

Distributed machine learning on Edge computing systems

Di Wu

A thesis submitted for the degree of PhD
at the
University of St Andrews



2024

Full metadata for this thesis is available in
St Andrews Research Repository
at:

<https://research-repository.st-andrews.ac.uk/>

Identifier to use to cite or link to this thesis:

DOI: <https://doi.org/10.17630/sta/1164>

This item is protected by original copyright

This item is licensed under a
Creative Commons Licence

<https://creativecommons.org/licenses/by-nc/4.0/>

Candidate's declaration

I, Di Wu, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 55,000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree. I confirm that any appendices included in my thesis contain only material permitted by the 'Assessment of Postgraduate Research Students' policy.

I was admitted as a research student at the University of St Andrews in January 2022.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Date 06/26/2024

Signature of candidate

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree. I confirm that any appendices included in the thesis contain only material permitted by the 'Assessment of Postgraduate Research Students' policy.

Date 26 June 2024

Signature of supervisor

Permission for publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Di Wu, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

Printed copy

No embargo on print copy.

Electronic copy

No embargo on electronic copy.

Date 06/26/2024

Signature of candidate

Date 06/26/2024

Signature of supervisor

Underpinning Research Data or Digital Outputs

Candidate's declaration

I, Di Wu, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Date 06/26/2024

Signature of candidate

Acknowledgements

General acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr Blesson Varghese. His invaluable guidance, support, and encouragement have been instrumental in completing this research. I still vividly remember how, after I finished the drafts of my research articles, Dr Varghese provided comprehensive feedback and suggestions for revisions, significantly improving the quality of the work. I am incredibly grateful for his mentorship throughout this journey.

I would also like to thank all the members of the Edge Computing Hub: Dr Rehmat Ullah, Dr Peter Kilpatrick, Dr Ivor Spence, Dr Paul Harvey, Dr Philip Rodgers, and Mr Leon Wong. Their collaboration, feedback, and camaraderie have enriched my research experience and contributed to the success of this thesis. I also thank my colleagues, Mr Dhananjay Saikumar, Mr Bailey Eccles, and Mr Zihan Zhang, for their regular input into my research.

I owe a profound debt of gratitude to my wife, Dan Peng, whose unwavering support, patience, and love have been my anchor throughout this challenging journey. To my parents and in-laws, Hong Chen, Jie Wu, Yumei Wang, and Fengxin Peng: Thank you for your constant encouragement, belief in my abilities, and the sacrifices you have made to support my education. I am also grateful to all my friends from the University of St Andrews, Queen's University Belfast, and Zhejiang University of Technology, whose companionship, understanding, and motivation have been a source of strength and inspiration.

Funding

This work was funded by Rakuten Mobile Inc., Japan, on the project entitled, "CASE: Context-aware, Adaptive and Scalable Edge Management".

Abstract

The demand for distributed machine learning (DML) systems, which distribute training workloads across multiple nodes, has surged over the past decade due to the rapid growth of datasets and computational requirements. Additionally, executing ML training at the edge has gained importance for data privacy and reducing communication costs associated with sending raw data to the cloud. These trends have motivated a new ML training paradigm: DML at the edge.

However, implementing DML systems at the edge presents four key challenges: (i) Hardware limited and heterogeneous resources at the edge result in impractical training times; (ii) The communication costs of DML systems at the edge are substantial; (iii) On-device training cannot be carried out on low-end devices; (iv) There is a lack of a comprehensive framework that can tackle the aforementioned challenges and support efficient DML systems at the edge.

This thesis presents four techniques to address the above. First, it proposes an adaptive deep neural network (DNN) partitioning and offloading technique to address limited device resources with cloud assistance. This DNN partitioning-based federated learning (DPFL) system is further optimized by a reinforcement learning agent to adapt to heterogeneous devices. The thesis then introduces the techniques of pre-training initialization and replay buffer to reduce gradient and activation communication, identified as bottlenecks in a DPFL system. Additionally, a dual-phase layer freezing technique is proposed to minimize the on-device computations. Finally, a holistic framework is developed to integrate these techniques, maximizing their application and impact.

The proposed framework supports the building of a new DPFL system that is more efficient than classic DML at the edge. Experimental evaluation on two real-world testbeds across various datasets and model architectures demonstrates the improvements of the proposed DML system on a range of quality and performance metrics, such as final accuracy, training latency, and communication cost.

Table of contents

List of figures	xi
List of tables	xv
Glossary Terms	xix
1 Introduction	1
1.1 Motivation	2
1.1.1 The Emergence of DML	2
1.1.2 Moving DML from the Cloud to the Edge	3
1.2 Research Questions	4
1.2.1 Limited and Heterogeneous Resources	4
1.2.2 Communication Cost	5
1.2.3 On-device Training Cost	6
1.2.4 Lacking A Holistic Framework	6
1.3 Research Methods	7
1.3.1 Adaptive Offloading to Address Resource Limitations and Device Heterogeneity	7
1.3.2 Pre-trained Initialization and Replay Buffer to Reduce Communication Costs	8
1.3.3 Early-Stage and Accuracy-Guaranteed Layer Freezing to Accelerate On-device Training	8
1.3.4 EPFL: A Holistic Framework to Integrate FedAdapt, EcoFed and FedFreeze	9
1.4 Research Contributions	10
1.5 Publications	11
1.6 Thesis Outline	12

2	Literature Review	15
2.1	From Centralized ML to Distributed ML	15
2.1.1	Centralized ML	16
2.1.2	Distributed ML	17
2.1.3	Summary	19
2.2	From Cloud Computing to Edge Computing	19
2.2.1	Cloud Computing	19
2.2.2	Edge Computing	20
2.2.3	Summary	22
2.3	The convergence of DML and EC	22
2.3.1	DML at the Edge	23
2.3.2	Challenges of Implementing DML in EC	24
2.3.3	Existing Solutions	25
2.4	DML Training Paradigms in EC	27
2.4.1	Federated Learning	27
2.4.2	Split Learning	30
2.4.3	DNN Partitioning-Based FL	31
2.4.4	Discussion	32
3	Offloading Workload from IoT Devices to Edge Nodes	35
3.1	Motivation	35
3.2	FedAdapt Overview	37
3.3	FedAdapt Modules	41
3.3.1	Pre-processor	41
3.3.2	Clustering	42
3.3.3	RL agent	43
3.3.4	Post-processor	50
3.4	Evaluation	51
3.4.1	Acceleration of DNN partitioning-based Offloading in FL	51
3.4.2	RL Optimization for Heterogeneity	55
3.4.3	Adapting to Changing Network Bandwidth	59
3.4.4	Comparing FedAdapt and Classic FL	61
3.5	Existing Solutions	63
3.6	Summary	65

4	Reducing Communication between IoT Devices and Edge Nodes	67
4.1	Motivation	68
4.2	EcoFed Overview	70
4.3	Technical Design	72
4.3.1	Pre-trained Initialization	73
4.3.2	Replay Buffer	74
4.3.3	EcoFed Training Pipeline	76
4.4	Convergence and Cost Analysis	77
4.4.1	Convergence Analysis	77
4.4.2	Cost Analysis	80
4.5	Evaluation	81
4.5.1	Test Environment	81
4.5.2	Evaluation Setup	84
4.5.3	Accuracy	85
4.5.4	Communication Cost	94
4.5.5	Training Latency	97
4.6	Existing Solutions	100
4.7	Summary	101
5	Accelerating On-Device Training by Layer Freezing	103
5.1	Motivation	104
5.1.1	Learning Quickly or Learning Effectively	104
5.1.2	Opportunity for Applying Early-stage Layer Freezing	106
5.2	FedFreeze Overview	108
5.2.1	FL Steps and Layer Freezing Formulation	108
5.2.2	FedFreeze Architecture	110
5.2.3	FL Training Workflow with FedFreeze	113
5.3	Design of Layer Freezing in FedFreeze	113
5.3.1	Aggressive Regularization-based Layer Freezing	113
5.3.2	Conservative Convergence-based Layer Freezing	117
5.4	Evaluation	118
5.4.1	Evaluation Setup	118
5.4.2	End-to-end Performance	120
5.4.3	Performance Breakdown	124
5.4.4	Discussion	128
5.5	Existing Solutions	129
5.6	Summary	130

6	Integrating FedAdapt, EcoFed, and FedFreeze into EPFL	133
6.1	Motivation	134
6.2	Overview	135
6.2.1	System Architecture	135
6.2.2	EPFL Training Mode	136
6.3	Module Integration	138
6.3.1	Redesigning FedAdapt Modules for EPFL	138
6.3.2	Redesigning EcoFed Modules for EPFL	140
6.3.3	Redesigning FedFreeze Modules for EPFL	141
6.4	Evaluation	141
6.4.1	Evaluation Setup	142
6.4.2	Comparison of Classic FL, Vanilla DPFL, and EPFL	144
6.4.3	Performance Breakdown	146
6.5	Summary	151
7	Conclusion	155
7.1	Insights and Broader Impact	155
7.1.1	Insights	155
7.1.2	Broader Impact	157
7.2	Limitations and Future Work	159
7.2.1	Lack of Practical Understanding of the Limits to Scalability	159
7.2.2	Need For Improving Resilience	161
7.2.3	Limited Consideration of Asynchronous Learning	162
7.3	Summary	162
	References	165
	Appendix A Proof of Convergence for EcoFed (Chapter 4.4)	183
	Appendix B Experimental Trials and Significance of The Results	189

List of figures

2.1	Overview of the areas considered in the literature review.	16
2.2	CML pipeline.	17
2.3	An example of a hierarchical DML system built on devices, edge nodes, and the cloud.	23
2.4	Steps in each round of FL training: Step 1 - The server initializes the parameters of the global model and sends the parameters to each device, Step 2 - Each device completes local training on its dataset and sends updated local parameters back to the server, and Step 3 - The server aggregates local models to generate a new global model for the next round.	28
2.5	An illustration of centralized and decentralized FL.	29
2.6	A typical SL architecture in an edge environment.	30
3.1	FedAdapt framework and its positioning within FL.	39
3.2	Training of the RL agent used in FedAdapt	48
3.3	Comparing training time per iteration in FedAdapt and classic FL	54
3.4	Actions produced for each group chosen by the RL agent during training for VGG5. Action is the proportion of training workloads that remain on the devices within the group.	58
3.5	Total training time per round in seconds for different devices in FedAdapt and classic FL using VGG5.	59
3.6	Actions produced for each group chosen by the RL agent during training of VGG5 that accounts for devices with low network bandwidth to the server. Action is the proportion of training workloads that remain on the devices within the group.	60
3.7	Comparing the training time per round in FedAdapt and classic FL for VGG5. Vertical lines define five time slots after the 50 th round. At the beginning of each slot, the highlighted device is limited to under 10 Mbps bandwidth.	62

