improves the accuracy but also further reduces the peak memory (2.7× saving on average and up to 3.4×) and computation (1.9× saving on average and up to 2.6×) over "MAML++". Note that "ANIL" only updates the last layer, and therefore consumes less memory but also yields a substantially lower accuracy.

4.6.4 Experiments on Reinforcement Learning

Settings. To show the versatility of p-Meta, we experiment with two few-shot reinforcement learning problems: 2D navigation and Half-Cheetah robot locomotion simulated with MuJoCo library [TET12]. We adopt vanilla policy gradient [Wil92] for the inner loop and trust-region policy optimization [SLM+15] for the outer loop. During the inner loop as well as few-shot learning, the agents rollout 20 episodes with a horizon size of 200 and are updated for one gradient step. The policy model is trained

for 500 meta epochs, and the model with the best average return during training is used for evaluation. The task batch size is set to 20 for 2D navigation, and 40 for robot locomotion. The (initial) inner step size α is set to 0.1. Each episode is considered as a data sample, and thus the gradients are accumulated 20 times for a gradient step.

Results. Table 4.5 lists the average return averaged over 400 new unseen tasks randomly drawn from simulated environments. We also report the average number of GFLOPs and the average peak memory per task according to Section 4.5. Note that the reported computation and peak memory do not include the estimations of the advantage [DCH+16], as they are relatively small and could be done during the rollout. p-Meta consumes a rather small amount of memory and computation, while often obtains the highest return in comparison to others. Therefore, p-Meta can fast adapt its policy to reach the new goal in the environment with less on-device resource demand.

4.6.5 Ablation Studies on Meta Attention

We study the effectiveness of our meta attention via the following two ablation studies. The experiments are conducted on "4Conv" in both 5-way 1-shot and 5-way 5-shot as Section 4.6.3.

Sparsity in Meta Attention. Table 4.6 shows the few-shot classification accuracy with different sparsity settings in the meta attention.

We first do not impose sparsity on γ_l^{fw} and γ_l^{bw} (i.e., set both ρ's as 0), and adopt forward attention and backward attention separately. In comparison to no meta attention at all, enabling either forward or backward attention improves accuracy. With both attention enabled, the model achieves the best performance.

We then test the effects when imposing sparsity on γ_l^{fw} or γ_l^{bw} (i.e., set $\rho > 0$). We use the same ρ for all layers. We observe a sparse γ_l^{bw} often cause a larger accuracy drop than a sparse γ_l^{fw}. Since a sparse γ_l^{bw} does not bring substantial memory or computation saving (see Section 4.5), we use $\rho = 0$ for backward attention and $\rho = 0.3$ for forward attention. Note that $\rho = 1$ means that the resulted γ_l are all zeros and the layers are not updated, which can be realized by imposing sparsity on layerwise learning rate in Section 4.4.2.

Attention scores γ_l introduce a dynamic channel-wise learning rate according to the new data samples. We further compare meta attention with a static channel-wise learning rate, where the channel-wise learning rate α^{Ch} is meta-trained as the layer-wise inner step sizes in Section 4.4.2 while without imposing sparsity. By comparing "α^{Ch}" with "0, 0" in Table 4.6, we conclude that the dynamic channel-wise learning rate yields a significantly higher accuracy.

Layer-wise Updating Ratios. To study the resulted updating ratios across

ρ		5-way 1-shot			5-way 5-shot		
fw	bw	Mini	Tiered	CUB	Mini	Tiered	CUB
x	x	47.1%	52.3%	41.8%	62.9%	68.3%	59.3%
0	x	48.1%	53.2%	41.7%	64.1%	68.4%	59.0%
x	0	47.8%	53.1%	40.9%	63.9%	68.5%	60.0%
0	0	**49.0%**	**54.2%**	**43.1%**	64.5%	**69.2%**	**60.2%**
0	0.3	48.5%	53.4%	42.2%	64.7%	68.2%	59.3%
0.3	0	48.8%	53.9%	42.6%	**65.0%**	68.5%	**60.2%**
0.3	0.3	48.7%	53.7%	42.3%	64.5%	68.3%	59.5%
0.5	0.5	48.2%	53.4%	42.7%	64.8%	68.1%	59.1%
$\boldsymbol{\alpha}^{\text{Ch}}$		47.8%	52.8%	41.0%	63.6%	68.1%	58.1%

x: no forward/backward meta attention, i.e., $\gamma_l^{\text{fw}} = 1$ or $\gamma_l^{\text{bw}} = 1$.
$\boldsymbol{\alpha}^{\text{Ch}}$: introduce an input- and output-channel inner step sizes $\boldsymbol{\alpha}^{\text{Ch}}$ per layer. We use $\boldsymbol{\alpha} \cdot \boldsymbol{\alpha}^{\text{Ch}}$ as inner step sizes. $\boldsymbol{\alpha}^{\text{Ch}}$ is meta-trained as $\boldsymbol{\alpha}$ while without sparsity.

Table 4.6: Ablation results of meta attention on 4Conv.

layers, i.e., the layer-wise sparsity of weight gradients, we randomly select 100 new tasks and plot the layer-wise updating ratios, see Figure 4.5. The "4Conv" backbone has 9 layers ($L = 9$), i.e., 8 alternates of `conv` and group normalization layers, and an `fc` output layer. As mentioned in Section 4.6.1, we do not apply meta attention to the output layer, i.e., $\gamma_9 = 1$. The used backbone is updated with 5 gradient steps ($K = 5$). We use $\rho = 0.3$ for forward attention, and $\rho = 0$ for backward. Note that Algorithm 4.1 adaptively determines the sparsity of γ_l, which also means different samples may result in different updating ratios even with the same ρ (see Figure 4.5). The size of $\boldsymbol{x}_{l-1}$ often decreases along the layers in current DNNs. As expected, the latter layers are preferred to be updated more, since they need a smaller amount of memory for updating. Interestingly, even if with a small $\rho(= 0.3)$, the ratio of updated weights is rather small, e.g., smaller than 0.2 in step 3 of 5-way 5-shot. It implies that the outputs of softmax have a large discrepancy, i.e., only a few channels are adaptation-critical for each sample, which in turn verifies the effectiveness of our meta attention mechanism.

We also randomly pair data samples and compute the cosine similarity between their attention scores γ_l. We plot the cosine similarity of step 1 in Figure 4.6. The results show that there may exist a considerable variation on the adaptation-critical weights selected by different samples, which is consistent with our observation in Table 4.6, i.e., dynamic learning rate outperforms the static one.

Sparse x and Sparse $g(y)$. Our meta attention modules take $\boldsymbol{x}_{l-1}$ and $\boldsymbol{g}(\boldsymbol{y}_l)$ as inputs, and output attention scores which are used to create sparse $\boldsymbol{g}(\boldsymbol{w}_l)$. However, applying the resulted sparse attention scores on $\boldsymbol{x}_{l-1}$ and $\boldsymbol{g}(\boldsymbol{y}_l)$ can also bring memory and computation benefits, as discussed in

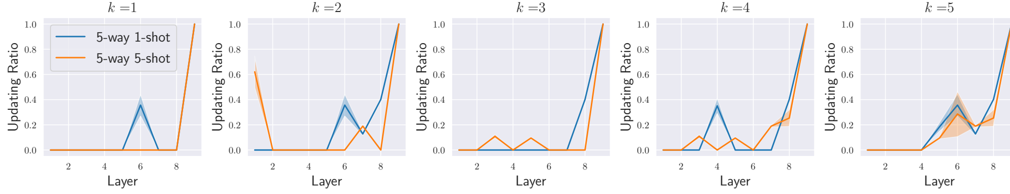

Figure 4.5: Layer-wise updating ratios (mean ± standard deviation) in each updating step. Note that the ratio of updated weights is determined by both static layer-wise inner step sizes $\alpha_{1:L}^{1:K}$ and the dynamic meta attention scores $\gamma_{1:L}$. The layer with an updating ratio of 0 means its $\alpha = 0$.

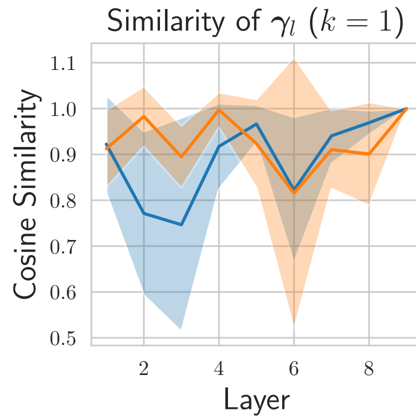

Figure 4.6: Cosine similarity (mean ± standard deviation) of $\gamma_{1:L}$ between random pair of data samples. The results are reported in step 1, because all samples are fed into the same initial model in step 1.

Section 4.4.1. We conduct the ablations when multiplying attention scores γ_l^{fw} and γ_l^{bw} on $\boldsymbol{g}(\boldsymbol{w}_l)$ (also the one used in the main text), or on $\boldsymbol{x}_{l-1}$ and $\boldsymbol{g}(\boldsymbol{y}_l)$ respectively. The results in Table 4.7 show that a channel-wise sparse $\boldsymbol{x}_{l-1}$ hugely degrades the performance, in comparison to only imposing sparsity on $\boldsymbol{g}(\boldsymbol{w}_l)$ while using a dense $\boldsymbol{x}_{l-1}$ in the forward pass. In addition, directly adopting a sparse $\boldsymbol{g}(\boldsymbol{y}_l)$ in backpropagation may even cause non-convergence in few-shot learning. We think this is due to the fact that the error accumulates along the propagation when imposing sparsity on $\boldsymbol{x}_{l-1}$ or $\boldsymbol{g}(\boldsymbol{y}_l)$.

4.6.6 Ablation Studies on Pooling & Normalization Layers

In this section, we test the backbone network with different types of pooling and normalization. Without further notations in the following experiments, we meta-train our "4Conv" backbone on MiniImageNet with full batch sizes, and conduct few-shot learning with gradient accumulation with a batch size of 1, as in Section 4.6.1. Here, we report the results with the original "MAML" method [FAL17] in Table 4.8. Clearly,

$\rho = 0.3$		5-way 1-shot			5-way 5-shot		
fw	bw	Mini	Tiered	CUB	Mini	Tiered	CUB
x	x	47.1%	52.3%	41.8%	62.9%	68.3%	59.3%
$g(w_l)$	x	48.2%	53.6%	41.2%	63.6%	69.0%	59.0%
x_{l-1}	x	37.4%	37.9%	35.4%	47.9%	49.3%	42.5%
x	$g(w_l)$	48.0%	53.0%	42.6%	64.0%	67.8%	59.9%
x	$g(y_l)$	22.8%	21.1%	20.6%	20.7%	21.0%	20.4%

x: no forward/backward (sparse) meta attention, i.e., $\gamma_l^{\text{fw}} = 1$ or $\gamma_l^{\text{bw}} = 1$.

Table 4.7: Ablation results of sparse x_{l-1} and sparse $g(y_l)$.

4Conv		5-way 1-shot		
Pooling	Normalization	Mini	Tiered	CUB
Average-pooling	Batch normalization	25.3%	27.2%	26.1%
Average-pooling	Group normalization	45.8%	50.3%	40.2%
Max-pooling	Batch normalization	27.6%	28.9%	26.5%
Max-pooling	Group normalization	46.2%	51.4%	39.9%

Table 4.8: Comparison between different pooling and normalization layers.

the discrepancy of batch statistics between meta-training phase and few-shot learning phase causes a large accuracy loss in batch normalization layers. Batch normalization works only if few-shot learning uses full batch sizes, i.e., without gradient accumulation, which however does not fit in our memory-constrained scenarios (see Section 4.4.4.1). In addition, max-pooling performs better than average-pooling. We thus use group normalization and max-pooling in our backbone model, see Section 4.6.1.

4.6.7 Ablation Studies on Sample Batch Size

In this section, we show the effects brought from different sample batch sizes. As the setting mentioned in Section 4.6.1, the full batch sizes is adopted in meta-training phase with our p-Meta. During the few-shot learning phase, gradient accumulation is applied to fit different on-device memory constraints. We report the accuracy when adopting different sample batch sizes in gradient accumulation. Although group normalization eliminates the variance of batch statistics, adopting different batch sizes may still result in diverse performance due to the batch-averaged scores in meta attention. The results in Table 4.9 show that different batch sizes yield a similar accuracy level, which indicates that our meta attention module is relatively robust to batch sizes.

	5-way 1-shot			5-way 5-shot		
Batch Size	1	2	5	1	5	25
Mini	48.8%	48.7%	48.3%	65.0%	65.1%	64.7%
Tiered	53.9%	53.6%	54.3%	68.5%	68.9%	68.1%
CUB	42.6%	42.1%	42.4%	60.2%	59.5%	60.6%

Table 4.9: Ablation results of sample batch sizes.

4.7 Summary

In this chapter, we propose a new meta learning method p-Meta for memory-efficient few-shot learning on unseen tasks. p-Meta enables efficient learning on edge devices. On-device learning of a DNN requires both data efficiency and memory efficiency. However, on the one hand, existing low memory training methods fail to learn a DNN given only a few training samples; on the other hand, current few-shot learning methods require a significant amount of dynamic memory. p-Meta addresses these challenges by (*i*) meta-training an initial backbone that can fast adapt to unseen tasks with only a few samples, (*ii*) meta-training a selection mechanism that can identify structurewise adaptation-critical weights to reduce the training memory. The main contributions of DRESS are summarized as follows,

- p-Meta enables data- and memory-efficient DNN (re-)training given new unseen tasks. p-Meta utilizes gradient-based model adaptation, and thus is applicable to various tasks, e.g., classification, regression, and reinforcement learning.

- p-Meta adopts structured partial parameter updates for low-memory training, which is realized by automatically identifying adaptation-critical weights both layer-wise and channel-wise. This hierarchical approach combines static selection of layers and dynamic selection of channels whose weights are critical for few-shot learning on the given new task, and avoids the redundant updating of non-critical weights. This way, the necessary memory consumption required for optimizing adaptation-critical weights decreases. To the best of our knowledge, p-Meta is the first meta learning method designed for on-device few-shot learning.

- Evaluations on few-shot image classification and reinforcement learning show that p-Meta not only improves the accuracy but also reduces the peak dynamic memory by a factor of 2.5 on average over the state-of-the-art few-shot learning methods. p-Meta can also simultaneously reduce the computation by a factor of 1.7 on average.

This chapter studied how to conduct learning on edge devices with limited dynamic memory and limited training data. Note that the methods proposed in the previous chapters solely target the application scenarios on a single edge platform. Edge-server-system is another common scenario of edge intelligence, where multiple resource-constrained edge nodes are remotely connected with a resource-sufficient central server. In the next chapter, we will study how to deploy DNNs on edge-server-system to achieve an efficient inference and an efficient updating.

5

Edge-Server System

In Chapter 2, Chapter 3 and Chapter 4, we studied how to conduct inference, adaptation, and learning on a single edge device, respectively. Edge-server system is another commonly used infrastructure for edge intelligent applications. In edge-server system, several resource-constrained edge devices are connected to a remote server with sufficient resources, and some public information is allowed to be communicated between edge devices and the server. In this chapter, we design a new pipeline to enable efficient inference and efficient updating for edge-server system.

Main Resource Constraints. The main resource constraints on edge-server system comprise two aspects, (*i*) the limited resources on edge devices e.g., from memory, computing power, and energy, as discussed in Chapter 2 and Chapter 3, (*ii*) the limited communication resources e.g., from bandwidth.

Principles. On-device inference is preferred over cloud inference, since it can achieve a fast and stable inference with less energy consumption. Due to a possible lack of relevant training data at the initial deployment, pretrained DNNs may either fail to perform satisfactorily or be significantly improved after the initial deployment. On such an edge-server system, the remote server retrains the DNNs with newly collected data from edge devices or from other sources and sends the updates to the edge device is preferred over on-device re-training (or federated learning), because of the limited memory and computing power on edge devices. To reduce the communication cost for sending the updated models, we propose a deep partial updating paradigm, where the server only selects and sends a small subset of critical weights that have a large contribution to the loss reduction during the retraining.

The contents of this chapter are established mainly based on the paper "Deep Partial Updating: towards Communication Efficient Updating

for On-Device Inference" that is submitted to European Conference on Computer Vision (ECCV), 2022.

5.1 Introduction

Compared to cloud inference, on-device inference is subject to severe limitations in terms of storage, energy, computing power and communication. On the other hand, it has many advantages, e.g., it enables fast and stable inference even with low communication bandwidth or interrupted communication, and can save energy by avoiding the transfer of data to the cloud, which often costs significant amounts of energy than sensing and computation [PW19, Guo18, LCI+19]. To deploy deep neural networks (DNNs) on resource-constrained edge devices, extensive research has been done to compress a well pretrained model via pruning [HMD16, FC19, RFC20] and quantization [CBD15, RORF16]. During on-device inference, compression may achieve a good balance between model performance and resource demand.

However, due to a possible lack of relevant training data at the time of initial deployment or due to an unknown sensing environment, pretrained DNN models may either fail to perform satisfactorily or be significantly improved after the initial deployment. In other words, retraining the models by using newly collected data (from *edge devices* or *other sources*) is typically required to achieve the desired performance during the lifetime of devices. More samples can better reveal the data distribution, even if new samples follow the same distribution. Thus, the model trained with more data yields a better generalization.

Due to the resource-constrained nature of edge devices in terms of memory and computing power, on-device re-training (or federated learning) is typically restricted to tiny batch size, small inference (sub-)networks or limited optimization steps, all resulting in a performance degradation. Instead, retraining often occurs on a remote server with sufficient resources. One possible strategy to allow for a continuous improvement of the model performance on edge devices is a two-stage iterative process: (*i*) at each round, edge devices collect new data samples and send them to the server, and (*ii*) the server retrains the network using all collected data, and then sends the updates to each edge device [BS06]. The first stage may even not be necessary if new training data is collected in other ways and made directly available to the server.

Example Scenarios. Example application scenarios of relevance include vision robotic sensing in an unknown environment (e.g., Mars) [MQC+17], local translators of low-resource languages on mobile phones [BSK+19, WKMR20], and sensor networks mounted in alpine areas [MFCP+19], automatic wildlife monitoring [SWP+18]. We detail two specific scenarios. *Hazard alarming on mountains:* Researchers in [MFCP+19] mounted

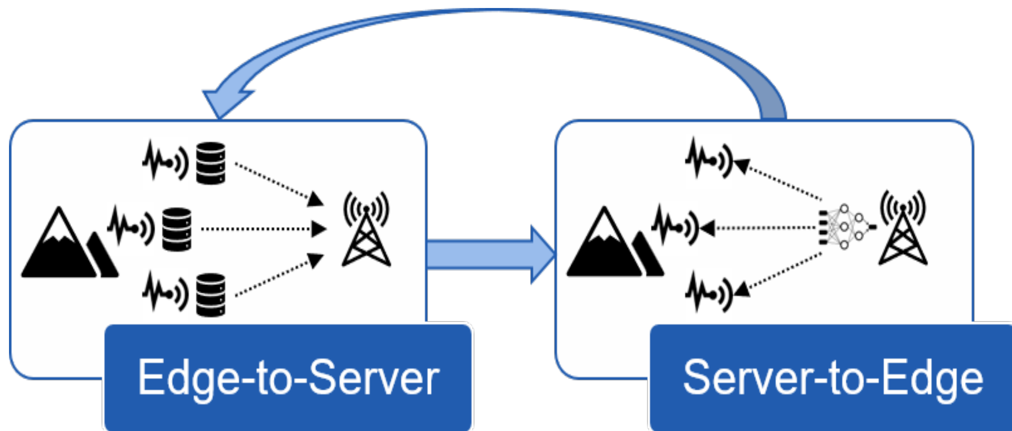

Figure 5.1: The iterative process for updating the deployed inference model on edge devices via a wireless communication. Edge-to-server communication: edge devices collect new data samples and send them to the server. Server-to-edge communication: the server retrains the model and then sends the updates to each edge device. The edge-to-server communication may not be necessary if new training data is collected from other sources and made directly available to the server.

tens of sensors nodes at different scarps in high alpine areas with cameras, geophones and high-precision GPS. The purpose is to achieve fast, stable, and energy-efficient hazard monitoring for early warning to protect people and infrastructure. To this end, a deep neural model is deployed on each node to on-device detect rockfalls and debris flows. The nodes regularly collect and send data to the server for labeling and retraining, and the server sends the updated model back through a low-power wireless network. Retraining during deployment is essential for a highly reliable hazard warning. *Endangered species monitoring:* To detect endangered species, researchers often deploy some audio or image sensor nodes in virgin rainforests [SWP+18]. Edge nodes are supposed to classify the potential signal from endangered species and send these relevant data to the server. Due to the limited prior information from environments and species, retraining the initially classifier with received data or data from other sources (e.g., other areas) is necessary.

Challenges. An essential challenge herein is that the transmissions in the server-to-edge stage are highly constrained by the limited communication resource (e.g., bandwidth, energy) in comparison to the edge-to-server stage, if necessary at all. Typically, state-of-the-art DNN models often require tens or even hundreds of mega-Bytes (MB) to store parameters, whereas a single batch of data samples (a number of samples that lead to reasonable updates in batch training) needs a relatively smaller amount of data. For example, for CIFAR10 dataset [KNH09], the weights of a popular VGGNet require 56.09MB storage, while one batch of 128 samples

only uses around 0.40MB [SZ15, RORF16]. As an alternative approach, the server sends a full update of the inference network once or rarely. But in this case, every node will suffer from a low performance until such an update occurs. Besides, edge devices could decide on and send only critical samples by using active learning schemes [AZK+20]. The server may also receive data from other sources, e.g., through data augmentation based on the data collected in previous rounds or new data collection campaigns. These considerations indicate that the updated weights that are sent to edge devices by the server become a major bottleneck.

Facing the above challenges, we ask the following question: *Is it possible to update only a small subset of weights while reaching a similar performance as updating all weights?* Doing such a *partial updating* can significantly reduce the server-to-edge communication overhead. Furthermore, fewer parameter updates also lead to less memory access on edge devices, which in turn results in smaller energy consumption than full updating [Hor14].

Why Partial Updating Works. Since the network deployed on edge devices is trained with the data collected beforehand, some learned knowledge can be reused. In other words, we only need to distinguish and update the weights which are critical to the newly collected data.

How to Select Weights. Our key concept for partial updating is based on the hypothesis, that *a weight shall be updated only if it has a large contribution to the loss reduction* during the retraining given newly collected data samples. Specially, we define a binary mask $\boldsymbol{m}$ to describe which weights are subject to update, i.e., $m_i = 1$ implies updating this weight and $m_i = 0$ implies fixing the weight to its initial value (also reusing the weight). For any $\boldsymbol{m}$, we establish the analytical upper bound on the difference between the loss value under partial updating and that under full updating. We determine an optimized mask $\boldsymbol{m}$ by combining two different view points: (*i*) measuring each weight's "global contribution" to the upper bound through computing the Euclidean distance, and (*ii*) measuring each weight's "local contribution" to the upper bound using gradient-related information. The weights to be updated according to $\boldsymbol{m}$ will be further sparsely fine-tuned while the remaining weights are rewound to their initial values.

5.2 Related Work

5.2.1 Partial Updating

Although partial updating has been adopted in some prior works, it is conducted in a fairly coarse-grained manner, e.g., layer-wise or neuron-wise, and targets at completely different objectives. Especially, under continual learning settings, [YYLH18, JACM20] propose to freeze all

weights related to the neurons which are more critical in performing prior tasks than new ones, to preserve existing knowledge. Under adversarial attack settings, [SS15] updates the weights in the first several layers only, which yield a dominating impact on the extracted features, for better attack efficacy. Under architecture generalization settings, [CNS20] studies the generalization performance through the resulting loss degradation when rewinding the weights of each individual layer to their initial values. Under meta learning settings, [RRBV20, SLQ+21] reuse learned representations by only updating a subset of layers for efficiently learning new tasks. Unfortunately, such techniques do not focus on reducing the number of updated weights, and thus cannot be applied in our problem setting.

5.2.2 Federated Learning

Communication-efficient federated learning (distributed training) [LHM+18, KMA+19, LSW+20] studies how to compress multiple gradients calculated on different sets of non-*i.i.d.* local data, such that the aggregation of these (compressed) gradients could result in a similar convergence performance as centralized training on all data. Such compressed updates are fundamentally different from our setting, where (*i*) updates are not transmitted in each optimization step; (*ii*) training data are incrementally collected; (*iii*) centralized training is conducted. Our typical scenarios focus on the outdoor, wilderness, or extreme areas, which generally do not involve data privacy issues, since these collected data are not personal data. In comparison to federated learning (local training), our proposed paradigm has the following advantages: (*i*) we do not conduct resource-intensive gradient calculation (backward propagation) on edge devices; (*ii*) the collected data do not need to be continuously accumulated and stored on memory-constrained edge nodes; (*iii*) we also avoid the difficult but necessary labeling process on each edge node in supervised learning tasks; (*iv*) when few events occur on some nodes in some rounds, the centralized training may avoid some degraded updates in local training, e.g., batch normalization.

5.2.3 Compression

The communication cost could also be reduced through some compression techniques, e.g., quantizing/encoding the updated weights and the transmission signal. But note that these techniques are orthogonal to our approach and could be applied in addition. Following the compression pipeline in [HMD16], the resulted sparse updating from our methods could be further quantized and Huffman-encoded.

5.2.4 Unstructured Pruning

Deep partial updating is inspired by recent unstructured pruning methods, e.g., [HMD16, FC19, RFC20, EGM+21, PIVA21]. Traditional pruning methods aim at reducing the number of operations and storage consumption by setting some weights to zero. Sending a pruned network with only non-zero's weights may also reduce the communication cost, but to a much lesser extent as shown in the experimental results, see Section 5.5.6. Since our objective namely reducing the server-to-edge communication cost when updating the deployed networks is fundamentally different from pruning, we can leverage some learned knowledge by retaining weights (partial updating) instead of zero-outing weights (pruning).

5.2.5 Domain Adaptation

Domain adaptation targets reducing domain shift to transfer knowledge into new learning tasks [ZQD+19]. This chapter mainly considers the scenario where the inference task is not explicitly changed along the rounds, i.e., the overall data distribution maintains the same along the data collection rounds. Thus, selecting critical weights (features) by measuring their impact on domain distribution discrepancy is invalid herein. Applying deep partial updating on streaming tasks where the data distribution varies along the rounds would be also worth studying, and we leave it for future works.

5.3 Notations and Settings

In this section, we define the notations used throughout this chapter, and provide a formalized problem setting, i.e., deep partial updating. We consider a set of remote edge devices that implement on-device inference. They are connected to a host server that is able to perform network training and retraining. We consider the necessary amount of information that needs to be communicated to each edge device to update its inference network.

Assume there are in total R rounds of network updates. The network deployed in the r-th round is represented with its weight vector $\boldsymbol{w}^r$. The training data used to update the network for the r-th round is represented as $\mathcal{D}^r = \delta\mathcal{D}^r \cup \mathcal{D}^{r-1}$. Also, newly collected data samples $\delta\mathcal{D}^r$ are made available to the server in round $r-1$.

To reduce the amount of information that needs to be sent to edge devices, only partial weights of $\boldsymbol{w}^{r-1}$ shall be updated when determining $\boldsymbol{w}^r$. The overall optimization problem for weight-wise partial updating in round $r-1$ is thus,

$$\min_{\delta \boldsymbol{w}^r} \quad \ell\left(\boldsymbol{w}^{r-1} + \delta\boldsymbol{w}^r; \mathcal{D}^r\right) \tag{5.1}$$

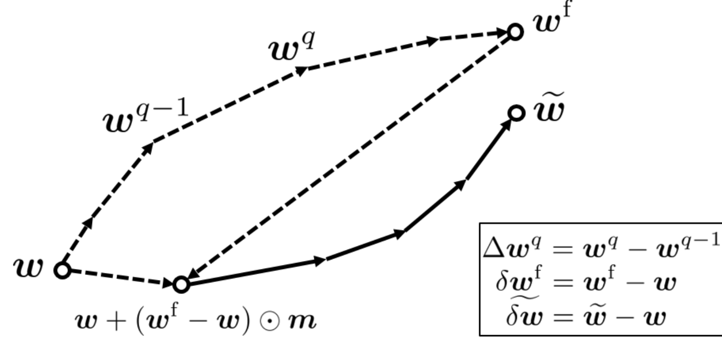

Figure 5.2: The figure depicts the overall approach that consists of two steps. The first step is depicted with dotted arrows and starts from the deployed network w. In Q optimization steps, all weights are trained to the optimum w^f. Based on the collected information, a binary mask m is determined that characterizes the set of weights that are rewound to the ones of w. Therefore, the second step (solid arrows) starts from $w + \delta w^f \odot m$. According to the mask, this initial solution is sparsely fine-tuned to the final weights $\widetilde{w}$, i.e., $\widetilde{\delta w}$ has only non-zero values where the mask value is 1

$$\text{s.t.} \quad ||\delta w^r||_0 \leq k \cdot I \tag{5.2}$$

where ℓ denotes the loss function, $||.||_0$ denotes the L0-norm, k denotes the updating ratio that is determined by the communication constraints in practical scenarios, and δw^r denotes the increment of w^{r-1}. Note that both w^{r-1} and δw^r are drawn from $\mathbb{R}^I$, where I is the total number of weights.

In this case, only a fraction of $k \cdot I$ weights and the corresponding index information need to be communicated to each edge device for updating the network in round r, namely the partial updates δw^r. It is worth noting that the index information is relatively small in size compared to the partially updated weights (see Section 5.5.0). On each edge device, the weight vector is updated as $w^r = w^{r-1} + \delta w^r$. To simplify the notation, we will only consider a single update, i.e., from weight vector w (corresponding to w^{r-1}) to weight vector $\widetilde{w}$ (corresponding to w^r) with $\widetilde{w} = w + \widetilde{\delta w}$.

5.4 Deep Partial Updating

We develop a two-step approach for resolving the partial updating optimization problem in Eq. (5.1)-Eq. (5.2). The final experimental implementation in Section 5.5 contains some minor adaptations that do not change the main principles as explained next. The overall approach is depicted in Figure 5.2.

- The First Step: **Full Updating and Rewinding.** The first step not only determines the subset of weights that are allowed to change their values, but also computes the initial values for the second step. In particular, we first optimize the loss function Eq. (5.1) by updating all weights from the initialization $\boldsymbol{w}$ with a standard optimizer, e.g., SGD or its variants. We thus obtain the minimized loss $\ell\left(\boldsymbol{w}^{\mathrm{f}}\right)$ with $\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}}$, where the superscript f denotes "full updating". To consider the constraint of Eq. (5.2), the information gathered during this optimization is used to determine the subset of weights that will be changed, also that are communicated to the edge devices. In the explanation of the method in Section 5.4.1, we use the mask $\boldsymbol{m}$ with $\boldsymbol{m} \in \{0,1\}^I$ to describe which weights are subject to change and which ones are not. The weights with $m_i = 1$ are trainable, whereas the weights with $m_i = 0$ will be rewound from the values in $\boldsymbol{w}^{\mathrm{f}}$ to their initial values in $\boldsymbol{w}$, i.e., unchanged. Obviously, we find $\|\boldsymbol{m}\|_0 = \sum_i m_i = k \cdot I$.
- The Second Step: **Sparse Fine-Tuning.** In the second step we start a sparse fine-tuning from a network with $k \cdot I$ weights from the optimized network $\boldsymbol{w}^{\mathrm{f}}$ and $(1-k) \cdot I$ weights from the previous, still deployed network $\boldsymbol{w}$. In other words, the initial weights for the second step are $\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}$, where $\odot$ denotes an element-wise multiplication. To determine the final solution $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta\boldsymbol{w}}$, we conduct a sparse fine-tuning (still with a standard optimizer), i.e., we keep all weights with $m_i = 0$ constant during the optimization. Therefore, $\widetilde{\delta\boldsymbol{w}}$ is zero wherever $m_i = 0$, and only weights where $m_i = 1$ are updated.

5.4.1 Metrics for Rewinding

We will now describe a new metric that allows us to determine the weights that should be kept constant, i.e., those whose masks satisfy $m_i = 0$. Like most learning methods, we focus on minimizing a loss function. The two-step approach relies on the following assumption: the better the loss $\ell(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m})$ of the initial solution for the second step, the better the final performance. Therefore, the first step in the method should select a mask $\boldsymbol{m}$ such that the loss difference $\ell(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}) - \ell(\boldsymbol{w}^{\mathrm{f}})$ is as small as possible.

To determine an optimized mask $\boldsymbol{m}$, we propose to upper-bound the above loss difference in two view points, and measure each weight's contribution to the bounds. The "global contribution" uses information contained in the difference $\delta\boldsymbol{w}^{\mathrm{f}}$ between the initial weights $\boldsymbol{w}$ and the optimized weights $\boldsymbol{w}^{\mathrm{f}}$ by full updating, namely the norm of incremental weights. The "local contribution" takes into account the gradient-based information that is gathered during the optimization in the first step,

i.e., in the path from $\boldsymbol{w}$ to $\boldsymbol{w}^{\mathrm{f}}$. Both contributions will be combined to determine an optimized mask $\boldsymbol{m}$.

The two view points are based on the concept of smooth differentiable functions, see for example [Nes98]. A function $f(x)$ with $f : \mathbb{R}^d \to \mathbb{R}$ is called L-smooth if it has a Lipschitz continuous gradient $g(x)$: $\|g(x) - g(y)\|_2 \leq L\|x - y\|_2$ for all x, y. Note that Lipschitz continuity of a gradient is essential to ensuring convergence of many gradient-based algorithms. Under such a condition, one can derive the following bounds, see also [Nes98]:

$$|f(y) - f(x) - g(x)^{\mathrm{T}} \cdot (y - x)| \leq L/2 \cdot \|y - x\|_2^2 \quad \forall x, y \tag{5.3}$$

This basic relation is used to justify the global and the local contributions, i.e., the rewinding metrics.

Global Contribution. Following some state-of-the-art pruning methods, one would argue that a large absolute value in $\delta\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^{\mathrm{f}} - \boldsymbol{w}$ indicates that this weight has moved far from its initial value in $\boldsymbol{w}$, and thus should not be rewound. This motivates us to adopt the widely used unstructured magnitude pruning to solve the problem of determining an optimized mask $\boldsymbol{m}$. Magnitude pruning prunes the weights with the lowest magnitudes, which is the current best-performed pruning method aiming at the trade-off between the model accuracy and the number of zero's weights [RFC20].

Using $a - b \leq |a - b|$, Eq. (5.3) can be reformulated as $f(y) - f(x) - g(x)^T(y - x) \leq |f(y) - f(x) - g(x)^T(y - x)| \leq L/2 \cdot \|y - x\|_2^2$. Thus, we can bound the relevant difference in the loss $\ell(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}) - \ell(\boldsymbol{w}^{\mathrm{f}}) \geq 0$ as

$$\ell(\boldsymbol{w}+\delta\boldsymbol{w}^{\mathrm{f}}\odot\boldsymbol{m})-\ell(\boldsymbol{w}^{\mathrm{f}}) \leq \boldsymbol{g}(\boldsymbol{w}^{\mathrm{f}})^{\mathrm{T}}\cdot\Big(\delta\boldsymbol{w}^{\mathrm{f}} \odot (\boldsymbol{m} - \mathbf{1})\Big)+L/2\cdot\|\delta\boldsymbol{w}^{\mathrm{f}}\odot(\boldsymbol{m}-\mathbf{1})\|_2^2 \tag{5.4}$$

where $\boldsymbol{g}(\boldsymbol{w}^{\mathrm{f}})$ denotes the gradient of the loss function at $\boldsymbol{w}^{\mathrm{f}}$, and $\mathbf{1}$ is a vector whose elements are all 1. As the loss is optimized at $\boldsymbol{w}^{\mathrm{f}}$, i.e., $\boldsymbol{g}(\boldsymbol{w}^{\mathrm{f}}) \approx \mathbf{0}$, we can assume that the gradient term is much smaller than the norm of the weight differences in Eq. (5.4). Therefore, we obtain approximately

$$\ell(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}) - \ell(\boldsymbol{w}^{\mathrm{f}}) \lesssim L/2 \cdot \|\delta\boldsymbol{w}^{\mathrm{f}} \odot (\mathbf{1} - \boldsymbol{m})\|_2^2 \tag{5.5}$$

The right hand side is clearly minimized if $m_i = 1$ for the largest absolute values of $\delta\boldsymbol{w}^{\mathrm{f}}$. As $\mathbf{1}^{\mathrm{T}}\cdot\Big(\boldsymbol{c}^{\mathrm{global}} \odot (\mathbf{1} - \boldsymbol{m})\Big) = \|\delta\boldsymbol{w}^{\mathrm{f}}\odot(\mathbf{1}-\boldsymbol{m})\|_2^2$, this information is captured in the contribution vector

$$\boldsymbol{c}^{\mathrm{global}} = \delta\boldsymbol{w}^{\mathrm{f}} \odot \delta\boldsymbol{w}^{\mathrm{f}} \tag{5.6}$$

The $k \cdot I$ weights with the largest values in $\boldsymbol{c}^{\mathrm{global}}$ are assigned to mask values 1 and are further fine-tuned in the second step, whereas all others are rewound to their initial values in $\boldsymbol{w}$. Algorithm 5.1 shows this first approach.