3.8	Test accuracy per round of FedAdapt and classic FL for VGG5.	62
3.9	Device and total training time per round in seconds in FedAdapt and classic FL for VGG8 when using the RL agent trained for VGG5.	63
3.10	Comparing the training time per round in FedAdapt and classic FL for VGG8 when using the RL agent trained for VGG5. Experimental setting is presented in Section 3.4.3.	64
4.1	Computation and communication time in DPFL training under typical (upload/download) network bandwidth. Numerical value above the bars is the percentage of communication time.	68
4.2	The training pipeline of classic FL, vanilla DPFL, local loss-based DPFL, and EcoFed for three rounds of training. Classic FL transfers the entire model from the devices to the server at the end of each round. Vanilla DPFL only needs to upload the device-side model at the end of each round. However, vanilla DPFL transfers the activation and gradient for each batch sample. Local loss-based DPFL reduces the communication by half since the gradient is computed locally. EcoFed reduces communication further as it transfers the activation only periodically (for example, once in two rounds) and further compresses the size of the activation.	70
4.3	EcoFed modules on the devices and server.	71
4.4	Test accuracy curves of EcoFed and the baselines using VGG11 and ResNet9 in I.I.D. and Non-I.I.D. settings for CIFAR-10 and CIFAR-100 datasets. . .	86
4.5	Accuracy for different ρ values with or without quantization in EcoFed. The results are an average of three independent runs with different random seeds.	90
4.6	Dynamic ρ values based on Table 4.10 in EcoFed. The results are an average of three independent runs with different random seeds.	91
4.7	The cumulative communication costs versus test accuracy of VGG11 and ResNet9 for the I.I.D and non-I.I.D. settings on CIFAR-10 and CIFAR-100 datasets.	96
4.8	Latency of one training round for VGG11 and ResNet9 under different network conditions for PP2. The results are an average of three independent runs.	97
4.9	Latency of one training round for VGG11 and ResNet9 under different PPs in 5G conditions. The results are an average of three independent runs. . . .	98
5.1	Accuracy of FL training using early-stage layer freezing approaches for the VGG11 model on CIFAR-10 dataset.	105

5.2	Test accuracy for different rounds and latency incurred for a target accuracy in FL training with accuracy-guaranteed layer freezing for VGG11 on CIFAR-10.	106
5.3	The SVCCA score of each layer during FL training for VGG11 on CIFAR-10. Layers are numbered bottom-up (i.e., from the initial to later layers).	107
5.4	The FedFreeze architecture.	110
5.5	Two prototypes are used to evaluate FedFreeze. A Raspberry Pi cluster with a laptop (edge server) and a Jetson Nano cluster with an Nvidia A6000 GPU server.	119
5.6	Test accuracy curves of Vanilla FL, ALF, AutoFreeze and FedFreeze for the LeNet, VGG11 and ResNet12 models on the FMNIST, CIFAR-10 and CIFAR-100 datasets with random and pre-trained initialization.	123
5.7	The breakdown of global freezing decisions when training LeNet, VGG11 and ResNet12 on the FMNIST, CIFAR-10 and CIFAR-100 datasets for both random and pre-trained initialization.	125
5.8	Local freezing decisions made by FedFreeze when training LeNet, VGG11 and ResNet12 on the FMNIST, CIFAR-10 and CIFAR-100 datasets for both random and pre-trained initialization. The results show the percentage of local frozen iterations for each layer out of the total iterations provided by the <i>Local Freezer</i> of FedFreeze. Layers that have no local freezing are omitted from the figures. The results are the average of using 10 devices for each round.	127
6.1	The EPFL system architecture.	136
6.2	The positioning and control flow of EPFL modules in a training round.	137
6.3	The training pipeline of EPFL in a given training round.	139
6.4	The data structure of buffer dictionary of <i>Replay Buffer</i> in EPFL.	141
6.5	Test accuracy curves of classic FL, vanilla DPFL and EPFL for the LeNet, VGG11 and ResNet12 models on the FMNIST, CIFAR-10 and CIFAR-100 datasets.	147
6.6	Accumulated latency versus test accuracy of LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.	148
6.7	Accumulated communication cost versus test accuracy of LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.	149
6.8	Overall latency of different methods for training LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.	152
6.9	Accumulated communication costs of different methods for training LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.	153

7.1 The computation versus communication latency when training VGG11 and ResNet9 models on 5,000 samples of CIFAR-10 using a Raspberry Pi 4 single-board computer using a Wi-Fi network. The numbers above each bar represent the percentage of communication latency. 156

List of tables

1.1	Computational resources (FLOPS and available memory) of end devices and cloud GPU [132]. This table is based on Table 2a which is presented by Pfeiffer et al. (2023) [132].	5
1.2	Resources required for training DNNs using PyTorch [127] on CIFAR-10 [79] with batch size 32 [132]. This table is based on Table 2b presented by Pfeiffer et al. (2023) [132].	6
2.1	Comparison of classic DML, FL, SL and DPFL. Classic DML refers to the traditional DML system that uses a distributed stochastic gradient descent [25] algorithm and operates in a cloud-only computing environment. In contrast, FL, SL, and DPFL adopt the training algorithms described in Section 2.4.1, Section 2.4.2, and Section 2.4.3, respectively. They are typically implemented across both cloud and edge computing environments.	33
3.1	Notation used in FedAdapt	44
3.2	Computational workload (on the device) and training time of FL, SL, SFL, and FedAdapt for one round	46
3.3	Architecture of the models used for evaluating FedAdapt. Convolution layers are denoted by C followed by the number of filters; filter size is 3×3 for all convolution layers, MaxPooling layer is MP, Fully Connected layer is FC with a given number of neurons, and the PP is followed with index.	53
3.4	Training time per iteration when layer offloading is used in FL for VGG5 under different network bandwidths.	53
3.5	Training time per iteration when layer offloading is used in FL for VGG8 under different network bandwidths.	54
3.6	Clustering devices into groups when using VGG5.	57
3.7	Training time per iteration in seconds for each device for all possible PPs in VGG5.	58

3.8	Example of clustering into groups for five devices with one low bandwidth device when using VGG5.	60
3.9	Comparison of FL, SL, SFL, and FedAdapt	65
4.1	Notation used in EcoFed	78
4.2	Computation and communication costs on the device for each round.	80
4.3	Models used for evaluation. Convolution layers denoted as C followed by the no. of filters; filter size is 3×3 for all convolution layers except for downsampling convolution (filter size is 1×1); MaxPooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers, a max pooling, and downsampling convolution layers; number followed is no. of output channels.	82
4.4	Different PP of VGG11 and ResNet9 used for evaluation. Convolution layers denoted as C followed by the no. of filters; filter size is 3×3 for all convolution layers except for the downsampling convolution where filter size is 1×1 ; Max Pooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers, a max pooling layer, and downsampling convolution layers; number followed is no. of output channels. Communication size is represented as <i>channels</i> \times <i>width</i> \times <i>height</i>	83
4.5	Typical network bandwidths available in the UK.	84
4.6	The test accuracy of EcoFed compared to the baselines for VGG11 and ResNet9 on two datasets on I.I.D. and Non-I.I.D. distribution. The results are an average of three independent runs with different random seeds.	87
4.7	The test accuracy of FedGKT (bi-direction) and FedGKT (uni-direction) on CIFAR-10. The results are an average of three independent runs with different random seeds.	88
4.8	The test accuracy of EcoFed compared to the baselines on CIFAR-10 with pre-trained initialization on the device-side model. The results are an average of three independent runs with different random seeds.	89
4.9	The test accuracy of device-side model of trainable \mathbf{w}_c and frozen \mathbf{w}_c^* in LGL (on CIFAR-10) and the highest test accuracy of respective server-side models under \mathbf{w}_c and \mathbf{w}_c^* . The results are an average of three independent runs with different random seeds.	90
4.10	Heuristic rules for dynamic ρ	91
4.11	Summary of pre-training datasets and training costs. The costs for pre-training the VGG11 model as pre-training time in seconds and the proportion relative to the total training time of EcoFed in brackets are reported.	92

4.12	The test accuracy of EcoFed on CIFAR-10 with different pre-training datasets. L and S are used to denote large-scale or small-scale datasets, respectively. Nat and Syn are used to denote whether the data is natural or synthetic. The results are an average of three independent runs with different random seeds.	93
4.13	Communication cost for one training round. It is worth noting that the communication cost is determined independently of different runs.	94
4.14	Communication costs incurred for different methods to achieve specific target accuracies.	95
4.15	The test accuracy of EcoFed on CIFAR-10 for different PPs using pre-trained ImageNet weights. The results are an average of three independent runs with different random seeds.	99
4.16	Comparing FL, vanilla DPFL, local loss-based DPFL, and EcoFed.	101
5.1	Notation used in FedFreeze	111
5.2	Evaluated models. Convolution layers denoted as C followed by the no. of filters; filter size of convolution layer is 5×5 for LeNet and 3×3 for VGG11 and ResNet12 except for downsampling convolution which is 1×1 ; Max Pooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers and a downsampling convolution layer; number following is no. of output channels. The batch normalization layer is applied after every convolutional layer in VGG11 and ResNet12.	119
5.3	Summary of evaluation. The highest accuracy achieved and total training time of FedFreeze and other baselines are reported. FedFreeze accelerates FL training by $1.07 \times$ to $1.30 \times$ while maintaining comparable highest accuracy.	121
5.4	Comparing different freezing techniques and FedFreeze.	131
6.1	Different PPs used for evaluating LeNet, VGG11, and ResNet12. Convolution layers denoted as C followed by the no. of filters; filter size is 3×3 for all convolution layers except for the downsampling convolution where filter size is 1×1 ; Max Pooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers, a max pooling layer, and downsampling convolution layers; number followed is the number of output channels.	143