Algorithm 5.1: Global Contribution Partial Updating (Prune Incremental Weights)

Input: Weights $\boldsymbol{w}$, updating ratio k, learning rate $\{\alpha^q\}_{q=1}^{Q}$
Output: Weights $\widetilde{\boldsymbol{w}}$

```
/* The first step:  full updating and rewinding      */
```

1 Initiate $\boldsymbol{w}^0 = \boldsymbol{w}$;
2 **for** $q \leftarrow 1$ **to** Q **do**
3 Compute the loss gradient $\boldsymbol{g}(\boldsymbol{w}^{q-1}) = \partial\ell(\boldsymbol{w}^{q-1})/\partial\boldsymbol{w}^{q-1}$;
4 Compute the optimization step with learning rate α^q as $\Delta\boldsymbol{w}^q$;
5 Update $\boldsymbol{w}^q = \boldsymbol{w}^{q-1} + \Delta\boldsymbol{w}^q$;
6 Set $\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^Q$ and get $\delta\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^{\mathrm{f}} - \boldsymbol{w}$;
7 Compute $\boldsymbol{c}^{\mathrm{global}} = \delta\boldsymbol{w}^{\mathrm{f}} \odot \delta\boldsymbol{w}^{\mathrm{f}}$ and sort in descending order;
8 Create binary masks $\boldsymbol{m}$ with 1 for Top-$(k \cdot I)$ indices, 0 for others;

```
/* The second step:  sparse fine-tuning              */
```

9 Initiate $\widetilde{\delta\boldsymbol{w}} = \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta\boldsymbol{w}}$;
10 **for** $q \leftarrow 1$ **to** Q **do**
11 Compute the optimization step with learning rate α^q as $\Delta\widetilde{\boldsymbol{w}}^q$;
12 Update $\widetilde{\delta\boldsymbol{w}} = \widetilde{\delta\boldsymbol{w}} + \Delta\widetilde{\boldsymbol{w}}^q \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta\boldsymbol{w}}$;

Local Contribution. As experiments show, one can do better when leveraging in addition some gradient-based information gathered during the first step, i.e., optimizing the initial weights $\boldsymbol{w}$ in Q traditional optimization steps, $\boldsymbol{w} = \boldsymbol{w}^0 \to \cdots \to \boldsymbol{w}^{q-1} \to \boldsymbol{w}^q \to \cdots \to \boldsymbol{w}^Q = \boldsymbol{w}^{\mathrm{f}}$.

Using $-a + b \leq |a - b|$, Eq. (5.3) can be reformulated as $f(x) - f(y) + g(x)^T(y - x) \leq |f(y) - f(x) - g(x)^T(y - x)| \leq L/2 \cdot ||y - x||_2^2$. This leads us to bound each optimization step as

$$\ell(\boldsymbol{w}^{q-1}) - \ell(\boldsymbol{w}^q) \leq -\boldsymbol{g}(\boldsymbol{w}^{q-1})^{\mathrm{T}} \cdot \Delta\boldsymbol{w}^q + L/2 \cdot ||\Delta\boldsymbol{w}^q||_2^2 \tag{5.7}$$

where $\Delta\boldsymbol{w}^q = \boldsymbol{w}^q - \boldsymbol{w}^{q-1}$. For a conventional gradient descent optimizer with a small learning rate we can use the approximation $|\boldsymbol{g}(\boldsymbol{w}^{q-1})^{\mathrm{T}} \cdot \Delta\boldsymbol{w}^q| \gg ||\Delta\boldsymbol{w}^q||_2^2$ and obtain $\ell(\boldsymbol{w}^{q-1}) - \ell(\boldsymbol{w}^q) \lesssim -\boldsymbol{g}(\boldsymbol{w}^{q-1})^{\mathrm{T}} \cdot \Delta\boldsymbol{w}^q$. Summing up over all optimization iterations yields approximately

$$\ell(\boldsymbol{w}^{\mathrm{f}} - \delta\boldsymbol{w}^{\mathrm{f}}) - \ell(\boldsymbol{w}^{\mathrm{f}}) \lesssim -\sum_{q=1}^{Q} \boldsymbol{g}(\boldsymbol{w}^{q-1})^{\mathrm{T}} \cdot \Delta\boldsymbol{w}^q \tag{5.8}$$

Note that we have $\boldsymbol{w} = \boldsymbol{w}^{\mathrm{f}} - \delta\boldsymbol{w}^{\mathrm{f}}$ and $\delta\boldsymbol{w}^{\mathrm{f}} = \sum_{q=1}^{Q} \Delta\boldsymbol{w}^q$. Therefore, with $\boldsymbol{m} \sim \boldsymbol{0}$ we can reformulate Eq. (5.8) as $\ell\left(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}\right) - \ell(\boldsymbol{w}^{\mathrm{f}}) \lesssim \mathrm{U}(\boldsymbol{m})$ with the upper bound $\mathrm{U}(\boldsymbol{m}) = -\sum_{q=1}^{Q} \boldsymbol{g}(\boldsymbol{w}^{q-1})^{\mathrm{T}} \cdot (\Delta\boldsymbol{w}^q \odot (\boldsymbol{1} - \boldsymbol{m}))$ where we suppose that the gradients are approximately constant for small $\boldsymbol{m}$. Therefore, an approximate incremental contribution of each weight dimension to

the upper bound on the loss difference $\ell\left(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}\right) - \ell(\boldsymbol{w}^{\mathrm{f}})$ can be determined by the negative gradient vector at $\boldsymbol{m} = \boldsymbol{0}$, denoted as

$$\boldsymbol{c}^{\mathrm{local}} = -\frac{\partial \mathrm{U}(\boldsymbol{m})}{\partial \boldsymbol{m}} = -\sum_{q=1}^{Q} \boldsymbol{g}(\boldsymbol{w}^{q-1}) \odot \Delta \boldsymbol{w}^{q} \tag{5.9}$$

which models the accumulated contribution to the overall loss reduction.

Combining Global and Local Contribution. So far, we independently calculate the global and local contributions c^{global} and c^{local}, respectively. To avoid the impact due to the scale, we first normalize each contribution by its significance in its own set (either global contribution set or local contribution set). We investigate the impacts and the different combination of both normalized contributions, see results in Section 5.5.4. Interestingly, the most straightforward combination (i.e., the sum of both normalized metrics) yields a better and more stable performance. Intuitively, local contribution can better identify critical weights w.r.t. the loss during training, while global contribution may be more robust for a highly non-convex loss landscape. Both metrics may be necessary when selecting weights to rewind. Therefore, the combined contribution is computed as

$$\boldsymbol{c} = \frac{1}{\boldsymbol{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{global}}} \boldsymbol{c}^{\mathrm{global}} + \frac{1}{\boldsymbol{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{local}}} \boldsymbol{c}^{\mathrm{local}} \tag{5.10}$$

and $m_i = 1$ for the $k \cdot I$ largest values of $\boldsymbol{c}$ and $m_i = 0$ otherwise. The pseudocode of Deep Partial Updating (DPU), i.e., rewinding according to the combined contribution to the loss reduction, is shown in Algorithm 5.2.

We further analyze the complexity of Algorithm 5.2. Recall that the dimensionality of the weights vector is denoted as I. In Q optimization iterations during the first step, Algorithm 5.2 introduces an extra time complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. The rest of the first step takes a time complexity of $O(I \cdot \log(I))$ and a space complexity of $O(I)$, (e.g., using heap sort or quick sort). In Q optimization iterations during the second step, Algorithm 5.2 introduces an extra time complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. Thus, a total extra time complexity is $O(2QI+I\cdot\log(I))$ and a total extra space complexity is $O(I)$.

5.4.2 (Re-)Initialization of Weights

In this section, we discuss the initialization of our method. $\mathcal{D}^1$ denotes the initial dataset used to train the network $\boldsymbol{w}^1$ from a randomly initialized network $\boldsymbol{w}^0$. $\mathcal{D}^1$ corresponds to the available dataset before deployment, or collected in the 0-th round if there are no data available

Algorithm 5.2: Deep Partial Updating

Input: Weights $\boldsymbol{w}$, updating ratio k, learning rate $\{\alpha^q\}_{q=1}^{Q}$
Output: Weights $\widetilde{\boldsymbol{w}}$

```
/* The first step:  full updating and rewinding      */
```

1 Initiate $\boldsymbol{w}^0 = \boldsymbol{w}$ and $\boldsymbol{c}^{\text{local}} = \boldsymbol{0}$;
2 **for** $q \leftarrow 1$ **to** Q **do**
3   Compute the loss gradient $\boldsymbol{g}(\boldsymbol{w}^{q-1}) = \partial\ell(\boldsymbol{w}^{q-1})/\partial\boldsymbol{w}^{q-1}$;
4   Compute the optimization step with learning rate α^q as $\Delta\boldsymbol{w}^q$;
5   Update $\boldsymbol{w}^q = \boldsymbol{w}^{q-1} + \Delta\boldsymbol{w}^q$;
6   Update $\boldsymbol{c}^{\text{local}} = \boldsymbol{c}^{\text{local}} - \boldsymbol{g}(\boldsymbol{w}^{q-1}) \odot \Delta\boldsymbol{w}^q$;
7 Set $\boldsymbol{w}^{\text{f}} = \boldsymbol{w}^Q$ and get $\delta\boldsymbol{w}^{\text{f}} = \boldsymbol{w}^{\text{f}} - \boldsymbol{w}$;
8 Compute $\boldsymbol{c}^{\text{global}} = \delta\boldsymbol{w}^{\text{f}} \odot \delta\boldsymbol{w}^{\text{f}}$;
9 Compute $\boldsymbol{c}$ as Eq. (5.10) and sort in descending order;
10 Create binary masks $\boldsymbol{m}$ with 1 for Top-$(k \cdot I)$ indices, 0 for others;

```
/* The second step:  sparse fine-tuning              */
```

11 Initiate $\widetilde{\delta\boldsymbol{w}} = \delta\boldsymbol{w}^{\text{f}} \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta\boldsymbol{w}}$;
12 **for** $q \leftarrow 1$ **to** Q **do**
13   Compute the optimization step with learning rate α^q as $\Delta\widetilde{\boldsymbol{w}}^q$;
14   Update $\widetilde{\delta\boldsymbol{w}} = \widetilde{\delta\boldsymbol{w}} + \Delta\widetilde{\boldsymbol{w}}^q \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta\boldsymbol{w}}$;

before deployment. $\{\delta\mathcal{D}^r\}_{r=2}^{R}$ denotes newly collected samples in each subsequent round.

Experimental results show (see Section 5.5.2) that training from a randomly initialized network can yield a higher accuracy *after a large number of rounds*, compared to always training from the last round with weights $\boldsymbol{w}^{r-1}$. As a possible explanation, the optimizer could end in a hard to escape region of the search space if always trained from the last round for a long sequence of rounds. Thus, we propose to re-initialize the weights after a certain number of rounds. In such a case, Algorithm 5.2 does not start from the previous weights $\boldsymbol{w}^{r-1}$ but from randomly initialized weights. The randomly re-initialized network can be efficiently sent to the edge devices via a single random seed. The device can determine the weights by means of a random generator. This process realizes a random shift in the search space, which is a communication-efficient way in comparison to other alternatives, such as learning to increase the loss or using the (averaged) weights in the previous rounds, as these fully changed weights still need to be sent to each node. Each time the network is randomly re-initialized, the new partially updated network might suffer from an accuracy drop in a few rounds. However, we can simply avoid such an accuracy drop by not updating the network if the validation accuracy does not increase compared to the last round, see in Section 5.5.3. Note that the learned

knowledge thrown away by re-initialization can be re-learned afterwards, since all collected samples are continuously stored and accumulated in the server. This also makes our setting different from continual learning, that aims at avoiding catastrophic forgetting without accessing old data.

To determine after how many rounds the network should be re-initialized, we conduct extensive experiments on different partial updating settings, see more discussions and results in Section 5.5.3. In conclusion, the network is randomly re-initialized as long as the number of total newly collected data samples exceeds the number of samples when the network was re-initialized last time. For example, assume that at round r the model is randomly (re-)initialized and partially updated from this random network on dataset $\mathcal{D}^r$. Then, the model will be re-initialized again at round $r+n$, if $|\mathcal{D}^{r+n}| > 2 \cdot |\mathcal{D}^r|$, where $|.|$ denotes the number of samples in the dataset.

5.5 Evaluation

In this section, we experimentally show that through updating a small subset of weights, DPU can reach a similar accuracy as full updating while requiring a significantly lower communication cost. We implement DPU with Pytorch [PGC+17], and evaluate on public vision datasets, including MNIST [LC10], CIFAR10 [KNH09], CIFAR100 [KNH09], ImageNet [RDS+15], using multilayer perceptron (MLP), VGGNet [CBD15, RORF16], ResNet56 [HZRS16], MobileNetV1 [HZC+17], respectively. Particularly, we partition the experiments into multi-round updating and single-round updating.

Multi-Round Updating. We consider there are limited (or even zero) samples before the initial deployment, and data samples are continuously collected and sent from edge devices over a long period (the event rate is often low in real cases [MFCP+19]). The server retrains the model and sends the updates to each device in multiple rounds. Regarding the highly-constrained communication resources, we choose low resolution image datasets (MNIST [LC10] and CIFAR10/100 [KNH09]) to evaluate multi-round updating. We conduct one-shot rewinding in multi-round DPU, i.e., rewinding is executed only once to achieve the desired updating ratio at each round (as in Algorithm 5.2), which avoids hand-tuning hyperparameters (e.g., updating ratio schedule) frequently over a large number of rounds.

Single-Round Updating. The deployed model is updated once via server-to-edge communication when new data from other sources become available on the server after some time, e.g., releasing a new version of mobile applications based on newly retrieved internet data. Although DPU is elaborated under multi-round updating settings, it can be applied

directly during single-round updating. Since transmission from edge devices may be even not necessary, we evaluate single-round DPU on the large scale dataset ImageNet. Iterative rewinding is adopted here due to its better performance. Particularly, we alternatively perform rewinding 20% of the remaining trainable weights according to Eq. (5.10) and sparse fine-tuning until reaching the desired updating ratio.

General Settings. Let $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ represent the available data samples along rounds, where $|\delta\mathcal{D}^r|$ is supposed to be constant along rounds. Both $\mathcal{D}^1$ and $\delta\mathcal{D}^r$ are randomly drawn from the original training dataset to simulate the data collection. In each round, the test accuracy is reported, when the validation dataset achieves the highest Top-1 accuracy during retraining. When the validation accuracy does not increase compared to the previous round, the models are not updated to reduce communication overhead. This strategy is also applied to other baselines to enable a fair comparison. We use the average cross-entropy as the loss function, and use Adam variant of SGD for MLP and VGGNet, Nesterov SGD for ResNet56 and MobileNetV1. More implementation details are provided in Section 5.5.1.

Indexing. DPU generates a sparse tensor. In addition to the updated weights, the indices of these weights also need to be sent to each edge device. A simple implementation is to send the mask $\boldsymbol{m}$, i.e., a binary vector of I elements. Let S_w denote the bitwidth of each single weight, and S_x denote the bitwidth of each index. Directly sending $\boldsymbol{m}$ yields an overall communication cost of $I \cdot k \cdot S_w + I \cdot S_x$ with $S_x = 1$. To save the communication cost on indexing, we further encode $\boldsymbol{m}$. Suppose that $\boldsymbol{m}$ is a random binary vector with a probability of k to contain 1. The optimal encoding scheme according to Shannon yields $S_x(k) = k \cdot \log(1/k) + (1-k) \cdot \log(1/(1-k))$. Coding schemes such as Huffman block coding can come close to this bound. Partial updating results in a smaller communication data size than full updating, if $S_w \cdot I > S_w \cdot k \cdot I + S_x(k) \cdot I$. Under the worst case for indexing cost, i.e., $S_x(k = 0.5) = 1$, as long as $k < (32-1)/32 = 0.97$, partial updating can yield a smaller communication data size with $S_w = 32$-bit weights. We use $S_w \cdot k \cdot I + S_x(k) \cdot I$ to report the size of data transmitted from server to each node at each round, contributed by the partially updated weights plus the encoded indices of these weights.

5.5.1 Benchmarking Details

5.5.1.1 MLP on MNIST

The MNIST dataset [LC10] consists of 28×28 gray scale images in 10 digit classes. It contains a training dataset with 60000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with

size of 128 training on 1 GeForce RTX 3090 GPU. The number of training epochs is chosen as 60 at each round. We use Adam variant of SGD as the optimizer, and use all default parameters provided by Pytorch. The initial learning rate is 0.05, and it decays with a factor of 0.1 every 20 epochs. For fair comparison, we adopt the same learning rate for other baseline methods. The used MLP contains two hidden layers, and each hidden layer contains 512 hidden units. The input is a 784-dim tensor of all pixel values for each image. All weights in MLP need around 2.67MB. Each data sample needs 0.784KB. The size of MLP equals around 3400 data samples. The used MLP architecture is presented as,
2×512FC - 10SVM.

5.5.1.2 VGGNet on CIFAR10

The CIFAR10 dataset [KNH09] consists of 32×32 color images in 10 object classes. It contains a training dataset with 50000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with size of 128 training on 1 GeForce RTX 3090 GPU. The number of training epochs is chosen as 60 at each round. We use Adam variant of SGD as the optimizer, and use all default parameters provided by Pytorch. The initial learning rate is 0.05, and it decays with a factor of 0.2 every 20 epochs. The used VGGNet is widely adopted in many previous compression works [CBD15, RORF16], which is a modified version of the original VGG [SZ15]. All weights in VGGNet need around 56.09MB. Each data sample needs 3.072KB. The size of VGGNet equals around 18200 data samples. The used VGGNet architecture is presented as,
2×128C3 - MP2 - 2×256C3 - MP2 - 2×512C3 - MP2 - 2×1024FC - 10SVM.

5.5.1.3 ResNet56 on CIFAR100

Similar to CIFAR10, the CIFAR100 dataset [KNH09] consists of 32×32 color images in 100 object classes. It contains a training dataset with 50000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with size of 128 training on 1 GeForce RTX 3090 GPU. The number of training epochs is chosen as 100 at each round. We use Nesterov SGD with weight decay 0.0001 as the optimizer. The initial learning rate is 0.1, and it decays with the cosine annealing schedule. The ResNet56 used in our experiments is proposed in [HZRS16]. All weights in ResNet56 need around 3.44MB. Each data sample needs 3.072KB. The size of ResNet56 equals around 1100 data samples.

5.5.1.4 MobileNetV1 on ImageNet

The ImageNet dataset [RDS+15] consists of high-resolution color images in 1000 object classes. It contains a training dataset with 1.28 million data samples, and a validation dataset with 50000 data samples. Following the commonly used pre-processing [Pyt19b], each sample (single image) is randomly resized and cropped into a 224×224 color image. We use the original training dataset for training; and randomly select 15000 samples in the original validation dataset for validation, and the rest 35000 samples for testing. We use a mini-batch with size of 1024 training on 4 GeForce RTX 3090 GPUs. The number of training epochs is chosen as 100 at each round. We use Nesterov SGD with weight decay 0.0001 as the optimizer. The initial learning rate is 0.5, and it decays with the cosine annealing schedule. The MobileNetV1 used in our experiments is proposed in [HZC+17]. All weights in MobileNetV1 need around 16.93MB. Each data sample needs 150.528KB. The size of MobileNetV1 equals around 340 data samples.

5.5.2 Ablation Studies on Full Updating

Settings. In this section, we compare full updating with different initialization at each round to confirm the best-performed full updating baseline. The compared full updating methods include, (*i*) the network is trained from a random initialization at each round; (*ii*) the network is trained from a same random initialization at each round, i.e., with a same random seed; (*iii*) the network is trained from the weights $\boldsymbol{w}^{r-1}$ of the last round at each round. The experiments are conducted on VGGNet using CIFAR10 dataset with different amounts of training samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$. Each experiment runs for three times using random data samples and different random seeds.

Results. We report the mean and the standard deviation of test accuracy (over three runs) under different initialization in Figure 5.3. The results show that training from a same random initialization yields a similar accuracy level while sometimes also a lower variance, as training from a (different) random initialization at each round. In comparison to training from scratch (i.e., random initialization), training from $\boldsymbol{w}^{r-1}$ may yield a higher accuracy in the first few rounds; yet training from scratch can always outperform after a large number of rounds. Thus, in this chapter, we adopt training from a same random initialization at each round, i.e., (*ii*), as the baseline of full updating.

5.5.3 Number of Rounds for Re-Initialization

Settings. In these experiments, we re-initialize the network every n rounds under different partial updating settings to determine a heuristic

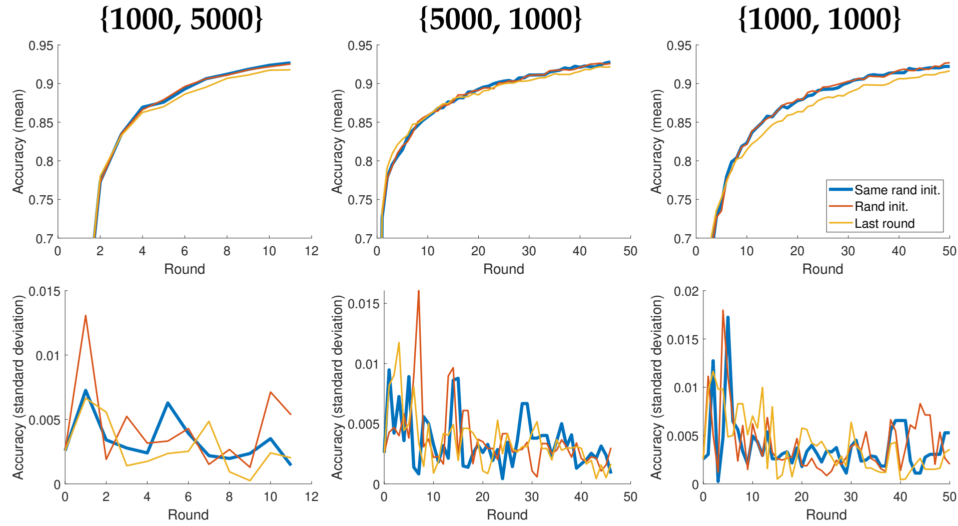

Figure 5.3: Comparing full updating methods with different initialization methods at each round.

rule to set the number of rounds for re-initialization. We conduct experiments on VGGNet using CIFAR10 dataset, with different amounts of training samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and different updating ratios k. Every n rounds, the network is (re-)initialized again from a same random network (as mentioned in 5.5.2), then partially updated in the next n rounds with Algorithm 5.2. We choose $n = 1, 5, 10, 20$. Specially, $n = 1$ means that the network is partially updated from the same random network every round, i.e., without reusing the learned knowledge at all. Each experiment runs three times using random data samples.

Results. We plot the mean test accuracy along rounds in Figure 5.4. By comparing $n = 1$ with other settings, we can conclude that within a certain number of rounds, the current deployed network $\boldsymbol{w}^{r-1}$ (i.e., the network from the last round) is a better starting point for Algorithm 5.2 than a randomly initialized network, i.e., partially updating from the last round may yield a higher accuracy than partially updating from a random network. This is straightforward, since such a network is already pretrained on a subset of the currently available data samples, and the previous learned knowledge could help in the new training process. Since any newly collected samples are continuously stored in the server, complete information about all past data samples is available. This also makes our setting different from continual learning setting, which aims at avoiding catastrophic forgetting without accessing (at least not all) old data.

Each time the network is re-initialized, the new partially updated network might suffer from an accuracy drop in a few rounds. Although this accuracy drop may be relieved if we carefully tune the partial updating training scheme every time, this is not feasible regarding the

large number of updating rounds. However, we can simply avoid such an accuracy drop by not updating the network if the validation accuracy does not increase compared to the last round (as discussed in Section 5.5). Note that in this situation, the partially updated weights (as well as the random seed for re-initialization) still need to be sent to the edge devices, since this is an on-going training process.

After implementing the above strategy, we plot the mean accuracy in Figure 5.5. In addition, we also add the related results on full updating in Figure 5.5, where the network is re-initialized every n rounds from a same random network. Note that full updating with re-initialization every round ($n = 1$) is exactly the same as "same rand init." in Figure 5.3 in 5.5.2. From Figure 5.5, we can conclude that the network needs to be re-initialized more frequently in the first several rounds than in the following rounds to achieve a higher accuracy level. The network also needs to be re-initialized more frequently with a large partial updating ratio k. Particularly, the ratio between the number of current data samples and the number of following collected data samples has a larger impact than the updating ratio.

Thus, we propose to re-initialize the network as long as the number of total newly collected data samples exceeds the number of samples when the network is re-initialized last time. For example, assume that at round r the model is randomly (re-)initialized and partially updated from this random network on dataset $\mathcal{D}^r$. Then, the model will be re-initialized at round $r + n$, if $|\mathcal{D}^{r+n}| > 2 \cdot |\mathcal{D}^r|$.

5.5.4 Impacts from Global/Local Contributions

5.5.4.1 Balancing between Global and Local Contributions

Settings. In Eq. (5.10), the combined contribution is calculated by adding both normalized contributions together. However, both normalized contributions may have different importance when determining the critical weights. In order to investigate which one plays a more essential role in the combined contribution, we introduce another hyper-parameter λ to tune the proportion of both normalized contributions as

$$\boldsymbol{c}_\lambda = \lambda \cdot \frac{1}{\mathbf{1}^\mathrm{T} \cdot \boldsymbol{c}^{\mathrm{global}}} \boldsymbol{c}^{\mathrm{global}} + (1 - \lambda) \cdot \frac{1}{\mathbf{1}^\mathrm{T} \cdot \boldsymbol{c}^{\mathrm{local}}} \boldsymbol{c}^{\mathrm{local}} \tag{5.11}$$

Note that the combined contribution $\boldsymbol{c}$ used in the previous experiments is the same as $\boldsymbol{c}_\lambda$ when $\lambda = 0.5$, since only the order matters when determining the critical weights. We implement partial updating methods with the rewinding metric $\boldsymbol{c}_\lambda$ under different values of λ. We compare these methods under updating ratios $k = 0.01, 0.05, 0.1$ and different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ settings on VGGNet using CIFAR10 dataset, and with the re-initialization scheme described in Section 5.4.2. Each experiment runs three times using random data samples.

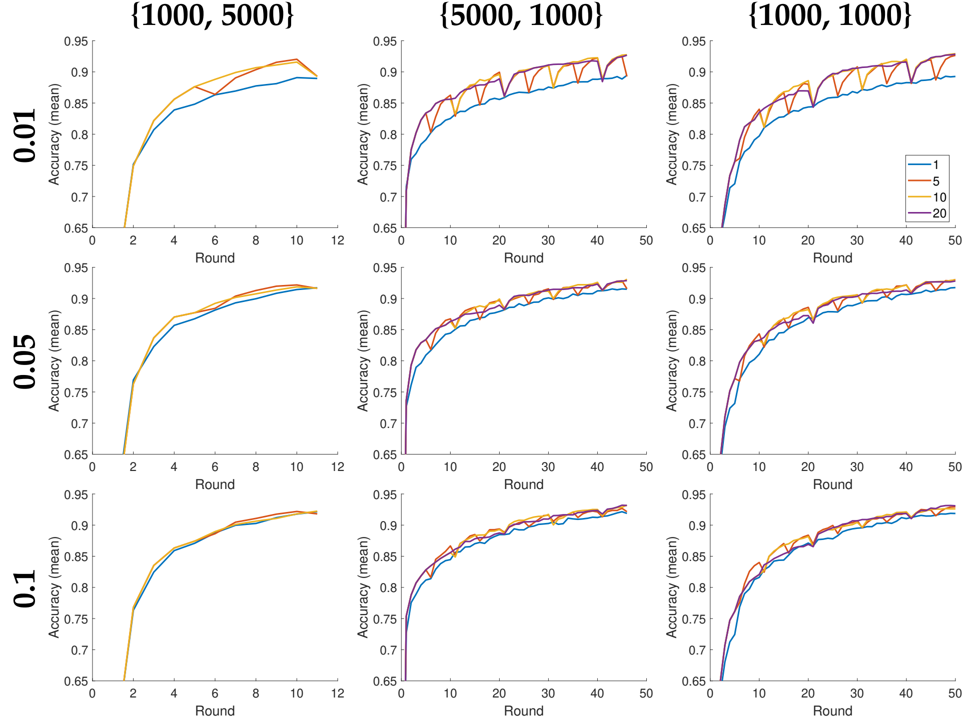

Figure 5.4: Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds ($n = 1, 5, 10, 20$) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

Results. To clearly illustrate the impact of λ, we compare the difference between the accuracy under partial updating methods with various λ and that under full updating. The mean accuracy difference (over three runs) are plotted in Figure 5.6. As seen in Figure 5.6, $\lambda = 0.5$ always obtains the best performance in general, especially when the updating ratio is small. Thus, in the following experiments, we fix this hyper-parameter λ as 0.5. In other words, the combined contribution is chosen as

$$\boldsymbol{c}_\lambda(\lambda = 0.5) = 0.5 \cdot \frac{1}{\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\text{global}}} \boldsymbol{c}^{\text{global}} + 0.5 \cdot \frac{1}{\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\text{local}}} \boldsymbol{c}^{\text{local}} \tag{5.12}$$

which has exactly the same functionality as Eq. (5.10). Note that it may be possible to manually find another hyper-parameter λ that achieves better performance in certain cases. However, setting λ as 0.5 already yields a satisfactory performance in general, and can avoid meticulous and computationally expensive hyper-parameter tuning.

5.5.4.2 Ablation Study of Rewinding Metrics

Settings. We conduct a set of ablation experiments regarding different rewinding metrics discussed in Section 5.4.1. We compare the influence of the local and global contributions as well as their combination, in terms

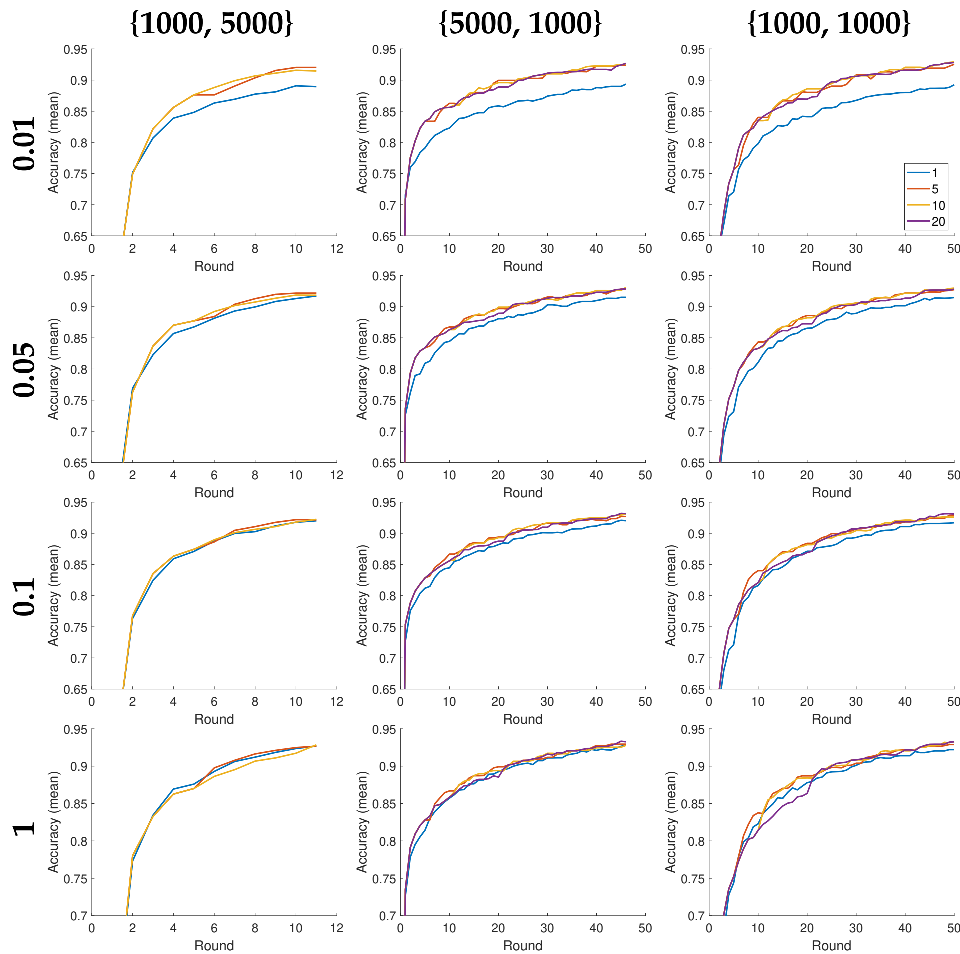

Figure 5.5: Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds ($n = 1, 5, 10, 20$) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$ and full updating $k = 1$) settings.

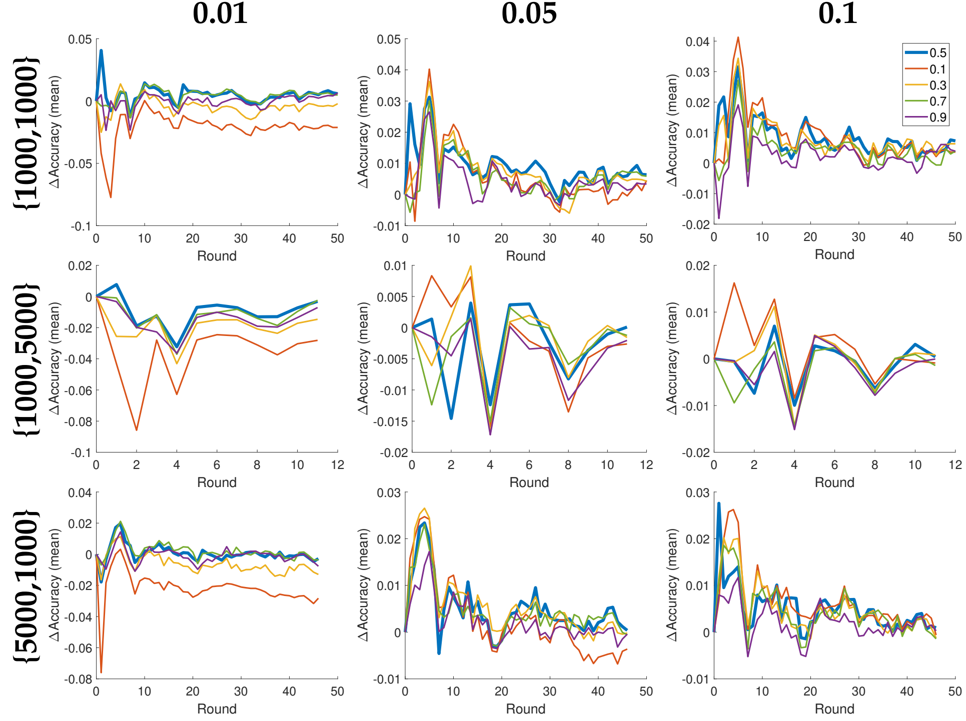

Figure 5.6: Comparison w.r.t. the mean accuracy difference (full updating as the reference) under $\lambda = 0.5, 0.1, 0.3, 0.7, 0.9$. The chosen settings are updating ratios $k = 0.01, 0.05, 0.1$, $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\} = \{1000, 1000\}, \{1000, 5000\}, \{5000, 1000\}$.

k	Training loss at $\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}$ (Test accuracy at $\widetilde{\boldsymbol{w}}$)		
	Global	Local	Combined
0.01	3.04 ± 0.07 (55.0 ± 0.1%)	**2.59** ± 0.08 (55.6 ± 0.1%)	2.66 ± 0.09 (**56.5** ± 0.0%)
0.05	2.51 ± 0.06 (57.3 ± 0.2%)	1.80 ± 0.10 (57.8 ± 0.1%)	**1.67** ± 0.06 (**58.2** ± 0.1%)
0.1	2.03 ± 0.05 (58.3 ± 0.0%)	1.34 ± 0.08 (59.0 ± 0.1%)	**0.99** ± 0.03 (**59.0** ± 0.1%)
0.2	1.20 ± 0.05 (59.0 ± 0.1%)	0.74 ± 0.03 (59.6 ± 0.2%)	**0.42** ± 0.01 (**60.1** ± 0.2%)

Table 5.1: Comparing training loss after rewinding and the final test accuracy under different metrics.

of the training loss after the rewinding and the final test accuracy. We conduct single-round updating on VGGNet. The initial model are fully trained on a randomly selected dataset of 10^3 samples. After adding 10^3 new randomly selected samples, we conduct the first step of our approach with all three rewinding metrics, i.e., global contribution, local contribution, and combined contribution. Accordingly, the second step (sparse fine-tuning) is executed. The experiment is executed over five runs with different random seeds.

Results. The training loss after rewinding (i.e., $\ell(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m})$) and the final test accuracy after sparse fine-tuning (i.e., at $\widetilde{\boldsymbol{w}}$) are reported in Table 5.1. We use the form of mean ± standard deviation. As seen

in the table, the combined contribution always yields a lower or similar training loss after rewinding compared to the other two metrics. The smaller deviation also indicates that adopting the combined contribution yields more robust results. This demonstrates the effectiveness of our proposed metric, i.e., the combined contribution to the analytical upper bound on loss reduction. Rewinding with the combined contribution also acquires a higher final accuracy, which in turn verifies the hypothesis we made for partial updating, a weight shall be updated only if it has a large contribution to the loss reduction.

5.5.4.3 Number of Updated Weights across Layers under Different Rewinding Metrics

Settings. To further study the impact of adopting different rewinding metrics, we show the distribution of updated weights across layers in this section. We implement partial updating methods with three rewinding metrics (i.e., global contribution, local contribution, and combined contribution, see in Section 5.4.1) on VGGNet using CIFAR10 dataset. We compare these methods with different updating ratios k under $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\} = \{1000, 1000\}$. To study the distribution of updated weights along all rounds, we let the network partially updated in every round even if the validation accuracy may not increase compared to the last round. All methods start from the same randomly initialized network, and are re-initialized with this random network according to the proposed scheme in Section 5.4.2.