- 6.2 Summary of evaluation. The highest accuracy achieved, total training time and total communication cost of FL, DPFL and EPFL are reported. EPFL accelerates training by $1.66\times$ to $5.57\times$ compared to FL and DPFL while maintaining a comparable accuracy. In addition, the communication cost of EPFL is up to $18.66\times$ lower than vanilla DPFL. 144
- 6.3 Evaluation performances when only employing FedAdapt. The highest accuracy achieved, total training time and total communication cost of FedAdapt and EPFL are reported. It is worth noting that FedAdapt is equivalent to the vanilla DPFL training on the Raspberry Pi testbed and equivalent to the classic FL training on the Jetson Nano testbed. 150

Glossary Terms

Automatic layer freezing (ALF)
Adaptive parameter freezing (APF)
Cloud computing (CC)
Canonical correlation analysis (CCA)
Centralized machine learning (CML)
Convolutional neural network (CNN)
Central processing unit (CPU)
Denoising diffusion probabilistic model (DDPM)
Distributed machine learning (DML)
Deep neural network (DNN)
DNN partitioning-based federated learning (DPFL)
Distributed stochastic gradient descent (DSGD)
Edge computing (EC)
Edge intelligence (EI)
Exponential moving average (EMA)
Efficient partitioning-based federated learning (EPFL)
Federated averaging (FedAvg)
Federated learning (FL)
Floating-point operations per second (FLOPS)
Floating point operations (FLOPs)
Graphics processing unit (GPU)
High-performance computing (HPC)
Infrastructure as a service (IaaS)
Independent and identically distributed (I.I.D.)
Industrial internet of things (IIoT)
Internet of things (IoT)
Local generated loss (LGL)
Multiply-accumulate operations (MAC)

Microcontroller unit (MCU)

Machine learning (ML)

Mobile neural network (MNN)

Non-independent and identically distributed (non-I.I.D.)

Platform as a service (PaaS)

Partitioning point (PP)

Proximal policy optimization (PPO)

Random-access memory (RAM)

Reinforcement learning (RL)

Software as a service (SaaS)

Single-board computers (SBC)

Splitfed learning (SFL)

Stochastic gradient descent (SGD)

Split learning (SL)

Singular vector canonical correlation analysis (SVCCA)

Tensor processing unit (TPU)

Chapter 1

Introduction

In recent decades, machine learning (ML) has achieved significant success across many domains [146, 73, 68]. Notable milestones include achieving human-level performance in image classification tasks [27], solving highly complex tasks in gaming and scientific discovery [157, 74], and developing promising solutions for language processing, such as ChatGPT [124]. Concurrently, interest in ML research and applications has surged within both the research and industry communities.

To further expand the usage of ML, research has explored building ML systems for various tasks, domains, and application scenarios, making ML ubiquitous in our daily lives [146, 33]. However, a single measurement of learning performance, such as accuracy in a classification task, cannot comprehensively evaluate the overall performance of an ML system across various applications. Other factors, such as system deployment requirements, must also be considered, as they can be crucial for specific ML systems. For instance, when building ML systems on resource-constrained devices, such as smartphones and sensors, factors such as execution latency, memory cost, and energy consumption are critical [138]. Another example is building an ML system that handles sensitive data, such as medical records, where privacy must be considered [183].

An application of such an ML system is federated learning (FL), which has gained popularity in recent years [190]. FL is a distributed machine learning (DML) paradigm for training a global model on the sensitive data of end-users using their own devices. The global model is an ML model constructed by aggregating models that are trained on multiple end-user devices using locally generated datasets. End-user devices typically reside on the network edge and have limited resources compared to those in a cloud data centre [6]. A natural question arises: why has FL become increasingly popular in recent years, and what advantages does it offer compared to traditional ML training paradigms? Section 1.1

addresses this question and motivates the central topic of this thesis, which is building a DML system that can be deployed at the edge.

1.1 Motivation

This section initially motivates the emergence of DML from traditional centralized machine learning (CML) on a single machine or computing node. It then discusses the transition of DML from the cloud to the edge, explaining the need to build a DML system at the edge.

1.1.1 The Emergence of DML

Advancements in ML in recent years can be attributed to three key factors: data, algorithms, and hardware. The availability of large datasets (e.g., ImageNet [27]) has enabled ML algorithms to achieve better performance by learning from a large corpus of training samples. Along with large datasets, more advanced ML algorithms, such as deep learning models (e.g., convolutional neural networks [92] and transformers [49, 41]), enhance the capabilities of ML models. Additionally, the upgrading of underlying hardware, such as central processing units (CPUs), graphics processing units (GPUs), and tensor processing units (TPUs), supports the efficient training processes of these advanced ML algorithms on large datasets.

Efficiency is a term often used ambiguously, particularly in discussions of system performance. In many cases, it is treated as a synonym for speed or overall throughput, when in fact, true efficiency refers to the optimal use of resources—such as energy, memory, or computational power—to achieve a desired outcome. Speed alone does not equate to efficiency; a system can be fast but wasteful in terms of energy or resource consumption. Therefore, it's essential to distinguish between raw performance and efficiency, which should be understood as performance per unit of resource. Clarifying this distinction allows for a more accurate evaluation of system design, especially in contexts like deep learning, where both speed and resource efficiency are critical considerations.

However, the rapidly increasing size of datasets and the complexity of models present significant challenges for the traditional training paradigm, which relies on single-machine or single-node ML training [32]. Specifically, building an ML system to train a complex model on large datasets using a single machine or computing node becomes impractical due to the substantial resources required [21]. For instance, training a GPT-3 model would take 3 years on a single V100 GPU device and require 700 GB of memory to store the parameters [22]. This results in impractically long training times or hardware requirements being insufficient on a single machine or node.

To address this issue, DML has been proposed to scale the size of training resources. In a typical DML system, multiple computing nodes are connected to support DML training, with each node executing individual training workloads in parallel. Additionally, these nodes are periodically synchronized to collaboratively contribute to the training of a global model. By distributing the training workloads across multiple nodes and enabling them to work in collaboration, DML systems can support ML training with large datasets and more complex models [170].

DML use cases. In general, DML can support the training of various types of ML algorithms, including supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning, as well as applications such as computer vision, natural language processing, and recommendation systems. It typically serves as an infrastructure system supporting training tasks for ML models. Some well-known DML platforms include Apache Spark [112], Horovod [149], and Ray [117], which are frequently used in the ML community.

1.1.2 Moving DML from the Cloud to the Edge

Early research on DML focused on building systems on multiple computing nodes within a cloud data center. Additionally, the datasets used for training were commonly stored in the cloud. These datasets may be generated within the cloud or uploaded from various sources, such as end-user devices. Cloud-based DML systems enhance ML performance by accelerating training times and reducing single-node memory consumption.

Two practical challenges have led to new requirements for DML systems. First, the widespread proliferation of Internet of Things (IoT) devices has significantly increased data generation on end-user devices [110]. This surge results in substantial communication costs for transferring distributed data to the cloud for DML training [111]. Second, privacy concerns have risen in relation to end-users transferring raw data generated on their devices to the cloud [175]. Additionally, regulations such as GDPR [23] mandate that certain sensitive user data must remain on the user's own devices.

The need to address the transmission costs of decentralized data and the privacy of end users has driven the development of new DML systems that reduce data communication and protect user privacy. In this context, several privacy-preserving DML paradigms at the edge have been proposed over the past decade [111, 113]. Among these paradigms, FL stands out as one of the most popular privacy-preserving DML paradigms. An FL system is typically built across both the cloud and the edge, ensuring that raw data remains in its original location, thereby mitigating privacy concerns. Consequently, FL requires orchestrating distributed workloads across both the cloud and the edge, unlike traditional cloud-only DML systems.

This fundamental difference in system architecture presents new challenges for FL compared to classic cloud-only DML systems.

1.2 Research Questions

This section discusses four challenges of building DML systems at the edge: limited resources and device heterogeneity, communication costs, on-device training costs, and the lack of a holistic solution. Based on these challenges, four key research questions are identified that will be addressed in this thesis. The research questions correspond to the objectives of Chapter 3, Chapter 4, Chapter 5, and Chapter 6, where the challenges will be further explored.

1.2.1 Limited and Heterogeneous Resources

One of the fundamental differences between building DML in the cloud and at the edge lies in the computational resources available. Modern cloud servers are equipped with high-end CPUs and GPUs, whereas computational capabilities on end devices or edge nodes are more constrained. Table 1.1 illustrates the varied computational capabilities of end devices at the edge compared to server GPUs in the cloud. Despite recent advancements in computational resources for end-user devices, such as the Jetson series (embedded GPUs) and high-end smartphones, a significant gap still exists between end-user devices and server GPUs. Contemporary ML models, such as deep neural networks (DNNs), are primarily designed for executing training on cloud infrastructure, leveraging server GPU acceleration. However, at the edge, the limited resources of end-user devices become a bottleneck, resulting in impractical training times for these computation-intensive ML models. In some cases, it is even infeasible to train these models due to hardware limitations, such as memory constraints [34].

Deploying DML at the edge presents the challenge of increased heterogeneity compared to the cloud environment. This is due to the variety of devices at the edge, rather than relying solely on GPU resources in the cloud. As shown in Table 1.1, the available floating-point operations per second (FLOPS) and random-access memory (RAM) of end-user devices can vary significantly, ranging from 10^5 to 10^{12} FLOPS, and from 0.5 MB to 100 GB RAM, respectively. This heterogeneity makes training more complex for DML at the edge. For example, the training time of DML can be significantly extended by the slowest devices, commonly referred to as the straggler in synchronized DML [119, 184].

Table 1.1 Computational resources (FLOPS and available memory) of end devices and cloud GPU [132]. This table is based on Table 2a which is presented by Pfeiffer et al. (2023) [132].

Device	FLOPS	RAM
Microcontroller Unit (MCU)	10^5-10^8	0.5–512 MB
Single-board Computers (SBC)	10^8-10^{10}	512 MB-8 GB
Low-End Smartphones	$10^{10}-10^{11}$	1-2 GB
Embedded GPUs	$10^{11}-10^{12}$	2-4 GB
High-End Smartphones	$10^{11}-10^{12}$	4-8 GB
Server GPUs	$10^{13}-10^{14}$	32-100 GB

Mitigating resource constraints and device heterogeneity when implementing DML at the edge raises the first research question of this thesis: **RQ1: How can techniques that surmount the challenges of limited resources and device heterogeneity be developed to effectively deploy DML at the edge?**

1.2.2 Communication Cost

Reducing communication costs is a significant research topic in classic DML within cloud environments [170, 82, 87]. The transfer of data between DML nodes is crucial to ensure that all nodes maintain consistent and up-to-date training data, typically including gradients generated by models. Efforts to reduce communication costs have been extensively explored in traditional cloud-only DML settings [82, 87, 35]. However, achieving efficient communication in DML at the edge is both more essential and challenging compared to DML in the cloud.

Unlike the high-bandwidth communication infrastructure in cloud environments, edge deployments often have constrained communication capabilities among end-user devices, such as mobile networks or Wi-Fi, rather than high-speed inter-node networks. Consequently, the overhead of communication becomes more critical when implementing DML at the edge. Furthermore, DML at the edge encompasses various system implementations, each incurring different communication overheads. For example, FL requires the regular exchange of model parameters. In contrast, another DML paradigm referred to as Split Learning (SL) [169] involves additional transfer of intermediate activations and gradients between devices and the server¹. For instance, the communication cost for one training round on a LeNet [81] model using the FMNIST [181] dataset for FL and SL are 0.93 GB and 52.95 GB, respectively.

¹For more information on the communication analysis of FL, SL, and other variants, please refer to Chapter 4 and Chapter 6.

Table 1.2 Resources required for training DNNs using PyTorch [127] on CIFAR-10 [79] with batch size 32 [132]. This table is based on Table 2b presented by Pfeiffer et al. (2023) [132].

DNN	# MAC	Memory
LeNet [81]	1.24×10^6	0.5 GB
ResNet18 [54]	1.12×10^9	0.8 GB
EfficientNet [164]	6.4×10^7	0.9 GB
MobileNetV2 [145]	1.92×10^8	1.4 GB
ResNet152 [54]	7.4×10^9	5.3 GB

To address this challenge, the second research question of this thesis is: **RQ2: How can we devise techniques to reduce the communication costs associated with deploying DML at the edge?**

1.2.3 On-device Training Cost

During DML training, computational workloads are distributed across nodes for parallel execution, including both end devices and the server. Training modern DNNs requires substantial computational demands, presenting a challenge for devices with limited resources to manage or complete within a reasonable time.

Table 1.2 presents the number of multiply-accumulate (MAC) operations required and the memory footprint when training different DNNs. It is important to note that the number of MAC operations only accounts for the requirements of one batch of data and does not consider other overheads, such as memory access costs. An illustrative example of training time on end devices is the training of the MobileNet model [145] using a Raspberry Pi 3 on the CIFAR-10 dataset. 8 hours and 41 minutes are required for just one training round of FL [39]. Therefore, the third research question posed in this thesis is: **RQ3: What techniques can reduce on-device training costs when deploying DML at the edge?**

1.2.4 Lacking A Holistic Framework

Lastly, the aforementioned three research questions target different, but complementary challenges in building DML at the edge. A natural question arises: Can the solutions to the individual research questions be meaningfully combined to enhance system performance? This underscores the need for a mechanism that integrates the solutions developed to address the three research questions. Therefore, the final research question of this thesis is: **RQ4: How can the techniques developed to address RQ1 to RQ3 be integrated to collectively address the challenges of deploying DML at the edge?**

1.3 Research Methods

This section briefly outlines four solutions proposed in this thesis to address the aforementioned research questions. These solutions are based on the FL training paradigm, a popular DML approach that integrates end devices, edge nodes, and servers [75, 119, 11]. The detailed description of the solutions will be presented in Chapter 3, Chapter 4, Chapter 5, and Chapter 6, respectively.

1.3.1 Adaptive Offloading to Address Resource Limitations and Device Heterogeneity

To address **RQ1**, this thesis proposes an adaptive DNN partitioning and offloading technique to tackle the challenges of limited resources available on and computational heterogeneity of end-user devices. Specifically, a portion of the DNN layers is offloaded from the device to the server for execution. The specific layer after which the entire model is offloaded to the server is referred to as the partitioning point (PP) in this thesis. For example, if a DNN has 10 layers and the first four layers are trained on the device and the remaining on an edge server, then Layer 4 is the partitioning point. This approach alleviates the computational burden on a device during training and utilizes the computational resources of a server, referred to in this thesis as DNN partitioning-based Federated Learning (DPFL) training. Additionally, using DNN partitioning offers a mechanism to customize the placement of partitioning points for different devices with varying computational capabilities. To address device heterogeneity, the PPs in the DNN are further optimized by using a reinforcement learning (RL) agent to adapt to the computational capabilities of different devices. Moreover, changes to network bandwidths are taken into account by the RL agent. This consideration is crucial because network bandwidths can significantly influence the latency performance of the DNN partitioning method². Consequently, the RL agent can adjust the PPs to accommodate heterogeneous devices and varying network bandwidth during the training process. The DNN partitioning and offloading technique, along with the RL optimization agent, are developed as a set of modules named FedAdapt.

FedAdapt has been evaluated on a lab-based testbed consisting of computationally-limited devices with diverse capabilities. Network bandwidths were manually adjusted during the training process to simulate network fluctuations. The results demonstrate that FedAdapt can reduce the training time of a typical IoT device by half compared to the classic FL approach through DNN partitioning and offloading. Specifically, the training

²Please refer to Chapter 3 for further details

time of extreme stragglers can be reduced by up to 57%. Moreover, when accounting for variable network bandwidth, FedAdapt has been shown to decrease training time by up to 40% compared to the classic FL approach while maintaining accuracy.

1.3.2 Pre-trained Initialization and Replay Buffer to Reduce Communication Costs

To address **RQ2**, this thesis proposes the use of pre-trained initialization and replay buffer techniques to specifically reduce communication costs in DPFL training. The FedAdapt module incorporates DNN partitioning and offloading techniques to alleviate the computational requirements on end-user devices. Simultaneously, it mitigates communication costs for classic FL by synchronizing only a subset of bottom-layer parameters between devices and the server.

However, significant communication costs persist because DPFL introduces new overhead for transferring intermediate activations and gradients between devices and the server. This cost has been shown to be even more substantial than the communication costs in classic FL (Chapter 4.1). This thesis introduces, for the first time, a communication-efficient paradigm specifically designed for DPFL systems. Specifically, a pre-trained initialization of the DNN model on the device has been developed to fundamentally eliminate the transmission of gradients while maintaining high accuracy. Additionally, a new replay buffer mechanism is proposed to further reduce the communication cost of activations. As a result, the communication costs of model parameters, activations, and gradients are all considered and mitigated in the proposed communication-efficient DPFL training paradigm. These techniques are integrated into a set of modules named EcoFed.

The proposed EcoFed modules have been evaluated on a lab-based testbed under various network bandwidth conditions to assess communication costs in terms of both size and latency. Experimental results demonstrate that EcoFed can reduce communication costs by up to $133\times$ and accelerate training by up to $21\times$ compared to classic FL. Moreover, compared to vanilla DPFL, EcoFed achieves a $16\times$ reduction in communication and a $2.86\times$ speed-up in training latency.

1.3.3 Early-Stage and Accuracy-Guaranteed Layer Freezing to Accelerate On-device Training

To address **RQ3**, this thesis introduces early-stage and accuracy-guaranteed layer freezing techniques to accelerate on-device training in FL. Layer freezing is employed to reduce

the intensive workloads of DNN training by eliminating the need for gradient computation, thereby decreasing overall on-device training costs. Specifically, the devices and the server use two different layer freezing techniques. An aggressive early-stage layer freezing technique, based on layer regularization, is proposed to speed up on-device training. Additionally, an accuracy-guaranteed layer freezing technique, based on convergence analysis, is adopted to ensure high final accuracy. By combining the features of both early-stage and accuracy-guaranteed layer freezing, the proposed techniques effectively reduce on-device training costs in FL with minimal compromise to accuracy. These two layer freezing techniques are integrated into a dual-phase layer freezing module named FedFreeze.

The proposed FedFreeze modules have been evaluated across various experimental testbeds, DNN models, datasets, and initialization methods. Experimental results indicate that FedFreeze achieves a training speedup of up to $1.3\times$ with an accuracy loss of no more than 1.67% compared to classic FL. Furthermore, compared to state-of-the-art early-stage and accuracy-guaranteed layer freezing approaches, FedFreeze demonstrates greater savings in on-device training costs while maintaining superior accuracy performance.

1.3.4 EPFL: A Holistic Framework to Integrate FedAdapt, EcoFed and FedFreeze

To address **RQ4**, an integration mechanism is proposed in Chapter 6, resulting in the development of a holistic framework named EPFL. This framework incorporates the modules of FedAdapt, EcoFed, and FedFreeze, embedding their core functionalities, such as adaptive DNN partitioning and offloading, pre-trained initialization, replay buffer, and dual-phase layer freezing. These functionalities interact with each other through the corresponding interfaces of each module throughout the FL training process. Specifically, EPFL deploys the adaptive DNN partitioning and offloading techniques during the deployment phase and implements the pre-trained initialization, replay buffer, and dual-phase layer freezing techniques during the runtime phase. Additionally, EPFL refines the design of the modules within FedAdapt, EcoFed, and FedFreeze to facilitate their seamless collaboration.

The proposed EPFL framework was evaluated across two testbeds, three datasets, and two models, demonstrating significant efficiency improvements over classic FL and vanilla DPFL. For example, on a Raspberry Pi testbed, FL training for a VGG11 [158] model on the CIFAR-10 [79] dataset takes 7,527.29 seconds of training time and 145.36 GB of communication. In contrast, EPFL completes the task in only 1,798.5 seconds and 38.26 GB, achieving a $4.19\times$ speedup in training time and a $3.8\times$ reduction in communication. Similarly, on a Jetson Nano testbed, vanilla DPFL training for ResNet12 [54] model on CIFAR-100 [79] requires

7,524.83 seconds and 610.71 GB of communication. In comparison, EPFL accomplishes this in just 1,350.26 seconds and 38.26 GB, representing a $5.57\times$ acceleration in training time and an $15.96\times$ reduction in communication. These improvements are achieved with an accuracy loss of up to 0.63%.