Results. We plot the number of updated weights across all layers along rounds, under updating ratio $k = 0.01, 0.05, 0.1$ in Figure 5.7, Figure 5.8, and Figure 5.9, respectively. We also plot the corresponding test accuracy along rounds in Figure 5.10. Generally, the metric of local contribution updates more weights in the first several layers and the last layer while with a large variance along rounds. On the contrary, global contribution selects more weights in the last several layers (until the penultimate layer) to update. Combined contribution (the sum of normalized local/global contribution) achieves a more robust and balanced distribution of updated weights across layers than the other contributions. It also results in the highest accuracy level especially under a small updating ratio. Intuitively, local contribution can better identify critical weights w.r.t. the loss during training, while global contribution may be more robust for a highly non-convex loss landscape. Both metrics may be necessary when selecting weights to rewind. Note that the proposed combined contribution is not the simple averaging of both local and global contribution. For example, in “layer 6” of Figure 5.9, the number of updated weights by combined contribution already exceeds the other two metrics.

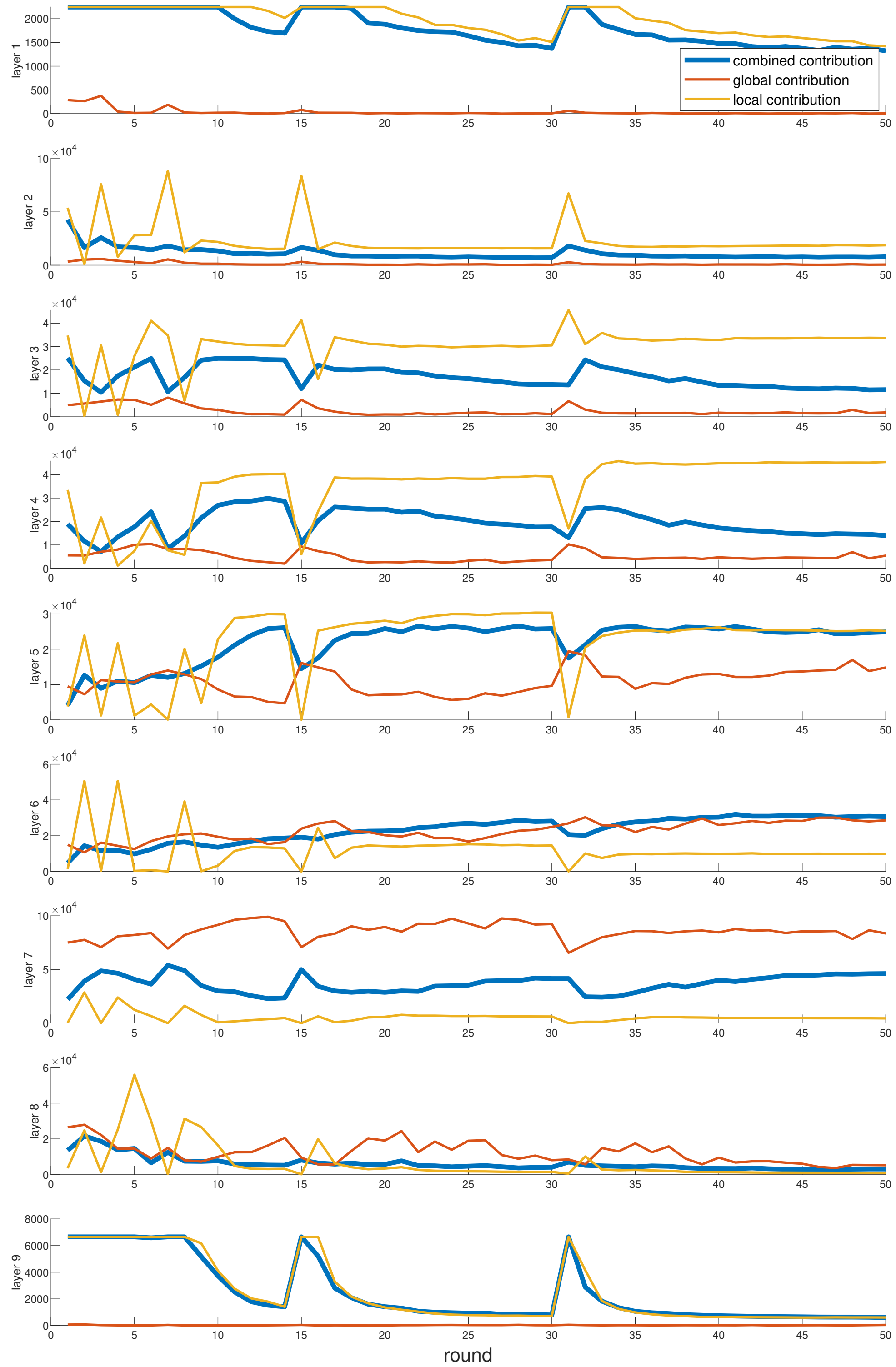

Figure 5.7: Number of updated weights across all layers (VGGNet) when adopting different rewinding metrics (updating ratio $k = 0.01$).

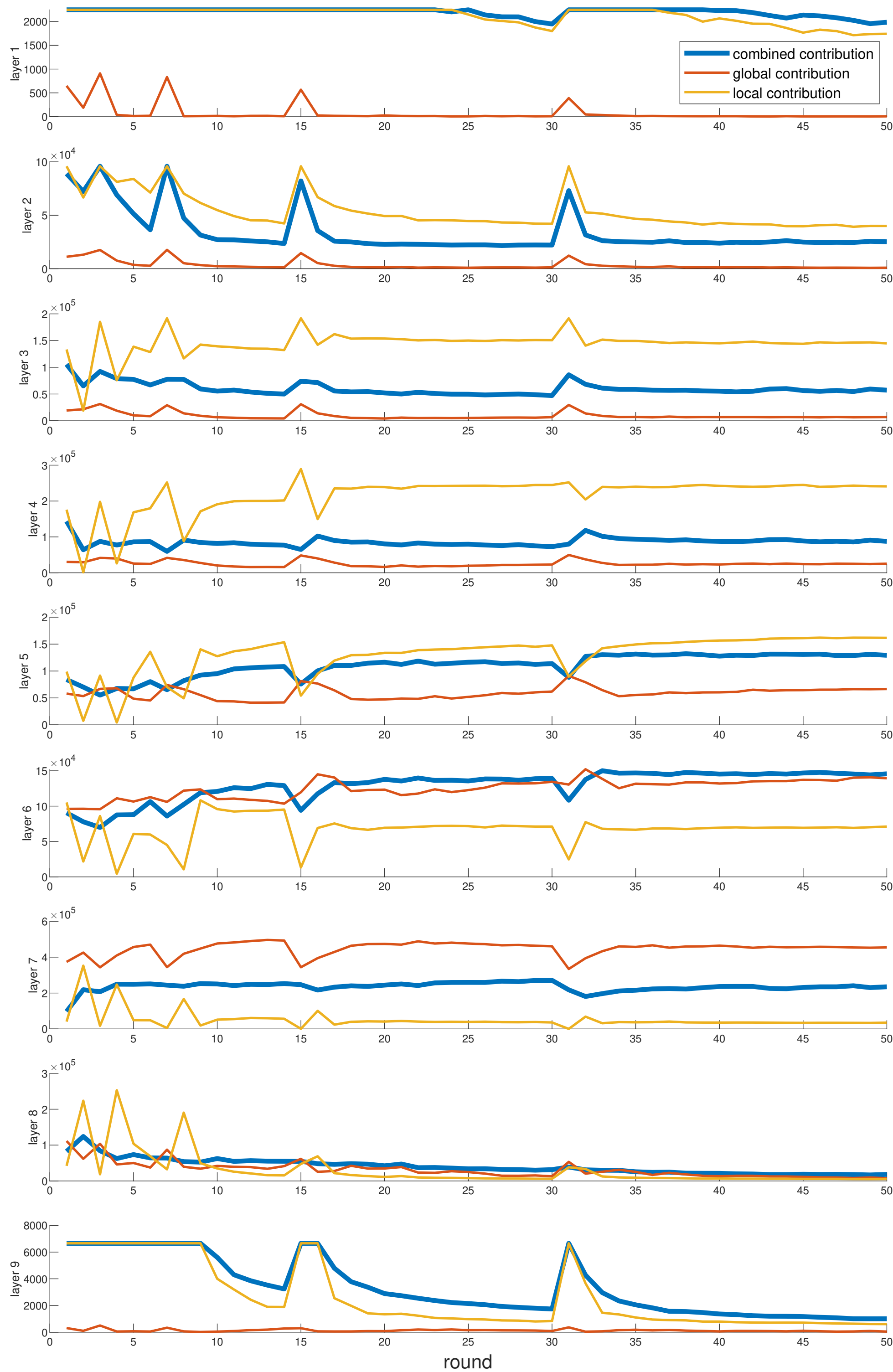

Figure 5.8: Number of updated weights across all layers (VGGNet) when adopting different rewinding metrics (updating ratio $k = 0.05$).

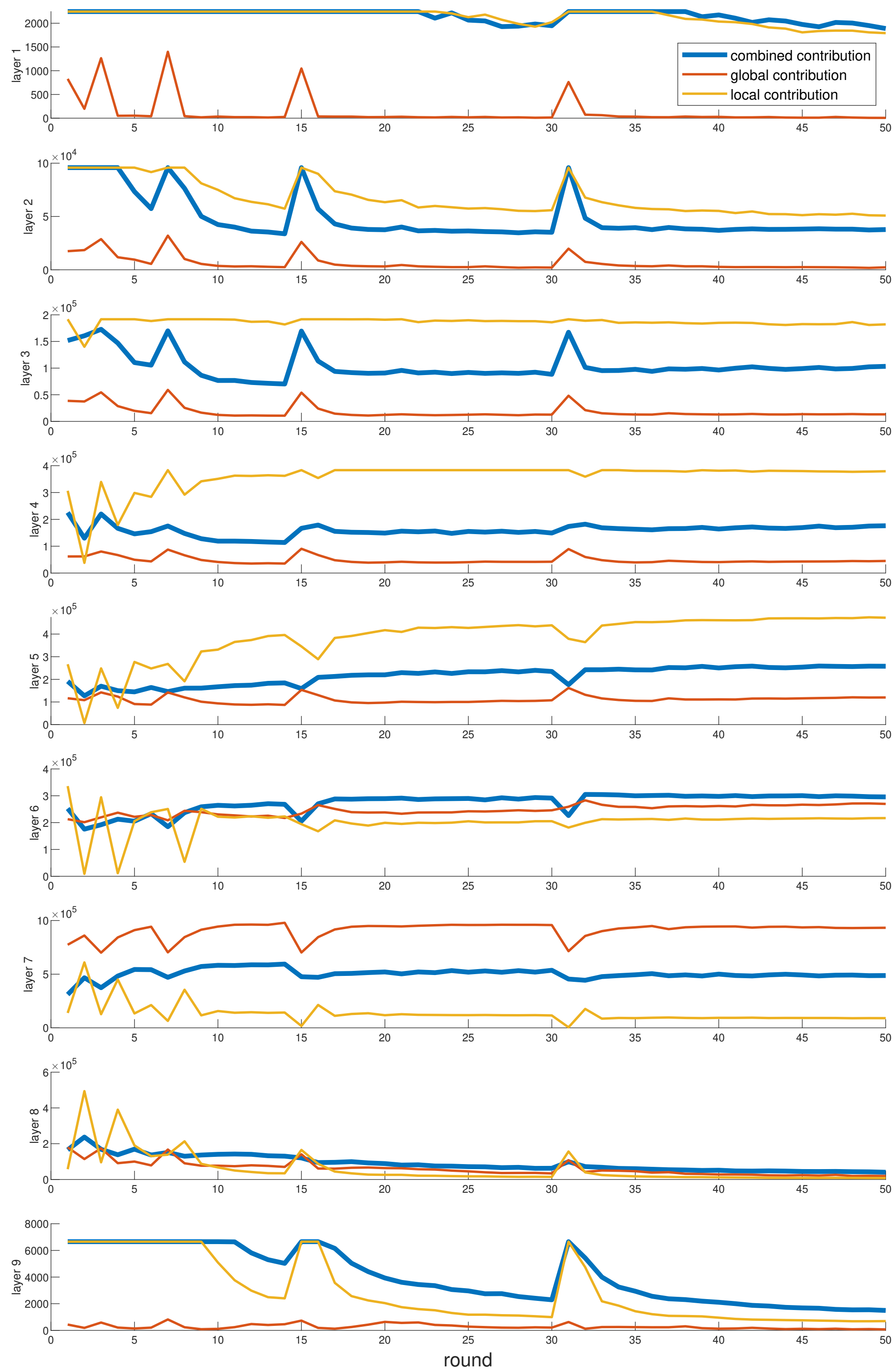

Figure 5.9: Number of updated weights across all layers (VGGNet) when adopting different rewinding metrics (updating ratio $k = 0.1$).

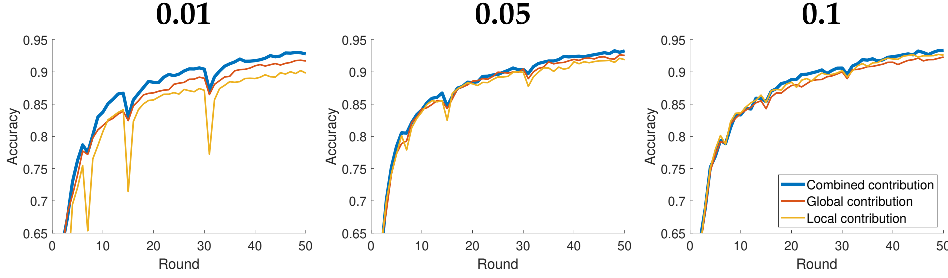

Figure 5.10: The test accuracy of partial updating methods with different rewinding metrics (updating ratio $k = 0.01, 0.05, 0.1$).

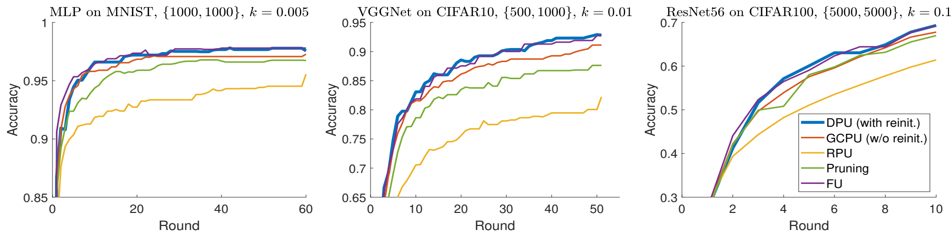

Figure 5.11: DPU is compared with other baselines on different benchmarks in terms of the test accuracy during multi-round updating.

5.5.5 Benchmarking Multi-Round Updating

Settings. To the best of our knowledge, this is the first work on studying weight-wise partial updating a network using newly collected data in iterative rounds. Therefore, we developed three baselines for comparison, including (*i*) full updating (FU), where at each round the network is fully updated with a random initialization (i.e., training from scratch, which yields a better performance as discussed in Section 5.4.2); (*ii*) random partial updating (RPU), where the network is trained from $\boldsymbol{w}^{r-1}$, while we randomly fix each layer's weights with a ratio of $(1-k)$ and sparsely fine-tune the rest; (*iii*) global contribution partial updating (GCPU), where the network is trained with Algorithm 5.1 without re-initialization described in Section 5.4.2; (*iv*) a state-of-the-art unstructured pruning method [RFC20], where the network is first trained from a random initialization at each round, then conducts one-shot magnitude pruning, and finally is sparsely fine-tuned with learning rate rewinding. The ratio of nonzero weights in pruning is set to the same as the updating ratio k to ensure the same communication cost. The experiments are conducted on different benchmarks as mentioned earlier.

Results. We report the test accuracy of the network $\boldsymbol{w}^r$ along rounds in Figure 5.11. All methods start from the same $\boldsymbol{w}^0$, an entirely randomly initialized network. As seen in this figure, DPU clearly yields the highest accuracy in comparison to the other partial updating schemes on different benchmarks. For example, DPU can yield a final Top-1 accuracy of

Method	Average accuracy difference			Ratio of communication cost		
	MLP	VGGNet	ResNet56	MLP	VGGNet	ResNet56
DPU	$\mathbf{-0.17\%}$	$\mathbf{+0.33\%}$	$\mathbf{-0.23\%}$	0.0071	0.0183	0.1147
GCPU	−0.72%	−1.51%	−1.67%	0.0058	0.0198	0.1274
RPU	−4.04%	−11.35%	−6.81%	0.0096	0.0167	0.1274
Pruning [RFC20]	−1.45%	−4.35%	−2.25%	0.0106	0.0141	0.1274

Table 5.2: The average accuracy difference over all rounds and the ratio of communication cost over all rounds related to full updating.

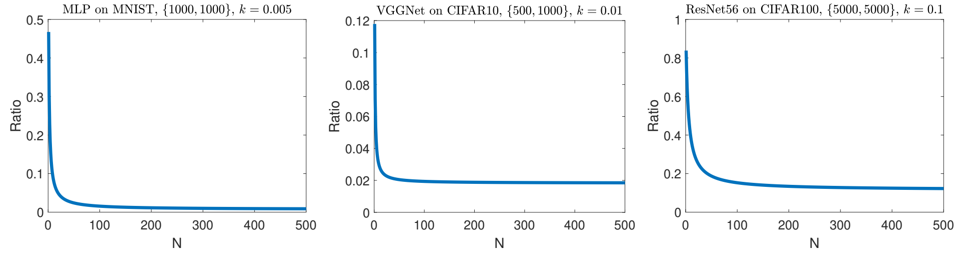

Figure 5.12: The ratio, between the total communication cost (over all rounds) under DPU and that under full updating, varies with the number of nodes N.

92.85% on VGGNet, even exceeds the accuracy (92.73%) of full updating, while GCPU, RPU, and Pruning only acquire 91.11%, 82.21%, and 87.62% respectively. In addition, we compare three partial updating schemes in terms of the accuracy difference related to full updating averaged over all rounds, and the ratio of the communication cost over all rounds related to full updating in Table 5.2. As seen in the table, DPU reaches a similar accuracy as full updating, while incurring significantly fewer transmitted data sent from the server to each edge node. Specially, DPU saves around 99.3%, 98.2% and 88.5% of transmitted data on MLP, VGGNet, and ResNet56, respectively (95.3% in average). The communication cost ratios shown in Table 5.2 differ a little even for the same updating ratio. This is because if the validation accuracy does not increase compared to the previous round, the model will not be updated to reduce the communication overhead (as discussed in Section 5.5). The horizontal straight line segments in Figure 5.11 represent those non-updated rounds under each method.