In this thesis, the term ‘efficiency’ has been used in the general sense as a state or quality rather than the technical sense of speed or performance per unit of resource. It has been used to describe the work presented in thesis where faster training and lower communication costs are incurred compared to baselines that have been selected to compare the work against. The thesis emphasizes optimizing model training speed and minimizing communication overhead associated with data, while minimizing loss of accuracy. Consequently, the training time or communication costs required to achieve the desired accuracy is minimized.

1.4 Research Contributions

The work presented in this thesis makes the following research contributions: :

- This thesis investigates the benefits of DNN partitioning and offloading from resource-limited IoT devices to the server within the context of FL. Building on this, an adaptive offloading technique utilizing clustering-based RL is proposed, designed to dynamically adjust the partitioning point by considering the heterogeneity of devices and varying operational conditions (e.g., network bandwidth).
- An in-depth analysis of the communication overhead in DPFL systems is presented. The study underscores the necessity of reducing communication costs in DPFL systems. Based on these conclusions, a communication-efficient DPFL training system is developed using the proposed communication reduction techniques, which include pre-trained initialization, a replay buffer, and quantization.
- The convergence behavior of DNN models during on-device training in FL is further examined, revealing a motivation to balance final accuracy with training speed. To address this, a dual-phase layer freezing technique is introduced. Specifically, an early-stage layer freezing technique is applied during local training to achieve acceleration, while an accuracy-guaranteed layer freezing technique is deployed on the server to maintain final accuracy.
- A holistic framework integrates the aforementioned techniques into a cohesive system. This framework jointly combines adaptive offloading, pre-trained initialization, replay buffer, and dual-phase layer freezing techniques to enhance the overall performance

of FL at the edge. Additionally, empirical evaluations with real IoT devices and edge nodes validate the proposed framework. The results demonstrate significant system improvements compared to classic FL and vanilla DPFL.

1.5 Publications

The core techniques presented in Chapters 3 and 4 have been published in peer-reviewed journals. A research article capturing the key techniques from Chapter 5 is currently under preparation for submission to a journal. Additionally, four patents based on the above articles have been filed and are currently under review. Furthermore, the aforementioned three articles also contribute to the publication of two collaborative articles in closely related fields. The articles and corresponding patents are listed below:

Research Articles.

As First Author:

- **[IOTJ 22]** **D. Wu**, R. Ullah, P. Harvey, P. Kilpatrick, I. Spence, B. Varghese, FedAdapt: Adaptive Offloading for IoT Devices in Federated Learning, *IEEE Internet of Things Journal*, vol. 9, no. 21, pp. 20889–20901, Nov. 2022.
- **[TPDS 24]** **D. Wu**, R. Ullah, P. Rodgers, P. Kilpatrick, I. Spence, B. Varghese, EcoFed: Efficient Communication for DNN Partitioning-based Federated Learning, *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 3, pp. 377-390, Mar. 2024.
- **[TBD]** **D. Wu**, L. Wong, B. Varghese, FedFreeze: Early-Stage and Accuracy-Guaranteed Layer Freezing in Federated Learning, Under preparation for future submission.

As Co-author:

- **[UCC 22]** G. Cleland, **D. Wu**, R. Ullah, B. Varghese, FedComm: Understanding Communication Protocols for Edge-Based Federated Learning, *IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 71-81, Dec. 2022.
- **[CM 22]** R. Ullah, **D. Wu**, P. Harvey, P. Kilpatrick, I. Spence, B. Varghese, FedFly: Toward Migration in Edge-Based Distributed Federated Learning, *IEEE Communications Magazine*, vol. 60, no. 11, pp. 42–48, Aug. 2022.

Patents Filed.**As Primary Inventor:**

- [US Patent] Adaptive Offloading of Federated Learning, **D. Wu**, R. Ullah, P. Harvey, P. Kilpatrick, I. Spence, B. Varghese, US20230016827A1.
- [US Patent] Collaborative Training with Compressed Transmissions, **D. Wu**, B. Varghese, P. Rodgers, R. Ullah, P. Kilpatrick, I. Spence, PCT/US2022/052496.
- [US Patent] Collaborative Training with Buffered Activations, **D. Wu**, B. Varghese, P. Rodgers, R. Ullah, P. Kilpatrick, I. Spence, PCT/US2022/053601.
- [US Patent] Accelerating Local Training and Achieving a Highly Accurate Global Model for Federated Learning (FL), **D. Wu**, L. Wong, B. Varghese.

As Co-Inventor:

- [US Patent] Cooperative Training Migration, R. Ullah, **D. Wu**, P. Harvey, P. Kilpatrick, I. Spence, B. Varghese, PCT/US2022/021587.

1.6 Thesis Outline

The remainder of the thesis is organized as follows:

Chapter 2 - Literature Review. This chapter presents the literature relevant to this thesis, providing the necessary background. Specifically, it discusses two key shifts: from CML to DML and from cloud computing (CC) to edge computing (EC). The natural convergence of DML and EC is then examined, followed by an introduction to three foundational DML training paradigms in the context of EC.

Chapter 3 - Offloading Workload from IoT Devices to Edge Nodes. This chapter presents the initial work in this thesis, focusing on the adaptive offloading of FL training workloads from IoT devices to the server. The chapter details the techniques used to achieve adaptive partitioning and offloading of DNN models in FL using RL techniques. This includes the implementation of DNN partitioning, the clustering of devices, and the development of an RL agent. Additionally, it describes how these techniques are assembled into the FedAdapt framework and discusses the performance improvements achieved through FedAdapt compared to traditional FL approaches.

Chapter 4 - Reducing Communication between IoT Devices and Edge Nodes. This chapter presents the second study in this thesis, focusing on mitigating communication overheads in a DPFL system. It introduces techniques such as pre-trained initialization,

replay buffer, and quantization to reduce communication overheads between IoT devices and the edge server, thereby enhancing the system's communication efficiency. These techniques are integrated into modules within the EcoFed framework. Detailed evaluations and comparisons with state-of-the-art baselines are provided to demonstrate that EcoFed can minimize communication costs, accelerate training latency, and maintain accuracy.

Chapter 5 - Accelerating On-Device Training by Layer Freezing. This chapter introduces the third study in this thesis, presenting a new approach to further reducing the on-device training workload by employing dual-phase layer freezing techniques on both devices and servers. The chapter details two components: regularization-based and convergence-based layer freezing. These two types of layer freezing techniques are then integrated into modules within the FedFreeze framework. The proposed FedFreeze technique is evaluated on two testbeds, across three training tasks, using two initialization methods to showcase the impact of FedFreeze on training speed and accuracy performance.

Chapter 6 - Integrating FedAdapt, EcoFed, and FedFreeze into EPFL. This chapter describes the integrated framework EPFL, which orchestrates the functionalities of FedAdapt, EcoFed, and FedFreeze. It outlines the system architecture of EPFL, the overall training workflow, and the redesigns of modules. The chapter also provides an evaluation of EPFL, including its end-to-end performance compared to classic FL and vanilla DPFL. Additionally, the breakdown of individual module performance is discussed to highlight the benefits of integration.

Chapter 7 - Conclusion. This chapter discusses the insights and broader impact of this thesis. Additionally, it presents the limitations of the current work and suggests directions for future research. Finally, it summarizes and concludes the thesis.

Chapter 2

Literature Review

This chapter provides a review of the literature relevant to the research considered in this thesis. Firstly, an overview of the areas considered in the literature is provided, as illustrated in Figure 2.1. Two shifts in the computing landscape relevant to building ML systems for real-world applications are noted: (i) transitioning from CML to DML, and (ii) moving from utilizing CC infrastructure to employing EC nodes. These highlight two crucial factors when building an ML system: whether the system is centralized or distributed, and whether the infrastructure resides in cloud data centers or spans across cloud and edge nodes. Section 2.1, and Section 2.2 consider these shifts, including the background, techniques and challenges. These two shifts naturally lead to the central topic of this thesis: the convergence of DML and EC. Specifically, Section 2.3 discusses the background, motivation, and challenges of this convergence. Finally, three prominent DML methods in EC are introduced that are foundational to the work of this thesis. In particular, Section 2.3 introduces three techniques, namely FL, SL, and DPFL.

The literature review in this chapter provides a high-level background for the thesis. Comparisons between existing research and the individual techniques proposed in this thesis are presented in the technical chapters (refer Section 3.5, Section 4.6, and Section 5.5).

2.1 From Centralized ML to Distributed ML

In this section, CML is first introduced along with its background and technical details, such as its pipeline and challenges. Then, DML is introduced with a discussion on the challenges of traditional CML. In addition, two types of DML parallelism are discussed, and the major challenge of communication overhead in DML is highlighted.

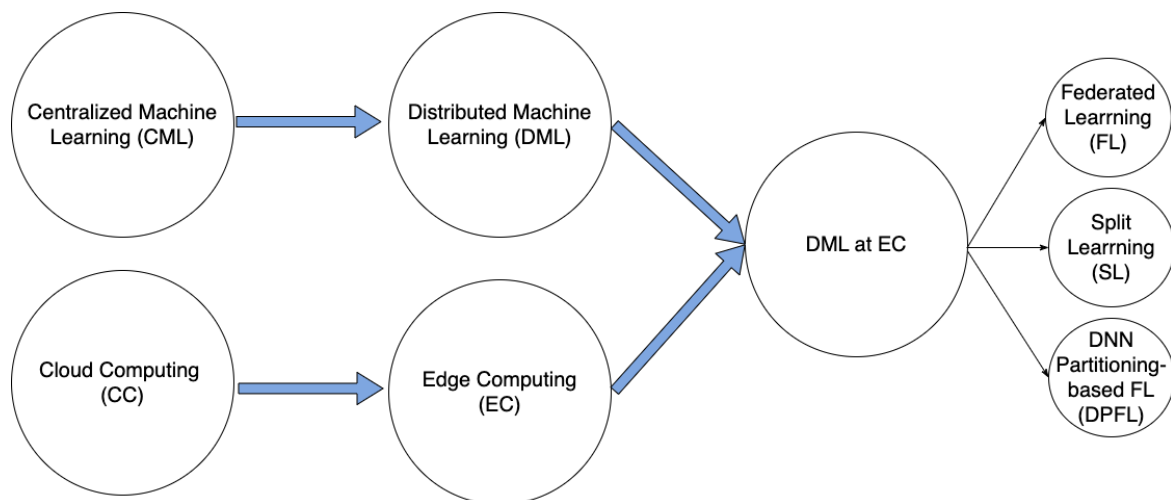


Fig. 2.1 Overview of the areas considered in the literature review.

2.1.1 Centralized ML

ML has achieved remarkable success in recent decades. For example, in the ImageNet competition [27], ML algorithms have achieved human-level performance in image classification tasks [54]. Another example is AlphaGo [157], which utilized RL to defeat the number one Go player in the world in 2017. More recently, OpenAI presented an AI software named ChatGPT [124] that leverages ML generative algorithms [14]. ChatGPT demonstrates human-level proficiency in certain natural language processing and coding tasks [124].

The success of ML can be attributed to three reasons. First, significant progress has been made in ML algorithms, particularly in developing deep learning algorithms for complex tasks [33, 155, 28]. Secondly, advancements in computational capabilities, such as the rapid upgrade of modern GPUs, has facilitated the training of these advanced ML algorithms. The third reason is the availability of large datasets, such as CCMatrix [148] or ImageNet [27], which have provided the data required for training ML models to a high accuracy.

CML pipeline: CML is a key training paradigm for many ML algorithms. Specifically, ML training is typically conducted on a centralized server located in a cloud data center, where data are stored and processed. In this training paradigm, the training data is pre-processed and stored on the cloud server. The preparation of data often requires human effort, including data collection, calibration, and labeling. In addition, a central server is responsible for aggregating, processing and, storing the data. The centralized collection of data offers advantages for data management, as all data is available in one location.

Figure 2.2 illustrates the detailed steps involved in a typical CML pipeline. The CML pipeline starts with data collection, where data is sent from devices to a central server. After the data collection, the server will execute the step of data pre-processing, including data

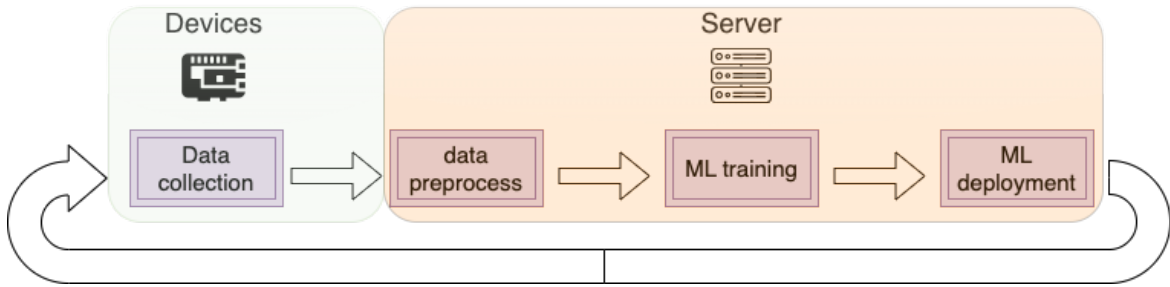


Fig. 2.2 CML pipeline.

cleaning, augmentation, and transformation to eliminate noise and inconsistencies in the data. Subsequently, the server will train ML models on the pre-processed data. These ML models are typically designed by ML engineers to achieve high learning performance, measured by metrics such as accuracy.

After completing the training of the ML model, the model needs to be deployed for providing ML services. There are two ways to deploy the ML services using the trained models. The first approach involves distributing the trained models back to the devices. However, this approach presents challenges because the training environment on the server differs from the deployment environment on the devices, potentially rendering the model unsuitable for deployment. Therefore, in the majority of CML paradigms, an alternative approach is to maintain the model on the cloud as a remote service. In this case, when a device requests the ML service, the cloud utilizes the trained model to compute and send the results back to the device. This deployment method is more practical since both the training and deployment of ML models occur on the cloud. Notably, the process mentioned above can be restarted. As new data is generated on devices over time, the CML steps need to be re-executed to ensure the model remains up-to-date.

The challenge of training large ML models on large datasets: To improve the performance of ML models, large-scale datasets comprising millions of data samples [27] are required. In addition, more advanced ML algorithms require larger models to be trained. These requirements can only be met by substantial computational resources. However, traditional single-machine training or single-node training cannot meet the requirements easily for training computationally-intensive ML models on large-scale data [170]. This poses an obstacle to traditional CML training of large ML models using large datasets.

2.1.2 Distributed ML

To address the above challenge, DML was proposed to distribute the training workloads across different computational nodes [186]. The DML training system is developed across

multiple computation nodes instead of a single machine. The advantage of DML lies in its ability to facilitate parallel training across multiple computation nodes, thereby enhancing the efficiency of training computation-intensive models on large-scale datasets. For example, Facebook [43] proposed a distributed synchronous Stochastic Gradient Descent (SGD) algorithm to complete ImageNet training in less than an hour utilizing 256 GPUs.

Data and model parallelism: In a DML system, two types of parallelism are commonly employed: data parallelism and model parallelism.

Data parallelism involves dividing the dataset across multiple computational nodes. Each node independently processes its assigned portion of the data [151, 88] for training. The parameters of a global ML model are replicated and distributed to each node. During the DML training with data parallelism, gradients of the parameters are computed locally for each batch of data on each node. These local gradients are then collected and averaged across all nodes before updating the parameters of the shared global model. The advantage of data parallelism lies in its efficiency in processing large datasets, as each node operates on a subset of the entire dataset concurrently. Data parallelism can accelerate the training process and enable the use of larger batch sizes for one global update step, leading to more stable and faster convergence of the training. By using data parallelism the same training performance as centralized training with a large batch size can be achieved.

Model parallelism, on the other hand, addresses the memory constraints of a single node when a large model cannot entirely fit into the memory of the node [156]. In model parallelism, different segments of the global model are allocated to separate nodes. During the DML training with model parallelism, the data flows within the computation of the model are distributed across nodes. Each node is responsible for processing specific layers or components of the global model. Model parallelism enables the training of large models that would otherwise be impractical due to memory limitations. By distributing different parts of the global model across nodes, it becomes feasible to effectively train larger models. By using model parallelism, the same training performance as in centralized training can be achieved, as only the execution location of different parts of the model changes.

The challenge of communication overhead in DML: Communication overhead in DML refers to the overhead for exchanging data, model parameters, and intermediate results between multiple nodes during DML training. Although DML accelerates training and reduces memory consumption on a single node, it incurs additional communication overhead between nodes. This communication is essential to synchronize training updates (e.g., activations and gradients) among multiple nodes. However, the transfer of data in DML systems often becomes a bottleneck due to the extensive volume of intermediate data that

is generated. As a result, a DML system needs to balance acceleration performance with communication costs.

2.1.3 Summary

Traditional CML is easy to implement because both the ML model and the data can be accessed on a single node. However, driven by the demand for training more computationally intensive models on larger datasets, DML has emerged as an effective solution to the challenge of insufficient training resources on a single node. This shift from single-machine CML to multi-node DML necessitates changes in system considerations when building DML, such as managing the communication overhead.

2.2 From Cloud Computing to Edge Computing

This section first discusses the concept of traditional CC and the ML services provided on the cloud. Then, EC is introduced with an additional emphasis on the benefits it provides.

2.2.1 Cloud Computing

Cloud computing refers to the computing paradigm where computational services are executed in remote cloud centers using cloud infrastructure and accessed over the Internet [109, 162, 143]. This approach enables users to access and utilize a vast array of computing resources including servers, storage, databases, networking, and software without the need for investments in physical hardware or infrastructure maintenance. Depending on the offerings provided by cloud service providers, service models can be categorized as: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [115]. In a IaaS model, users gain access to virtualized computing resources such as virtual machines, storage, and networking [115]. The PaaS model provides users with ready-to-use platforms and tools for building, deploying, and managing applications [115]. In addition, SaaS solutions deliver fully functional software applications over the internet, accessible via web browsers or APIs [115].

CC offers various services tailored to different phases of an ML pipeline. For example, it provides IaaS and PaaS for training ML models and SaaS for deploying the trained models [40, 129]. Specifically, during CML or DML training on the cloud, the cloud provides essential computational resources for model training. Users only need to consider the virtualized infrastructure and training APIs, without requiring in-depth expertise in underlying ML or DML training systems [144]. Moreover, during the deployment phase

with trained models, the cloud also offers mature solutions for deploying ML models as services to handle requests from devices [90]. Cloud-native ML services encompass a range of solutions, from fully managed platforms for developing and training to the deployment of trained models and APIs for inference. With the help of elastic compute resources [130], cloud-native ML services enable users to efficiently train and deploy ML models on datasets of varying sizes and complexities, accelerating the development of ML applications.

2.2.2 Edge Computing

It is estimated that by 2025, more than 55 billion IoT devices will be connected to the internet generating vast amounts of data at the network edge [65]. EC has emerged as a new computing paradigm, aiming to bring computational resources and data storage closer to where they are generated, rather than relying solely on traditional cloud data centers [168, 167, 154]. In the traditional CC paradigm, data is sent from IoT devices to remote cloud data centers for processing, analysis, and storage. However, sending all data from devices to the cloud can lead to network congestion and raise privacy concerns, particularly when dealing with large volumes of data containing sensitive user information. EC addresses these challenges by moving computing resources closer to the data source. This approach enables faster response times, reduces data communication requirements, and enhances privacy.

Implementing ML in EC environments offers two benefits: first, it can enhance system performance in terms of both computation and communication. Second, EC can address concerns regarding data privacy by keeping data processing closer to the devices.

EC for improving system performance. The EC paradigm serves as an intermediate computation location between IoT devices and the cloud. This new computing paradigm can be utilized to enhance the performance of an ML system in two ways. First, it enables the migration of ML services from the cloud to edge nodes or even directly to end devices, thereby reducing response time and communication overhead. Second, it facilitates the transfer of the ML workload, including both training and inference tasks, from resource-limited IoT devices to edge nodes that are equipped with relatively more computational resources.