5.5.5.1 Experiments on Total Communication Cost Reduction

Settings. In this section, we show the benefit due to DPU in terms of *the total communication cost reduction*, as DPU has no impact on the edge-to-server communication which may involve sending newly collected data samples on nodes. The total communication cost includes both edge-to-server communication and server-to-edge communication. This

experimental setup assumes that all data samples in $\delta\mathcal{D}^r$ are collected by N edge nodes during all rounds and sent to the server on a per-round basis. In other words, the first stage (see in Section 5.1) is anyway necessary for sending new training data to the server. For clarity, let S_d denote the data size of each training sample. During round r, we define the per-node total communication cost under DPU as $S_d \cdot |\delta\mathcal{D}^r|/N + (S_w \cdot k \cdot I + S_x(k) \cdot I)$. Similarly, the per-node total communication cost under full updating is defined as $S_d \cdot |\delta\mathcal{D}^r|/N + S_w \cdot I$.

In order to simplify the demonstration, we consider the scenario where N nodes send a certain amount of data samples to the server in $R-1$ rounds, namely $\sum_{r=2}^{R} |\delta\mathcal{D}^r|$ (see Section 5.4.2). Thus, the average data size transmitted from each node to the server in all rounds is $\sum_{r=2}^{R} S_d \cdot |\delta\mathcal{D}^r|/N$. A larger N implies a fewer amount of transmitted data from each node to the server.

Results. We report the ratio of the total communication cost over all rounds required by DPU related to full updating, when DPU achieves a similar accuracy level as full updating (corresponding to three evaluations in Figure 5.11). The ratio clearly depends on $\sum_{r=2}^{R} S_d \cdot |\delta\mathcal{D}^r|/N$, i.e., the number of nodes N. The relation between the ratio and N is plotted in Figure 5.12.

We observe that DPU can still achieve a significant reduction on the total communication cost, e.g., reducing up to 88.2% even for the worst case (a single node). Single node corresponds to the largest data size during edge-to-serve transmission per node, i.e., the worst case. Moreover, DPU tends to be more beneficial when the size of data transmitted by each node to the server becomes smaller. This is intuitive because in this case the server-to-edge communication (thus the reduction due to DPU) dominants in the entire communication.

5.5.6 Different Number of Data Samples and Updating Ratios

Settings. In this section, we show that DPU outperforms other baselines under varying number of training samples and updating ratios in multi-round updating. We also conduct ablations concerning the re-initialization of weights discussed in Section 5.4.2. We implement DPU with and without re-initialization, GCPU with and without re-initialization, RPU, and Pruning [RFC20] (see more details in Section 5.5.5) on VGGNet using CIFAR10 dataset. We compare these methods with different amounts of samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and different updating ratios k. Without further notations, each experiment runs three times using random data samples.

Results. We compare the difference between the accuracy under each partial updating method and that under full updating. The mean accuracy difference (over three runs) is plotted in Figure 5.13. In

addition, we also plot the mean and standard deviation of the absolute accuracy of these methods in respectively. As seen in Figure 5.13, DPU (with re-initialization) always achieves the highest accuracy. DPU also significantly outperforms the pruning method, especially under a small updating ratio. Note that we preferred a smaller updating ratio in our context because it explores the limits of the approach and it indicates that we can improve the deployed network more frequently with the same accumulated server-to-edge communication cost. The dashed curves and the solid curves with the same color can be viewed as the ablation study of our re-initialization scheme. Particularly given a large number of rounds, it is critical to re-initialize the start point $\boldsymbol{w}^{r-1}$ after several rounds (as discussed in Section 5.4.2).

In the first few rounds, partial updating methods almost always yield a higher test accuracy than full updating, i.e., the curves are above zero. This is due to the fact that the amount of available samples is rather small, and partial updating may avoid some co-adaptation in full updating, thus results in a higher test accuracy. Three partial updating methods perform almost randomly in the first round compared to each other, because the limited data are not sufficient to distinguish critical weights from the random initialization $\boldsymbol{w}^0$. This also motivates us to (partially) update the deployed model when new samples are available.

Pruning Weights vs. Pruning Incremental Weights. One of our chosen baselines, global contribution partial updating (GCPU, Algorithm 5.1), could be viewed as a counterpart of the pruning method [RFC20], i.e., pruning the incremental weights with the least magnitudes. Specially, the elements with the smallest absolute values in $\delta\boldsymbol{w}^{\mathrm{f}}$ are set to zero (also rewinding), while the remaining weights are further sparsely fine-tuned with the same learning rate schedule as training $\boldsymbol{w}^{\mathrm{f}}$. In comparison to traditional pruning on weights [RFC20], pruning on incremental weights has a different start point. Traditional pruning on weights first trains randomly initialized weights (a zero-initialized network cannot be trained due to the symmetry), and then prunes the weights with the smallest magnitudes. However, the increment of weights $\delta\boldsymbol{w}^{\mathrm{f}}$ is initialized with zero in Algorithm 5.1, since the first step starts from $\boldsymbol{w}$. By comparing GCPU (with or without re-initialization) with "Pruning", we conclude that retaining previous weights yields better performance than zero-outing the weights.

5.5.7 Benchmarking Single-Round Updating

Settings. To show the versatility of our methods, we test single-round updating on large-scale dataset ImageNet [RDS+15] with iterative rewinding. Single-round DPU is conducted on different (initial) deployed models (MobileNetV1 [HZC+17] as the backbone), including a floating-

#Samples	$\{8 \times 10^5, 4.8 \times 10^5\}$		
	Initial	Vanilla-update	DPU
FP32 Dense	68.5%	70.7% (1)	71.1% (0.22)
50%-Sparse	68.1%	70.5% (0.53)	70.8% (0.22)
INT8	68.4%	70.6% (0.25)	70.6% (0.07)

Table 5.3: The test accuracy of single-round updating on different initial deployed models (MobileNetV1 on ImageNet). The updating ratio k is set to 0.2 in DPU. The ratio of communication cost related to full updating is reported in brackets.

point (FP32) dense model and two compressed models, i.e., a 50%-sparse model and an INT8 quantized model. The sparse model is trained with a state-of-the-art dynamic pruning method [PIVA21]; the quantized model is trained with straight-through-estimator with a output-channel-wise floating point scaling factors similar to [RORF16]. To maintain the same on-device inference cost, partial updating is only applied on nonzero values of sparse models; for quantized models, the updated weights are still in INT8 format. Note that We do not impose sparsity or quantization on batch normalization and bias.

Results. We compare DPU with the vanilla-updates, i.e., the models are trained from scratch with the corresponding methods on all available samples. The test accuracy and the ratio of (server-to-edge) communication cost related to full updating on FP32 dense model are reported in Table 5.3. Results show DPU often yields a higher accuracy than vanilla updating while requiring substantially lower communication cost.

5.6 Summary

In this chapter, we propose a novel pipeline DPU for edge-server system. DPU enables deep learning on edge-server system that has limited on-device resources and limited communication resources. Particularly, when newly collected data samples from edge devices or from other sources are available at the server, the server smartly selects only a subset of critical weights to update at the server-to-edge communication round. This partial updating scheme reduces the redundant updating by reusing the pretrained weights, i.e., the learned knowledge on prior data, which achieves a similar performance as full updating yet with a significantly lower communication cost. The main contributions of DPU are summarized as follows,

- We formalize the deep partial updating paradigm, i.e., how to iteratively perform weight-wise partial updating of the inference

models on remote edge devices, if newly collected training samples are available at the server. This substantially reduces the computation and communication demand on edge devices.

- We propose a novel approach that determines the optimized subset of weights that shall be selected for partial updating, through measuring each weight's contribution to the analytical upper bound on the loss reduction. This simple yet effective metric can be applied to any models that are trained with gradient-based optimizers.

- Experimental results on public vision datasets show that, under the similar accuracy level along the rounds, our approach can reduce the size of the transmitted data by 95.3% on average (up to 99.3%), namely can update the model averagely 21 times more frequent than full updating.

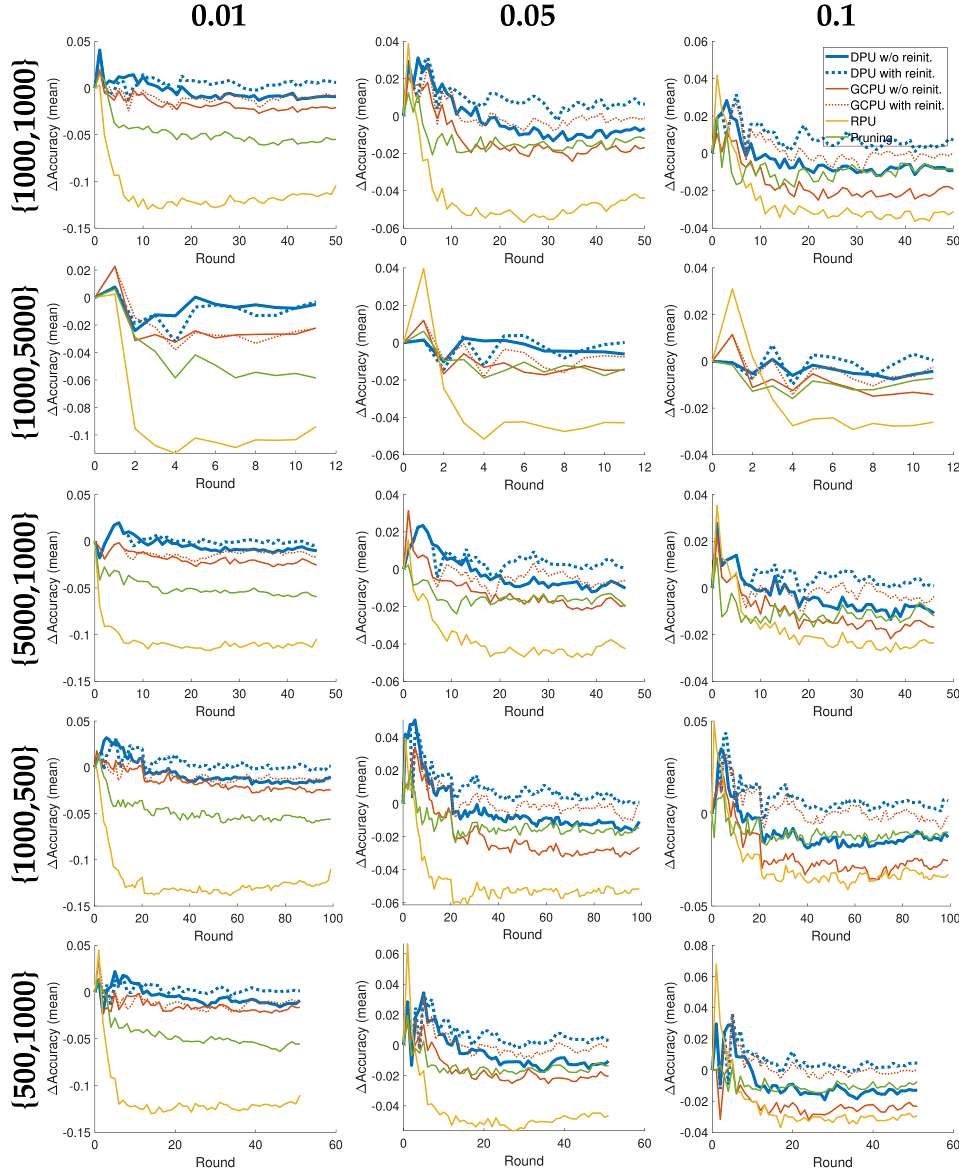

Figure 5.13: Comparison w.r.t. the mean accuracy difference (full updating as the reference) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ (representing the available data samples along rounds) and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

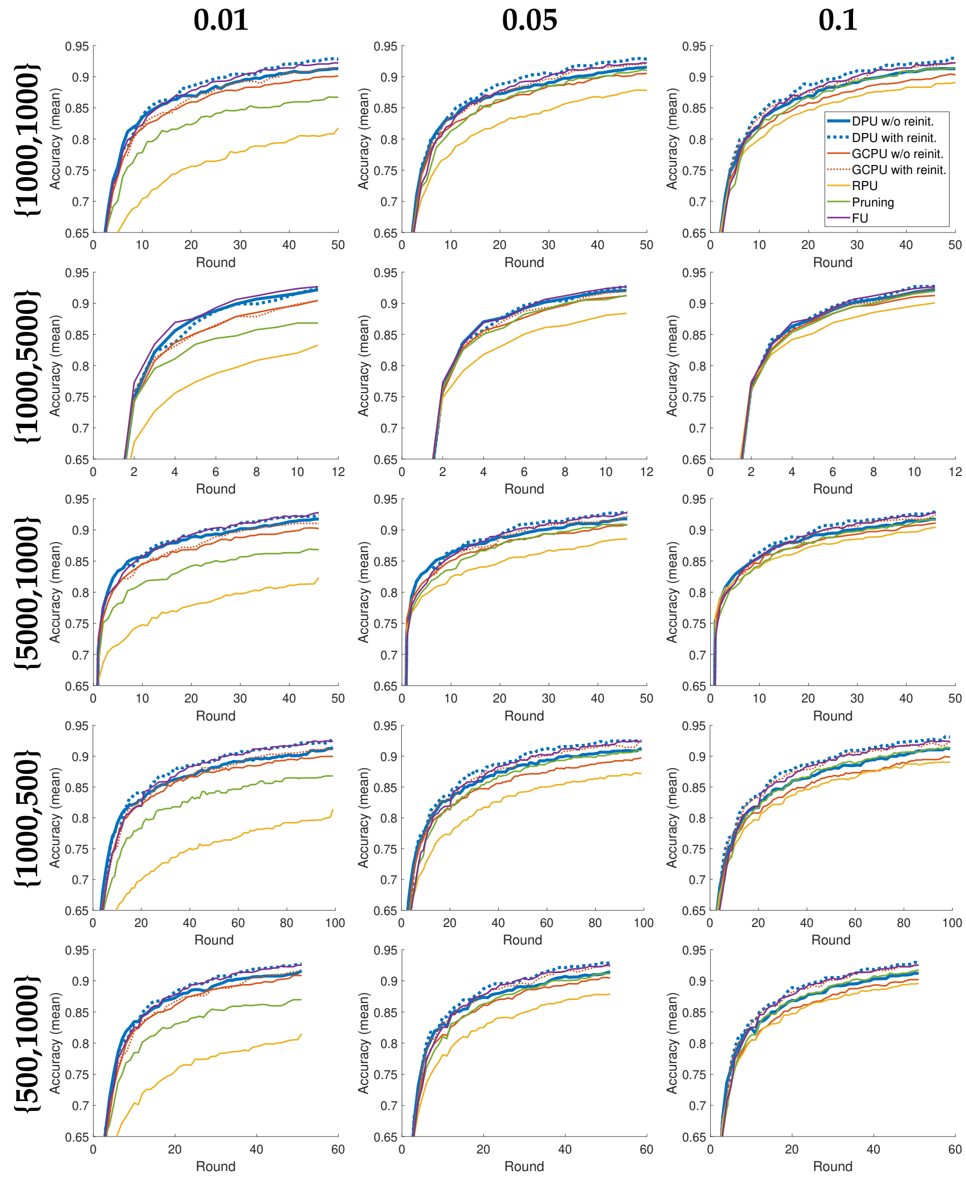

Figure 5.14: Comparison w.r.t. the mean accuracy under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ (representing the available data samples along rounds) and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

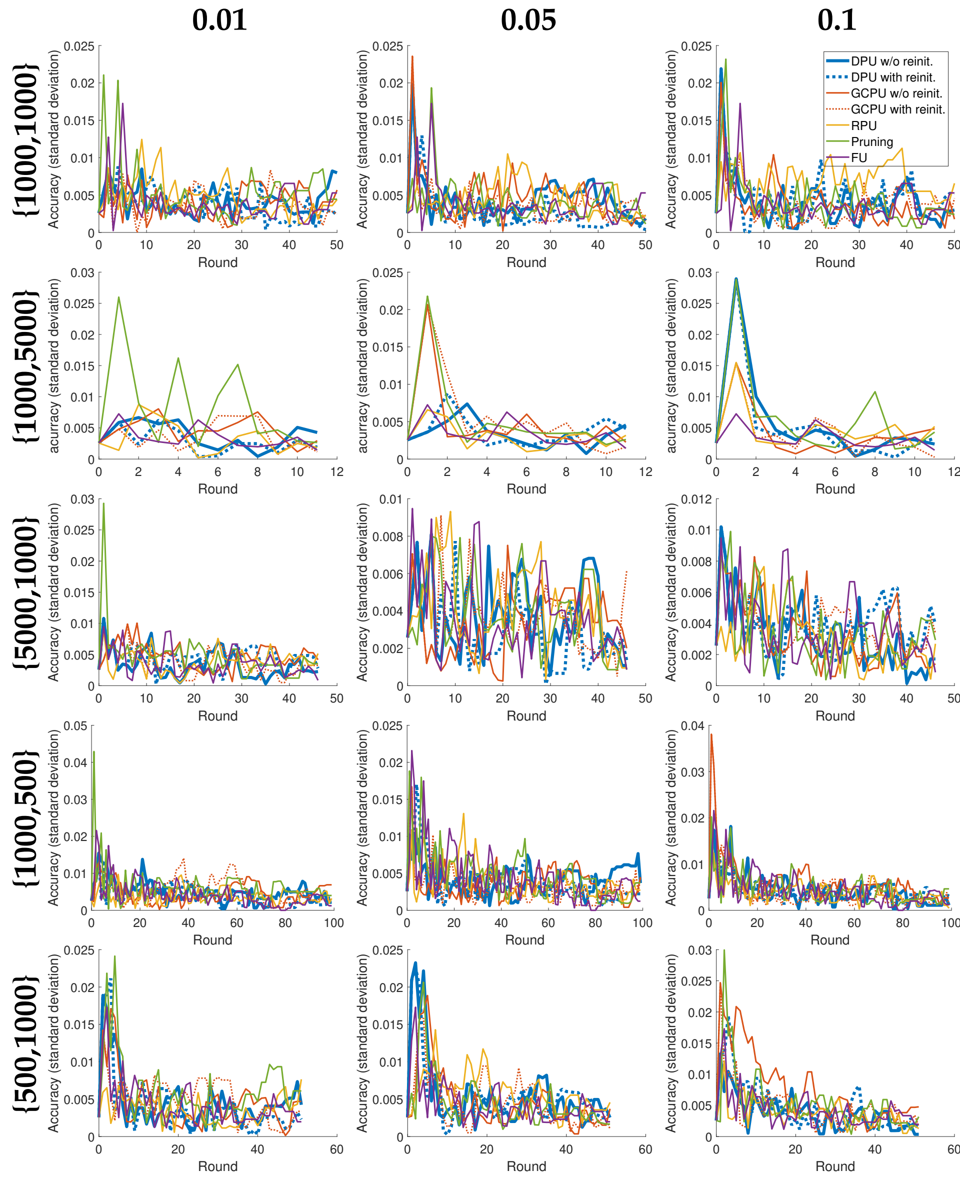

Figure 5.15: Comparison w.r.t. the standard deviation of accuracy under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ (representing the available data samples along rounds) and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

6

Conclusion

State-of-the-art DNNs achieve excellent prediction accuracy in many perception tasks, e.g., computer vision, natural language processing, reinforcement learning, etc. However, a large amount of resources is essential in both the inference phase and the training phase to ensure the high performance of DNNs. Due to the intensive resource demands, DNNs are often deployed on a cloud server with plenty of high-performed computers and shared storage infrastructures.