In the traditional CML paradigm, ML services are typically hosted in the cloud. However, when millions of IoT devices simultaneously require these services or when there are high network delays, the response times of the services become impractically slow. To overcome this limitation, EC enables the deployment of ML services on edge nodes, bringing the ML services closer to the devices. As a result, devices can access nearby ML services with lower network delay, which is essential for time-critical applications [97, 59]. Additionally, the

shift in deployment location reduces the volume of data transmitted to centralized servers, thereby alleviating communication burdens on the network.

ML computations can also be offloaded to edge nodes, thereby providing support to resource-constrained devices. Deploying ML services on resource-constrained devices is challenging due to the heavy computational workload required by ML applications. To address this issue, EC emerges as a valuable solution by offering edge nodes as offloading locations for deploying ML services on resource-constrained devices [102, 93, 69]. By leveraging the computation resources on the edge, the burden of ML computations on resource-constrained devices is reduced, thereby mitigating the issue of heavy computation requirements. Offloading ML computations from devices to edge nodes enhances the overall performance of the ML system, especially in scenarios where IoT devices cannot afford the computation cost of ML applications.

The motivation for running ML applications on millions of IoT devices stems from the need for real-time processing and scalability in several critical applications. For instance, in smart cities, IoT devices, such as traffic cameras and environmental sensors require local ML computation to provide timely insights for managing traffic flow or monitoring pollution [126]. Industrial IoT systems rely on edge-based ML for predictive maintenance of equipment, where low-latency decision-making is critical [30]. Similarly, autonomous vehicles and drones require on-device object detection in real-time to ensure safe navigation and maintain necessary safety thresholds [91].

The examples chosen in this thesis are based on image classification, a fundamental task in computer vision to support object detection and segmentation. In addition, by using image classification as a demonstrator, this thesis follows a standard approach commonly seen in the literature, specifically those at the intersection of distributed systems and machine learning paradigms, such as FL [75]. The experiments conducted in this thesis offer a benchmark for comparing the proposed techniques with existing works. However, it is worth noting that the approaches proposed in this thesis are not necessarily limited to image classification. The underlying principles can be applied to other tasks, such as object detection, segmentation, and beyond.

EC for resolving privacy concerns. Another benefit of EC is its ability to enhance the privacy of end-users. In traditional ML applications, both the training and deployment of an ML model require the processing of raw data from end-users. In a typical CML system, this raw data must be transferred from the devices to the cloud, which raises privacy concerns. Users are sensitive about how their data will be utilized and stored, particularly in light of the growing emphasis on privacy protection by both end-users and governments [23]. EC addresses these concerns by eliminating the need to transfer raw data to the remote cloud.

Instead, it brings the ML service or application to nearby edge nodes or even local end-user devices. In EC, data is processed closer to its source, meaning sensitive information passes through fewer third-party nodes over the network, which may alleviate some concerns around data privacy. Additionally, since EC distributes the processing across multiple devices and nodes, the risk of large-scale breaches can be potentially reduced compared to centralized (cloud) systems, often subject to single points of failure. As a result, EC has potential benefits for information-sensitive ML applications that makes it attractive as a paradigm for building modern distributed systems.

2.2.3 Summary

CC provides a range of solutions for training and deploying ML services in the cloud. However, managing ML applications with millions of IoT devices poses significant challenges for CC. To address these scalability challenges, EC helps by executing computations on end devices and edge nodes without sending them to the cloud, thereby enhancing the performance of ML systems. Additionally, it alleviates users' privacy concerns regarding their data. The shift from CC to EC is natural due to the increasing number of IoT devices and growing attention to data privacy.

2.3 The convergence of DML and EC

This section discusses the convergence of DML and EC, focusing on the implementation of DML systems at the edge. In particular, we introduce a hierarchical DML system consisting of two fundamental phases: DML training and DML inference. Additionally, we delve into the primary challenges encountered when applying DML in EC environments and explore existing techniques aimed at addressing these challenges.

Benefits from combining DML and EC. In recent years, the convergence of DML and EC, also known as edge intelligence (EI) [29, 198, 182], has gained popularity. The convergence of DML and EC is inherently beneficial in two ways. (i) DML brings advanced intelligence to EC, enabling smart AI applications on the edge, such as autonomous driving [97], smart home [15], and the industrial Internet of Things (IIoT) [86]. (ii) EC enhances the quality of DML applications by offering richer data and diverse application scenarios. The massive IoT data generated at the edge provides opportunities to enhance the performance of ML models.

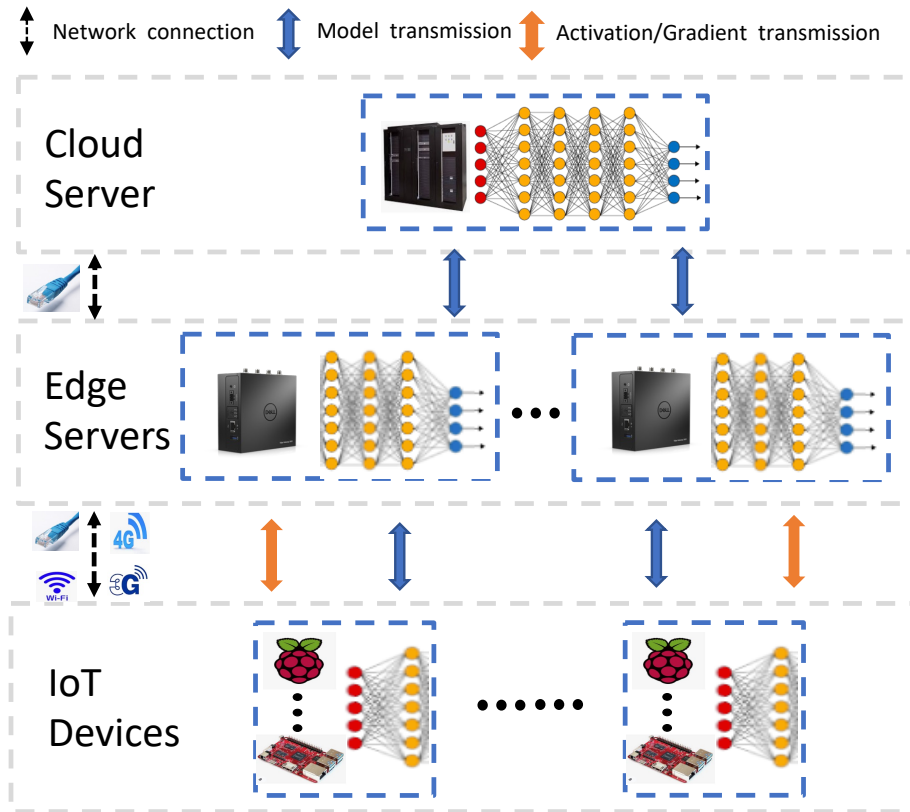


Fig. 2.3 An example of a hierarchical DML system built on devices, edge nodes, and the cloud.

2.3.1 DML at the Edge

Traditional DML systems are built on the cloud with homogeneous computation resources. For example, high-performance computing (HPC) clusters on the cloud are built with multiple homogeneous devices (e.g., CPUs and GPUs) connected by high-bandwidth networks [170, 5]. However, building a DML system with end devices and at the edge is more challenging [84, 198].

Figure 2.3 shows an example of a hierarchical DML system built on end devices, edge nodes, and the cloud. A portion of the ML workload is deployed on the end devices, while the remaining workload is distributed across edge nodes and the cloud. In this DML system across devices, edge nodes, and the cloud, ML workloads (e.g., DNN training) are partitioned into devices, the edge and the cloud. Devices, edge nodes, and servers collaborate to complete a DML task with the transfer of model parameters, activations, and gradients between the three tiers. Depending on the different phases of the ML pipeline, existing research on DML systems can be divided into two categories: those built for training and those built for inference [135].

DML inference at the edge. Building a DML inference system has been previously investigated by researchers and the industry community, as it primarily involves optimizing the forward pass of an ML model. For example, to address the challenge of limited computational resources on end devices, researchers have developed a device-edge co-inference framework. This framework offers a solution by partitioning the DNN into segments executed on the device side and edge side [84, 100]. To further reduce communication overheads, as intermediate activations need to be transferred to the edge server, research has focused on compressing these activations by incorporating specific neural network architectures that downsample the activations before transferring them to the server [36, 152]. However, these works consider only the inference phase and assume that the model will be trained on the cloud in advance.

DML training at the edge. In contrast to DML inference, orchestrating DML training at the edge has been less investigated. This is primarily because training necessitates more computational and communication resources and typically requires longer execution times. FedAvg [111] was the first to propose the FL paradigm, which involves independently training ML models on each end device and aggregating the global model in the cloud. Building on the FL paradigm, further research has incorporated edge nodes to reduce the computational workload of classic FL [53]. Additionally, a hierarchical FL architecture has been proposed to optimize communication costs between end devices, edge nodes, and the cloud [96]. However, research on orchestrating DML training at the edge has not been widely adopted in the industry. Current studies have yet to comprehensively address the challenges of implementing DML in edge computing, such as limited resources, device heterogeneity, and communication overhead.

2.3.2 Challenges of Implementing DML in EC

There are several challenges when implementing DML in EC.

Limited resources. The first challenge encountered when implementing DML in EC is the limited resources of EC systems compared to the cloud-only environment. Specifically, the computational capabilities of IoT devices and edge nodes are limited in comparison to cloud GPUs. The limited resources of EC systems pose a challenge when running computation-intensive ML workloads, such as DNN training and inference [34]. This is because the development of DNN algorithms typically assumes that training and inference occur on the cloud rather than on edge nodes or IoT devices. Therefore, techniques aimed at reducing the computational burden of these DNN algorithms are necessary to overcome the challenge posed by limited resources [138]. This challenge relates to RQ1 and RQ3

introduced in Chapter 1, and the corresponding solutions are discussed in Chapter 3 and Chapter 5.