On the other hand, there is a growing interest to deploy DNNs on edge devices to enable new edge intelligent applications, e.g., AR/VR, mobile assistants, IoT, autonomous driving, etc. In comparison to a cloud server, edge devices have a rather small amount of resources from memory, computation, and energy, and often also a limited scalability. Conventional DNNs need to be compressed in order to fit the resource constraints on edge devices. As DNNs are prone to be over-parameterized, this thesis focuses on reducing the redundancy of DNNs to achieve a better trade-off between resource consumption and model accuracy.

In this thesis, we studied how to enable deep learning on edge devices in four different scenarios. Especially, we studied (*i*) efficient inference on edge devices given fixed resource constraints in Chapter 2, (*ii*) efficient adaptation on edge devices under varying resource constraints in Chapter 3, (*iii*) efficient learning on edge devices with a few training samples of unseen tasks in Chapter 4, and (*iv*) efficient inference and updating on edge-server systems with a constrained communication bus in Chapter 5. Note that different scenarios may have different main resource constraints that hinder us from deploying DNNs on edge devices. According to the main resource constraints in these scenarios, we developed different methodologies to remove the redundant components, such that the compressed DNNs require a lower resource demand while reaching a similar accuracy level as the original ones.

In the following sections of this chapter, we will first summarize our main contributions in each scenario, then discuss the potential directions for future work.

6.1 Contributions

This section summarizes the main contributions of our work in each scenario.

6.1.1 Inference on Edge Devices (Chapter 2)

In Chapter 2, we enabled an efficient inference of DNNs on edge devices. In comparison to cloud inference, inference on edge devices does not need to upload the input data to the cloud server, which can achieve a more stable, fast, and energy-efficient inference, especially with a constrained communication bus. Regarding the main resource constraints from storing a large number of weights and computation during inference, we proposed ALQ, an adaptive loss-aware trained quantizer for multi-bit networks. ALQ reduces the redundancy on the quantization bitwidth.

Unlike prior multi-bit quantization that often assigns an empirical uniform bitwidth, ALQ learns an adaptive bitwidth assignment across different groups of weights according to their loss criticality. ALQ also proposes to optimize the multi-bit quantized weights by directly minimizing the loss function rather than the reconstruction error to the full precision weights. The multi-bit quantized network uses cheaper operations from `xnor` and `popcount` to replace the expensive FLOPs, achieving computation efficiency; the learned adaptive bitwidth yields a smaller average bitwidth by only allocating a high bitwidth to the loss-critical weights, achieving storage efficiency; the direct optimization objective (i.e., the loss) allows us to acquire a quantized network with higher prediction accuracy. In addition, ALQ also enables extremely low-bit networks with an average bitwidth below 1-bit by entirely pruned groups (i.e., 0-bit weights in some groups).

6.1.2 Adaptation on Edge Devices (Chapter 3)

The methods proposed in Chapter 2 are able to compress DNNs for efficient inference if the amount of available resources on edge devices is fixed and known beforehand. However, the resource constraints on the target edge devices may dynamically change during runtime, e.g., the allowed execution time, the allocatable RAM, and the battery energy. To maximize the model accuracy during on-device inference, in Chapter 3, we enabled a DNN with dynamic capacity, such that the DNN can be adapted and executed under varying resource constraints. Particularly, we developed a new synthesis approach DRESS that can sample and

execute sub-networks with different resource demands from a backbone network for on-device inference. DRESS reduces the redundancy among multiple sub-networks by weight sharing and architecture sharing.

DRESS samples sub-networks in a row-based unstructured manner (a.k.a. fine-grained structure sparsity) from the backbone network, and introduces a novel compressed sparse row (CSR) format to utilize sparse tensor computation provided by recent compilation libraries. In DRESS, the nonzero weights of the higher sparsity sub-networks are reused by the lower sparsity sub-networks, achieving memory efficiency; all sparse sub-networks leverage the same architecture as the backbone network, achieving re-configuration efficiency. The sub-networks have different sparsity, and thus can be fetched and executed under various resource constraints.

6.1.3 Learning on Edge Devices (Chapter 4)

In Chapter 2 and Chapter 3, we compressed DNNs to realize an efficient on-device inference under *fixed* and *varying* resource constraints, respectively. However, when facing unseen environments or users on edge devices, it is crucial to retrain the DNN with newly collected data samples to deliver consistent performance and customized services. On the one hand, data samples collected by edge devices are often private and limited; on the other hand, training a DNN often consumes several orders of magnitude more peak memory than inference. Hence, in Chapter 4, we proposed a new meta learning method p-Meta to enable memory-efficient few-shot learning on unseen tasks. p-Meta reduces the updating redundancy by fixing some weights during few-shot learning, which saves the memory consumption that is necessary for the updated weights.

p-Meta enables both data- and memory-efficient on-device learning given unseen tasks, which is realized by automatically identifying adaptation-critical weights during few-shot learning via a meta-trained selection mechanism. p-Meta adopts a hierarchical approach that combines a static selection on adaptation-critical layers and a dynamic selection on adaptation-critical channels. To the best of our knowledge, p-Meta is the first meta learning method designed for on-device few-shot learning. Evaluations on few-shot image classification and reinforcement learning show that p-Meta not only improves the accuracy but also reduces the peak dynamic memory by a factor of 2.5 on average over the state-of-the-art few-shot learning methods.

6.1.4 Edge-Server-System (Chapter 5)

In Chapter 2, Chapter 3 and Chapter 4, we enabled deep learning on a single edge platform in three different scenarios. In Chapter 5, we designed a new pipeline DPU to enable efficient inference and efficient updating for edge-server system. In edge-server system, a set of resource-

constrained edge devices are connected to a remote server with sufficient resources, and some information is allowed to be communicated between edge devices and the server. Due to the limited relevant training data beforehand, pretrained DNNs may be significantly improved after the initial deployment. On such an edge-server system, on-device inference is preferred over cloud inference, since it can achieve a fast and stable inference with less energy consumption. Yet retraining on the cloud server is preferred over on-device retraining (or federated learning) due to the limited memory and computing power on edge devices. Therefore, we proposed a two-stage iterative process to update the deployed inference models, (*i*) at each round, edge devices collect new data samples and send them to the server, and (*ii*) the server retrains the network using collected data, and then sends the updates to each edge device. In comparison to the edge-to-server stage, the transmissions in the server-to-edge stage are highly constrained by the limited communication resource (e.g., bandwidth, energy). Our DPU reduces the server-to-edge communication cost by distinguishing the redundant updating given newly collected samples.

Particularly, DPU studied how to iteratively perform weight-wise partial updating of inference models on remote edge devices, if newly collected training samples are available at the server. In each round, DPU smartly selects and updates a small subset of critical weights that have a large contribution to the loss reduction during the retraining. Experimental results show that DPU can reach a similar accuracy level as full updating yet with a significantly lower communication cost.

6.2 Potential Future Directions

In this section, we discuss some potential directions for the future work. These potential future directions are either some extensions or complementaries of the works presented in the main chapters, or some other edge intelligence scenarios that have not been studied yet due to the time limitation.

6.2.1 Hardware Accelerators of ALQ

ALQ exhibits a high compression ratio on the benchmark evaluations in Chapter 2 without introducing sparse tensor computation. To deploy the multi-bit networks generated by ALQ, the target hardware must support bitwise `xnor` and `popcount` operations for efficient execution. However, the current Arm Cortex CPUs [Arm22] do not include the computation units of `popcount`. Although some software libraries may provide functions for `popcount`, they are less efficient in pipelined computation. Designing some hardware accelerators e.g., with FPGA that can support

bitwise `xnor`, `popcount` and accumulation operations is a promising direction to enable efficient inference with multi-bit networks.

6.2.2 Quantized DRESS

Current DRESS samples sub-networks from a floating-point backbone network. Applying DRESS on a quantized backbone network (e.g., 8-bit integer network) is also worth studying. Especially, the sampled quantized sub-networks can be further accelerated by the fast kernels of sparse quantized computation. For example, CMSIS-NN [LSC18] can achieve a 4× acceleration on 8-bit integer quantized networks compared to 32-bit floating-point networks on a 32-bit Arm Cortex-M CPUs. In addition, it would be also interesting to explore the possibility of applying DRESS on multi-bit quantized networks, i.e., the combination of ALQ and DRESS.

6.2.3 Latency-Aware DRESS

Note also that current DRESS requires predefined sparsity levels. However, a higher sparsity level, i.e., a smaller number of nonzero weights, does not always result in a shorter inference latency [RFC20]. In the future, we encourage the following researchers to build a direct relation between sparsity and inference latency (or energy consumption). This can be realized by (*i*) measuring the inference latency with some hardware simulators, (*ii*) leveraging some real-time models to bound the computation time theoretically. The latency-aware DRESS that does not rely on proxies may fill the gap between the realistic speedup and the theoretical reduction of FLOPs mentioned in Section 3.5.

6.2.4 Low-Precision Few-Shot Learning

In Chapter 4, we introduced p-Meta, a hierarchical structured partial updating on meta-trained models when only a few training samples of new unseen tasks are given. Although p-Meta can dramatically reduce the peak dynamic memory as well as the computation burden during few-shot learning, it still needs full-precision calculation during the backward propagation. As noted in prior works [RA20, CBG+20, WCB+18], adopting a low-precision backward propagation can bring a similar performance as its full-precision versions in the vanilla training. A straightforward future direction is to apply low-precision training on few-shot learning scenarios, where weights, activation, and gradients are all presented in low-precision formats, e.g., 8-bit integer. The step size of 8-bit integer training could be the number of bit shifting, which may be also meta-trained in a per layer per step manner. Conducting 8-bit integer few-shot learning on edge devices can not only further reduce the peak memory consumption, but also speedup the training process in

Benchmark	WRN-28-2 [ZK16] CIFAR10	WRN-28-2 [ZK16] SVHN
Static Storage of Model (MB)	6.02	6.02
Static Storage of Samples (MB)	184.32	305.02

Table 6.1: Static memory of the model and the training samples in example self-supervised learning.

comparison to 32-bit floating-point training.

6.2.5 Streaming Self-Supervised Learning

In Chapter 4, we studied efficient few-shot learning on edge devices, where only a few training samples are given. In some other cases of on-device learning, although the labeled samples are limited due to the limited labor resources, it might be easy to collect a large number of unlabeled samples. Learning a DNN with a small number of labeled samples and a large number of unlabeled samples is known as self-supervised learning (or semi-supervised learning). Current self-supervised learning methods [BCG+19, SBL+20] often need to maintain all unlabeled samples. Even if on small-scale datasets, the static memory for storing samples is much larger than that for storing the self-supervised model. We summarize the static memory consumption for training samples and the self-supervised DNNs in two sample applications in Table 6.1.

We consider that the unlabeled samples are collected in a round-based streaming manner, and during the collection we can query the user for labeling. In this scenario, the main resource constraints are (*i*) the limited number of querying labels, (*ii*) the memory consumption for storing data samples, particularly unlabeled samples. We focus on reducing the redundancy of data samples. We will only select a coreset of unconfident samples to label and a coreset of representative samples to store [KZCI21]. The problem in round r is defined as follows.

Inputs. We have the current optimized model, and the stored datasets from the last round, which contain labeled set $\mathcal{D}_{\mathrm{S}}^{r-1}$ and unlabeled set $\mathcal{D}_{\mathrm{U}}^{r-1}$. We also receive some new unlabeled samples in $\delta\mathcal{D}^r$.

Outputs. We are expected to output the updated model according to newly collected data. We also need to update the datasets. Because of the limited memory and limited querying number, we update the datasets based on two selected coresets $\mathcal{C}_{\mathrm{S}}$ and $\mathcal{C}_{\mathrm{U}}$. Both coresets are selected from all available unlabeled samples, i.e., $\mathcal{C}_{\mathrm{S}}, \mathcal{C}_{\mathrm{U}} \subset \mathcal{D}_{\mathrm{U}}^{r-1} \cup \delta\mathcal{D}^r$.

Methods. In order to select two coresets, we use a confidence score $\boldsymbol{\alpha}$ to weight each unlabeled samples, where $\boldsymbol{\alpha} \in \mathbb{R}_+^{|\mathcal{D}_{\mathrm{U}}^{r-1}|+|\delta\mathcal{D}^r|}$ A larger

α means the sample can better match the learned likelihood, whereas a smaller alpha means the model has less confidence on that sample. Similar to [SBL+20, KZCI21], we also conduct a two-level minmax optimization. Particularly, in the inner loop, the binarized $\boldsymbol{\alpha}$ is used to weight the unsupervised loss, and the model will be then trained with semi-supervised loss. In the outer loop, the confidence score $\boldsymbol{\alpha}$ is optimized on the current labeled dataset $\mathcal{D}_{\mathrm{S}}^{r-1}$ with the optimized model. Both loops are conducted alternatively in several iterations. Then, the current unlabeled samples in $\mathcal{D}_{\mathrm{U}}^{r-1} \cup \delta\mathcal{D}^{r}$ are selected to build two coresets C_{S} and C_{U} according to the optimized score $\boldsymbol{\alpha}$. Note that both coresets C_{S} and C_{U} have a constrained cardinality due to the limited querying number and the limited memory, respectively. The samples in C_{S} will be further queried for labeling. The labeled dataset is then updated as $\mathcal{D}_{\mathrm{S}}^{r} = \mathcal{D}_{\mathrm{S}}^{r-1} \cup C_{\mathrm{S}}$, and the unlabeled dataset is updated as $\mathcal{D}_{\mathrm{U}}^{r} = C_{\mathrm{U}}$.

This concludes my thesis.

Bibliography

[AAH+20] H. Ahmad, T. Arif, M. A. Hanif, R. Hafiz, and M. Shafique. Superslash: A unified design space exploration and model compression methodology for design of deep learning accelerators with reduced off-chip memory access volume. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4191–4204, 2020.

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[ADJ+17] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[AES19] A. Antoniou, H. Edwards, and A. Storkey. How to train your MAML. In *International Conference on Learning Representations (ICLR)*, 2019.

[Arm22] Arm. Arm cpu architecture, 2022. Accessed: 2022-03-20.

[AZK+20] J. T. Ash, C. Zhang, A. Krishnamurthy, J. Langford, and A. Agarwal. Deep batch active learning by diverse, uncertain gradient lower bounds. In *International Conference on Learning Representations (ICLR)*, 2020.

[BCG+19] D. Berthelot, N. Carlini, I. Goodfellow, N. Papernot, A. Oliver, and C. Raffel. Mixmatch: A holistic approach to semi-supervised learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[BMR+20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *arXiv preprint, CoRR*, 2020.

[BS06] S. Brown and C. Sreenan. Updating software in wireless sensor networks: A survey. *Dept. of Computer Science, National Univ. of Ireland, Maynooth, Tech. Rep*, pages 1–14, 2006.

[BSH+21] L. Bernstein, A. Sludds, R. Hamerly, V. Sze, J. Emer, and D. Englund. Freely scalable and reconfigurable optical hardware for deep learning. *Scientific Reports*, 11, 02 2021.

[BSK+19] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. In *International Conference on Machine Learning (ICML), Joint Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations*, 2019.

[Cap20] Capitalone. What is a cluster? an overview of clustering in the cloud, 2020. Accessed: 2022-03-20.

[CBD15] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

[CBG+20] L. Cambier, A. Bhiwandiwalla, T. Gong, O. H. Elibol, M. Nekuii, and H. Tang. Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[CDL+20] Y. Chen, X. Dai, M. Liu, D. Chen, L. Yuan, and Z. Liu. Dynamic convolution: Attention over convolution kernels. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[CGW+20] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations (ICLR)*, 2020.

[CGZH20] H. Cai, C. Gan, L. Zhu, and S. Han. Tiny transfer learning: Towards memory-efficient on-device learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[Cho21] H. Cho. Risa: A reinforced systolic array for depthwise convolutions and embedded tensor reshaping. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[CLP+21] B. Chen, Z. Liu, B. Peng, Z. Xu, J. L. Li, T. Dao, Z. Song, A. Shrivastava, and C. Re. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations (ICLR)*, 2021.

[CMS+20] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko. End-to-end object detection with transformers. *arXiv preprint, CoRR*, 2020.

[CNS20] N. Chatterji, B. Neyshabur, and H. Sedghi. Intriguing role of module criticality in the generalization of deep networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[CWV+18] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan. PACT: parameterized clipping activation for quantized neural networks. *arXiv preprint, CoRR*, 2018.

[CXZG16] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost, 2016.

[CZX21] X. Chu, B. Zhang, and R. Xu. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[DCH+16] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning (ICML)*, 2016.

[DCLT19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

[DdVB15] Y. N. Dauphin, H. de Vries, and Y. Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

[Del18] T. Deleu. Model-Agnostic Meta-Learning for Reinforcement Learning in PyTorch, 2018. Available at: https://github.com/tristandeleu/pytorch-maml-rl.

[DHS11] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[DLH+20] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie. Model compression and hardware acceleration for neural networks: a comprehensive survey. *Proceedings of IEEE*, 108(4):485–532, 2020.

[DWS+19] T. Deleu, T. Würfl, M. Samiei, J. P. Cohen, and Y. Bengio. Torchmeta: A Meta-Learning library for PyTorch, 2019. Available at: https://github.com/tristandeleu/pytorch-meta.

[EAGT19] G. Eraslan, Ž. Avsec, J. Gagneur, and F. J. Theis. Deep learning: new computational modelling techniques for genomics. *Nat Rev Genet*, 20(7):389–403, jul 2019.

[EDGS20] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan. Fast sparse convnets. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[EGM+21] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning (ICML)*, 2021.

[FAL17] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning (ICML)*, 2017.

[FC19] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[FFBL18] J. Faraone, N. J. Fraser, M. Blott, and P. H. W. Leong. SYQ: learning symmetric quantization for efficient deep neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[GBCB16] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[GDG+17] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017.

[GHZ+21] D. Gao, X. He, Z. Zhou, Y. Tong, and L. Thiele. Pruning meta-trained networks for on-device adaptation. In *The Conference on Information and Knowledge Management (CIKM)*, 2021.

[GKSL19] T. Gong, Y. Kim, J. Shin, and S.-J. Lee. Metasense: few-shot adaptation to untrained conditions in deep mobile sensing. In *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.

[GLJ+19] R. Gong, X. Liu, S. Jiang, T. Li, P. Hu, J. Lin, F. Yu, and J. Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[GLYB14] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint, CoRR*, 2014.

[GMD+16] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

[Goo19] Google. Google xnnpack - github repository, 2019. Accessed: 2022-03-20.

[GRG+18] R. Girshick, I. Radosavovic, G. Gkioxari, P. Dollár, and K. He. Detectron, 2018.

[GRUG17] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[GSS17] K. Greff, R. K. Srivastava, and J. Schmidhuber. Highway and residual networks learn unrolled iterative estimation. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[GSZ+20] M. Gooneratne, K. C. Sim, P. Zadrazil, A. Kabel, F. Beaufays, and G. Motta. Low-rank gradient approximation for memory-efficient on-device training of deep neural network. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020.

[Guo18] T. Guo. Cloud-based or on-device: An empirical study of mobile deep inference. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, pages 184–190. IEEE Computer Society, 2018.

[GWRM18] L.-Y. Gui, Y.-X. Wang, D. Ramanan, and J. M. Moura. Few-shot human motion prediction via meta-learning. In *European Conference on Computer Vision (ECCV)*, 2018.

[GYZC17] Y. Guo, A. Yao, H. Zhao, and Y. Chen. Network sketching: exploiting binary structure in deep cnns. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[GZRD20] L. Guerra, B. Zhuang, I. Reid, and T. Drummond. Switchable precision neural networks. *arXiv preprint, CoRR*, 2020.

[HAMS20] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: a survey, 2020.

[HCI+21] I. Hubara, B. Chmiel, M. Island, R. Banner, S. Naor, and D. Soudry. Accelerated sparse neural training: A provable and efficient method to find n:m transposable masks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[HCL+18] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Weinberger. Multi-scale dense networks for resource efficient image classification. In *International Conference on Learning Representations (ICLR)*, 2018.

[HCS+17] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30, 2017.

[HDHB17] H. Hu, D. Dey, M. Hebert, and J. A. Bagnell. Learning anytime predictions in neural networks via adaptive loss balancing. *arXiv preprint, CoRR*, 2017.

[HGDG17] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2017.

[HHS+21] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[HK18] L. Hou and J. T. Kwok. Loss-aware weight quantization of deep networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[HLD+19] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

[HMD16] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.

[Hor14] M. 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)*, 2014.

[HSS18] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[HWC18] Q. Hu, P. Wang, and J. Cheng. From hashing to cnns: Training binary weight networks via hashing. In *The AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[HYK17] L. Hou, Q. Yao, and J. T. Kwok. Loss-aware binarization of deep networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[HZC+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint, CoRR*, 2017.

[HZRS16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[ITF+21] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *International Journal of Robotics Research*, 40(4-5):698–721, 2021.

[JACM20] S. Jung, H. Ahn, S. Cha, and T. Moon. Adaptive group sparse regularization for continual learning. *arXiv preprint, CoRR*, 2020.

[JSL+19] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[KL18] S. Khoram and J. Li. Adaptive quantization of neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[KMA+19] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. Advances and open problems in federated learning. *arXiv preprint, CoRR*, 2019.

[KNH09] A. Krizhevsky, V. Nair, and G. Hinton. Cifar (canadian institute for advanced research), 2009.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2012.

[KZCI21] K. Killamsetty, X. Zhao, F. Chen, and R. Iyer. Retrieve: Coreset selection for efficient and robust semi-supervised learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[LBB+98] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of IEEE*, 86(11):2278–2324, 1998.

[LC10] Y. LeCun and C. Cortes. MNIST handwritten digit database, 2010.

[LCI+19] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann. On-device neural net inference with mobile gpus. *arXiv preprint, CoRR*, 2019.

[LDG+17] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[LHM+18] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations (ICLR)*, 2018.

[LHSG20] G. Lan, B. Heit, T. Scargill, and M. Gorlatova. Gazegraph: graph-based few-shot cognitive context sensing from human visual behavior. In *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2020.

[LKD+17] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations (ICLR)*, 2017.

[LLM+20] F. Li, G. Li, Z. Mo, X. He, and J. Cheng. Fsa: A fine-grained systolic accelerator for sparse cnns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3589–3600, 2020.

[LMB+15] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár. Microsoft coco: Common objects in context. *arXiv preprint, CoRR*, 2015.

[LMW+20] G. Li, X. Ma, X. Wang, L. Liu, J. Xue, and X. Feng. Fusion-catalyzed pruning for optimizing deep learning on intelligent edge devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3614–3626, 2020.

[LMZ+19] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, T. K.-T. Cheng, and J. Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[LN19] S. Lee and S. Nirjon. Neuro.zero: a zero-energy neural network accelerator for embedded sensing and inference systems. In *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.

[LN20] S. Lee and S. Nirjon. Subflow: A dynamic induced-subgraph strategy toward real-time dnn inference and training. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[LRB+19] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling. Relaxed quantization for discretized neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[LSC18] L. Lai, N. Suda, and V. Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint, CoRR*, 2018.

[LSW+20] A. Li, J. Sun, B. Wang, L. Duan, S. Li, Y. Chen, and H. Li. Lotteryfl: Personalized and communication-efficient federated learning with lottery ticket hypothesis on non-iid datasets. *arXiv preprint, CoRR*, 2020.

[LTA16] D. Lin, S. Talathi, and S. Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, 2016.

[LWS+20] B. Li, B. Wu, J. Su, G. Wang, and L. Lin. Eagleeye: Fast sub-net evaluation for efficient neural network pruning. In *European Conference on Computer Vision (ECCV)*, 2020.

[LWW+21] C. Li, G. Wang, B. Wang, X. Liang, Z. Li, and X. Chang. Dynamic slimmable network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[LZL16] F. Li, B. Zhang, and B. Liu. Ternary weight networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

[LZP17] X. Lin, C. Zhao, and W. Pan. Towards accurate binary convolutional neural network. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[MBdG+21] A. Mathur, D. J. Beutel, P. P. B. de Gusmão, J. Fernandez-Marques, T. Topal, X. Qiu, T. Parcollet, Y. Gao, and N. D. Lane. On-device federated learning with flower. In *The Conference on Machine Learning and Systems (MLSys)*, 2021.

[MFCP+19] M. Meyer, T. Farei-Campagna, A. Pasztor, R. D. Forno, T. Gsell, J. Faillettaz, A. Vieli, S. Weber, J. Beutel, and L. Thiele. Event-triggered natural hazard monitoring with convolutional neural networks on the edge. In *Proceedings of International Conference on Information Processing in Sensor Networks (IPSN)*, IPSN '19. Association for Computing Machinery, 2019.

[MKH21] H. R. Mendis, C.-K. Kang, and P.-c. Hsiu. Intermittent-aware neural architecture search. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[MLP+21] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint, CoRR*, 2021.

[MM18] A. Mishra and D. Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *International Conference on Learning Representations (ICLR)*, 2018.

[MNCM18] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr. WRPN: Wide reduced-precision networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[MQC+17] Z. Meng, H. Qin, Z. Chen, X. Chen, H. Sun, F. Lin, and M. H. Ang. A two-stage optimized next-view planning framework for 3-d unknown environment exploration, and structural reconstruction. *IEEE Robotics and Automation Letters*, 2(3):1680–1687, 2017.

[Nes98] Y. Nesterov. Introductory lectures on convex programming volume i: Basic course. *Lecture notes*, 1:25, 1998.

[Nvi22] Nvidia. The power consumption of nvidia (table), 2022. Accessed: 2022-03-20.

[Ope20] OpenAI. Openai's gpt-3 language model: A technical overview, 2020. Accessed: 2022-03-20.

[ORL18] B. N. Oreshkin, P. Rodriguez, and A. Lacoste. Tadam: Task dependent adaptive metric for improved few-shot learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[OSKY21] C. Oh, J. So, S. Kim, and Y. Yi. Exploiting activation sparsity for fast cnn inference on mobile gpus. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[OYKY21] J. Oh, H. Yoo, C. Kim, and S.-Y. Yun. Boil: Towards representation change for few-shot learning. In *International Conference on Learning Representations (ICLR)*, 2021.

[PGC+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *Proceedings of NIPS Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*, 2017.

[PIVA21] A. Peste, E. Iofinova, A. Vladu, and D. Alistarh. Ac/dc: Alternating compressed/decompressed training of deep neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[PTT18] F. Pedersoli, G. Tzanetakis, and A. Tagliasacchi. Espresso: Efficient forward propagation for binary deep neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[PW19] D. S. Pete Warden. *TinyML*. O'Reilly Media, Inc., 2019.

[Pyt19a] Pytorch. Pytorch example of lenet5 on mnist, 2019. Accessed: 2021-09-28.

[Pyt19b] Pytorch. Pytorch example on resnet, 2019. Accessed: 2022-02-15.

[Pyt21] Pytorch. Object detection reference training scripts, 2021. Accessed: 2022-03-20.

[RA20] M. A. Raihan and T. M. Aamodt. Sparse weight activation training. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[RDS+15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[RFC20] A. Renda, J. Frankle, and M. Carbin. Comparing fine-tuning and rewinding in neural network pruning. In *International Conference on Learning Representations (ICLR)*, 2020.

[RHGS15] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

[RKK18] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations (ICLR)*, 2018.

[RORF16] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, 2016.

[RRBV20] A. Raghu, M. Raghu, S. Bengio, and O. Vinyals. Rapid learning or feature reuse? towards understanding the effectiveness of MAML. In *International Conference on Learning Representations (ICLR)*, 2020.

[RTR+18] M. Ren, E. Triantafillou, S. Ravi, J. Snell, K. Swersky, J. B. Tenenbaum, H. Larochelle, and R. S. Zemel. Meta-learning for semi-supervised few-shot classification. In *International Conference on Learning Representations (ICLR)*, 2018.

[SBL+20] K. Sohn, D. Berthelot, C.-L. Li, Z. Zhang, N. Carlini, E. D. Cubuk, A. Kurakin, H. Zhang, and C. Raffel. Fixmatch: Simplifying semi-supervised learning with consistency and confidence. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[SHM+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[SHZ+18] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[SK18] O. Sener and V. Koltun. Multi-task learning as multi-objective optimization. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[SLF18] O. Shayer, D. Levi, and E. Fetaya. Learning discrete weights using the local reparameterization trick. In *International Conference on Learning Representations (ICLR)*, 2018.

[SLM+15] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. In *International Conference on Machine Learning (ICML)*, 2015.

[SLQ+21] Z. Shen, Z. Liu, J. Qin, M. Savvides, and K. Cheng. Partial is better than all: Revisiting fine-tuning strategy for few-shot learning. In *The AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[SMNP21] F. P. Sunny, A. Mirza, M. Nikdast, and S. Pasricha. Robin: A robust optical binary neural network accelerator. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[Sor21] S. Soro. Tinyml for ubiquitous edge ai. *arXiv preprint, CoRR*, 2021.

[SS15] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[SSZ17] J. Snell, K. Swersky, and R. S. Zemel. Prototypical networks for few-shot learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[SWP+18] D. Stowell, M. Wood, H. Pamuła, Y. Stylianou, and H. Glotin. Automatic acoustic detection of birds through deep learning: The first bird audio detection challenge. *Methods in Ecology and Evolution*, 10, 10 2018.

[SYZ+18] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. S. Torr, and T. M. Hospedales. Learning to compare: Relation network for few-shot learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[SZ15] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[SZS+21] W. Sun, A. Zhou, S. Stuijk, R. G. J. Wijnhoven, A. Nelson, H. Li, and H. Corporaal. Dominosearch: Find layer-wise fine-grained n:m sparse schemes from dense neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[Ten] TensorFlow. On-device training with tensorflow lite. Accessed: 2022-01-15.

[TET12] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[VBL+16] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra. Matching networks for one shot learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

[Vel20] Vela. Arm vela compiler, 2020. Accessed: 2022-03-20.

[VOZK+21] J. Von Oswald, D. Zhao, S. Kobayashi, S. Schug, M. Caccia, N. Zucchet, and J. Sacramento. Learning where to learn: Gradient sparsity in meta and continual learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[VT20] T. Verelst and T. Tuytelaars. Dynamic convolutions: Exploiting spatial sparsity for faster inference. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[WBM+10] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona. Caltech-UCSD Birds 200. Technical Report CNS-TR-2010-001, California Institute of Technology, 2010.

[WCB+18] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[WH18] Y. Wu and K. He. Group normalization. In *European Conference on Computer Vision (ECCV)*, 2018.

[Wik22a] Wikipedia. Augmented reality, 2022. Accessed: 2022-03-20.

[Wik22b] Wikipedia. Cloud computing, 2022. Accessed: 2022-03-20.

[Wik22c] Wikipedia. Edge device, 2022. Accessed: 2022-03-20.

[Wik22d] Wikipedia. Internet of things, 2022. Accessed: 2022-03-20.

[Wik22e] Wikipedia. List of nvidia graphics processing units, 2022. Accessed: 2022-03-20.

[Wik22f] Wikipedia. Self-driving car, 2022. Accessed: 2022-03-20.

[Wik22g] Wikipedia. Virtual assistant, 2022. Accessed: 2022-03-20.

[Wik22h] Wikipedia. Virtual reality, 2022. Accessed: 2022-03-20.

[Wil92] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4), may 1992.

[WKM+19] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick. Detectron2, 2019.

[WKMR20] Z. Wang, K. K, S. Mayhew, and D. Roth. Extending multilingual bert to low-resource languages. *arXiv preprint, CoRR*, 2020.

[WSL+18] D. Wan, F. Shen, L. Liu, F. Zhu, J. Qin, L. Shao, and H. Tao Shen. Tbn: Convolutional neural network with ternary inputs and binary weights. In *European Conference on Computer Vision (ECCV)*, 2018.

[WWSH20] Y. Wu, Z. Wang, Y. Shi, and J. Hu. Enabling on-device cnn training by self-supervised instance filtering and error map pruning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3445–3457, 2020.

[XYL+18] C. Xu, J. Yao, Z. Lin, W. Ou, Y. Cao, Z. Wang, and H. Zha. Alternating multi-bit quantization for recurrent neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[YH19a] J. Yu and T. Huang. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv preprint, CoRR*, 2019.

[YH19b] J. Yu and T. Huang. Universally slimmable networks and improved training techniques. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[YJL+20] J. Yu, P. Jin, H. Liu, G. Bender, P.-J. Kindermans, M. Tan, T. Huang, X. Song, R. Pang, and Q. Le. Bignas: Scaling up neural architecture search with big single-stage models. In *European Conference on Computer Vision (ECCV)*, 2020.

[YLDM19] J. Yu, A. Lukefahr, R. Das, and S. Mahlke. Tf-net: Deploying sub-byte deep neural networks on microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[YYLH18] J. Yoon, E. Yang, J. Lee, and S. J. Hwang. Lifelong learning with dynamically expandable networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[YYX+19] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. Slimmable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[ZK16] S. Zagoruyko and N. Komodakis. Wide residual networks. In E. R. H. Richard C. Wilson and W. A. P. Smith, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press, September 2016.

[ZMZ+21] A. Zhou, Y. Ma, J. Zhu, J. Liu, Z. Zhang, K. Yuan, W. Sun, and H. Li. Learning n:m fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations (ICLR)*, 2021.

[ZQD+19] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He. A comprehensive survey on transfer learning. *arXiv preprint, CoRR*, 2019.

[ZST+19] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid. Structured binary neural networks for accurate image classification and semantic segmentation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[ZWN+16] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint, CoRR*, 2016.

[ZYG+17] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. In *International Conference on Learning Representations (ICLR)*, 2017.

[ZYWC18] A. Zhou, A. Yao, K. Wang, and Y. Chen. Explicit loss-error-aware quantization for low-bit deep neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[ZYYH18] D. Zhang, J. Yang, D. Ye, and G. Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *European Conference on Computer Vision (ECCV)*, 2018.

[ZYZ+18] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *European Conference on Computer Vision (ECCV)*, 2018.

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

Zhongnan Qu, Zimu Zhou, Yun Cheng, Lothar Thiele. **Adaptive Loss-aware Quantization for Multi-bit Networks.** *In Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR),* IEEE, 2020, Acceptance ratio: 22.1%. (Chapter 2)

Zhongnan Qu, Syed Shakib Sarwar, Xin Dong, Yuecheng Li, Huseyin Sumbul, Barbara De Salvo. **DRESS: Dynamic REal-time Sparse Subnets.** *In Efficient Deep Learning for Computer Vision (ECV),* CVPRWorkshop, 2022, Acceptance ratio: 29.9%. (Chapter 3)

Zhongnan Qu, Zimu Zhou, Yongxin Tong, Lothar Thiele. **p-Meta: Towards On-device Deep Model Adaptation.** *In Proceedings of ACM Conference on Knowledge Discovery and Data Mining (SIGKDD),* ACM, 2022, Acceptance ratio: 15.0%. (Chapter 4)

Zhongnan Qu, Cong Liu, Lothar Thiele. **Deep Partial Updating: towards Communication Efficient Updating for On-Device Inference.** *In **Submission** to European Conference on Computer Vision (ECCV)* Springer, 2022. (Chapter 5)

The following list includes publications that were written during the PhD studies, yet are not part of this thesis.

Fan Lu, Guang Chen, Yinlong Liu, Zhongnan Qu, Alois Knoll. **RSKDD-Net: Random Sample-based Keypoint Detector and Descriptor.** *In Proceedings of Annual Conference on Neural Information Processing Systems (NeurIPS),* 2020, Acceptance ratio: 20.1%.

Xin Dong, Barbara De Salvo, Meng Li, Chiao Liu, Zhongnan Qu, H.T. Kung, Ziyun Li. **SplitNets: Designing Neural Architectures for Efficient Distributed Computing on Head-Mounted Systems.** *In Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR),* IEEE, 2022, Acceptance ratio: 25.3%.

Bin Li, Hu Cao, Zhongnan Qu, Yingbai Hu, Zhenke Wang, Zichen Liang. **Event-based Robotic Grasping Detection with Neuromorphic Vision Sensor and Event-Stream Dataset.** *Frontiers in Neurorobotics,* 14, Oct 2020.

Curriculum Vitæ

Personal Data

Name	Zhongnan Qu
Date of Birth	May 5, 1992
Citizenship	China

Education

2018–2022	ETH Zurich, Zurich Switzerland Computer Engineering and Networks Laboratory Ph.D. under the supervision of Prof. Dr. Lothar Thiele
2014–2018	TU Munich, Munich Germany M.Sc. in Electrical and Computer Engineering
2014–2017	TU Munich, Munich Germany M.Sc. in Mechanical Engineering
2010–2014	Tongji University, Shanghai China B.Eng. in Mechatronics Engineering
2013–2014	Munich University of Applied Sciences, Munich Germany B.Eng. in Mechatronics Engineering

Professional Experience

2018–2022	ETH Zurich, Zurich Switzerland Research and teaching assistant
2021	Meta Reality Labs, Seattle (Remotely) US Research Intern
2017	BMW Group, Munich Germany Intern
2014	Canon Group Company • Océ Printing Systems GmbH, Munich Germany Intern and Thesis Student
2013	State Grid, Henan China Intern