System heterogeneity. Another challenge encountered when implementing a DML system in EC environments is system heterogeneity [62]. Unlike traditional cloud environments where hardware and software configurations are relatively uniform, EC systems often consist of heterogeneous devices with varying computational capabilities, storage capacities, network bandwidths, and power constraints. This heterogeneity introduces additional complexities in designing and deploying DML systems, as the DML algorithm needs to be adapted and optimized across diverse devices in order to achieve optimal performance [61]. Moreover, orchestrating DML training across heterogeneous edge nodes further complicates the development of DML systems in EC environments [159]. This challenge relates to the device heterogeneity of RQ1 introduced in Chapter 1, and the corresponding solutions are discussed in Chapter 3.

Communication overhead. The communication overhead is the third challenge when implementing DML in EC. This cannot be ignored because of two reasons. (i) The communication capacity of end devices and edge nodes is also limited compared to cloud nodes [108]. (ii) In some DML paradigms, the communication overhead is typically the bottleneck of the system [78, 178]. Therefore, a detailed analysis of communication cost and effective techniques for reducing the communication cost are required when implementing DML in EC [78, 16]. The challenge of communication overhead relates to RQ2 introduced in Chapter 1, and the corresponding solutions proposed by this thesis are discussed in Chapter 4.

2.3.3 Existing Solutions

To solve the above challenges when developing DML training and inference in EC, several techniques have been proposed such as compression algorithms and framework optimization. These techniques aim to reduce the total training workload (i.e., computation and communication costs) of DNNs on resource-constrained devices and edge nodes. Examples of model compression techniques include model pruning [99], quantization [51], reduced synchronization, and DNN framework optimization [70].

Model pruning. Model pruning is a DNN model compression technique to reduce the size and computations of DNNs without compromising performance significantly. In particular, model pruning eliminates unimportant connections (weights) or entire neurons from the DNN, effectively compressing the size of the DNN and reducing computations [199, 101]. Pruning methods can be categorized into two approaches: unstructured pruning and structured pruning. Unstructured pruning, also known as weight pruning, involves removing individual weights [38]. Structured pruning, on the other hand, aims to remove entire filters, channels,

or even layers, resulting in more structured sparsity within the model. Model pruning not only reduces the model size and memory footprint but also accelerates inference and training. As a result, pruning can lower the requirements for deploying DML on resource-constrained devices [94].

Quantization. Quantization is another model compression technique to reduce the precision of numerical values in a model [134]. The parameters of DNNs are typically represented as 32-bit floating-point numbers both in the training and inference phases. In quantization, the parameters are converted from high-precision 32-bit floating-point numbers into lower-precision fixed-point numbers or integers. As a result, it reduces the memory footprint and computational workloads of the model, making DML deployment on edge nodes and IoT devices efficient. One popular category of quantization is post-training quantization [176], where a trained model is transformed into a quantized model without re-training. This technique has been widely adopted due to its simplicity of implementation, as it does not require re-training with the original dataset. In a typical 8-bit post-training quantization, weights and activations are quantized to 8 bits using various quantization schemes such as uniform quantization [176], non-uniform quantization [7], or adaptive quantization [197]. In addition, the quantization-aware training technique can be further adopted to reduce the accuracy degradation caused by post-training quantization [37]. However, it requires adopting quantization during the training phase to get the final model.

Reduced synchronization frequency. To reduce the communication overhead, an effective approach is to decrease the frequency of communication in the DML system. Communication often occurs when the DML system synchronizes information between nodes. Therefore, reducing the synchronization frequency is an efficient way to mitigate the overall communication cost. For instance, in a typical DML training system, once the global model is downloaded by multiple devices, each device starts to train locally. As each device computes and updates the model, these updates are communicated back to a central node for synchronization. The number of communication rounds between the devices and the central node can be reduced through more efficient model updates [150]. However, reducing synchronization frequency may slow convergence during training.

Optimizing DNN frameworks. Another technique to accelerate the training or inference of DNN models on resource-constrained devices is to develop an optimized training or inference framework that considers the hardware characteristics. For instance, Mobile Neural Network (MNN), a highly efficient and lightweight DNN inference framework, can reduce inference time on diverse mobile devices by approximately 30% [70]. However, current research in this avenue mostly focuses on optimizations on the inference phase for a single device.

2.4 DML Training Paradigms in EC

This section introduces three popular DML training paradigms in EC, which are the foundational blocks upon which this thesis is built. These include FL, SL, and DPFL.

2.4.1 Federated Learning

FL is a privacy-preserving DML technique that has recently gained popularity [185, 75, 12, 190, 52]. Using this DML paradigm, a global ML model (such as a DNN) is independently executed on several end devices. The model on each device is trained without sending raw data (which may be sensitive) from the device to a server. Instead, the server is sent intermediate models generated by the devices, which are aggregated on the server to create a global model. Thus, an ML model can be trained without exposing sensitive data from a device to an external server. For instance, a successful application of FL is that Google has applied FL on Android Gboard, the Google Keyboard, to train a next-word prediction model using the daily typing history on user mobile devices instead of collecting the data on a cloud server [186].

Typically, a round of FL comprises three key steps, as shown in Figure 2.4. In the first step, a global model, W , is initialized on the server and distributed to all devices. In the second step, each device independently trains the ML model using data generated by the device. Typically, one epoch of local training utilizes the entire dataset from each device. After independent training, the models trained on the local devices (updated model parameters, W_k) are sent to the server. In the third step, a new global model is aggregated as follows [111]:

$$W = \sum_{k=1}^N \frac{n_k}{n} W_k \quad (2.1)$$

where N denotes the number of devices, n_k is the number of data samples on the device k and $n = \sum_{k=1}^N n_k$ is the number of all data samples across all devices. In subsequent rounds of FL training, the aggregated model is distributed to all devices, and the above steps are repeated until the training loss converges or a time limit is exceeded.

FL is efficient for scaling with a large number of devices since local training can be carried out in parallel. However, FL is known to be less efficient for heterogeneous devices that have different computational capabilities, as local training in the straggler will become a bottleneck [39, 171, 177, 184]. In addition, the computationally intensive workload (training of W_k) is solely executed on the device. An external server, located at the edge node of the network or in the cloud, only aggregates the weights collected from the end devices, which is relatively less computationally expensive. The majority of the computational workload

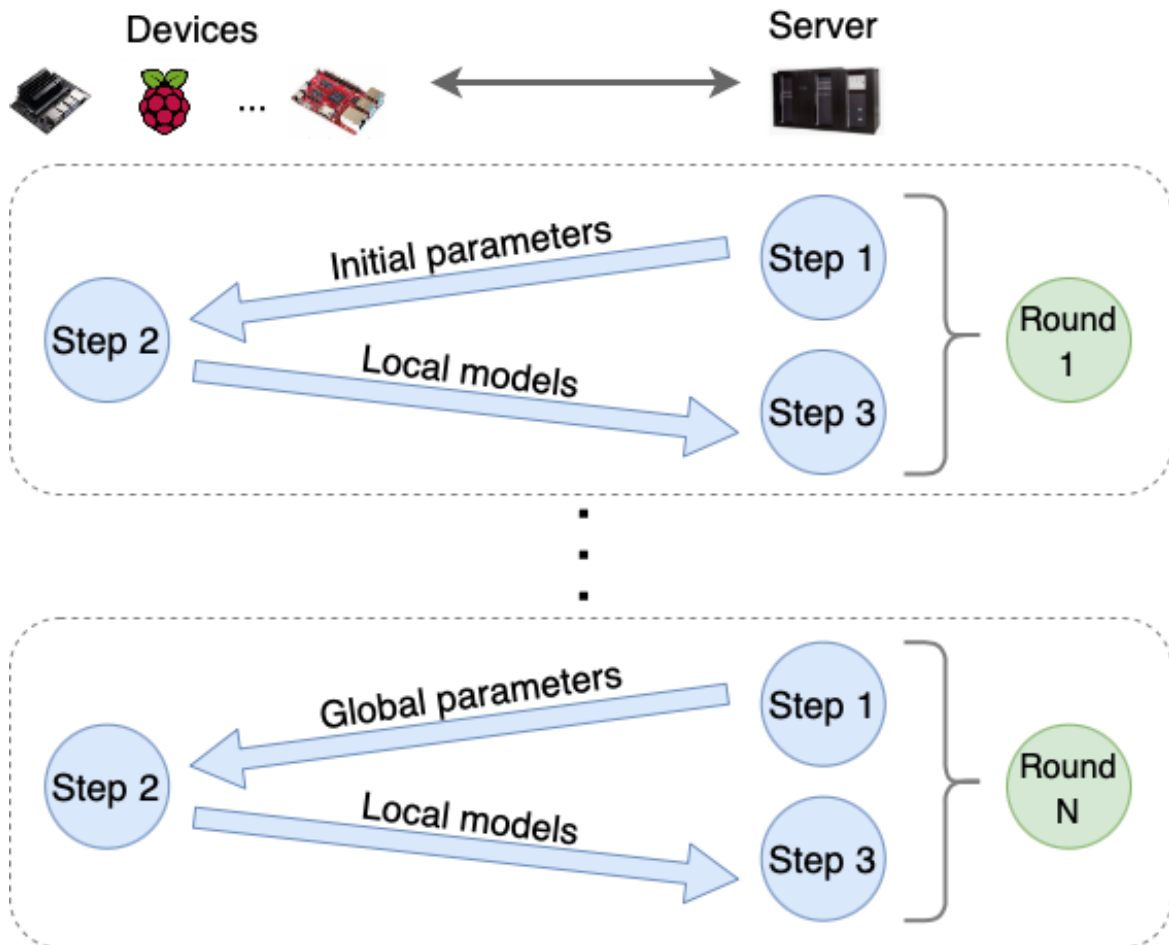


Fig. 2.4 Steps in each round of FL training: Step 1 - The server initializes the parameters of the global model and sends the parameters to each device, Step 2 - Each device completes local training on its dataset and sends updated local parameters back to the server, and Step 3 - The server aggregates local models to generate a new global model for the next round.

(local training) is allocated to the devices, which often have limited resources and energy budgets [66]. In addition, each device simultaneously communicates with the server to exchange updated weights through diverse network configurations, which further exacerbates the burden. In general, the typical FL system is effortless to scale out due to the high degree of parallelism, which is efficient when dealing with a large number of devices. However, it also has a drawback that puts most of the computation workload individually on each device, which has less computing capability than the server.

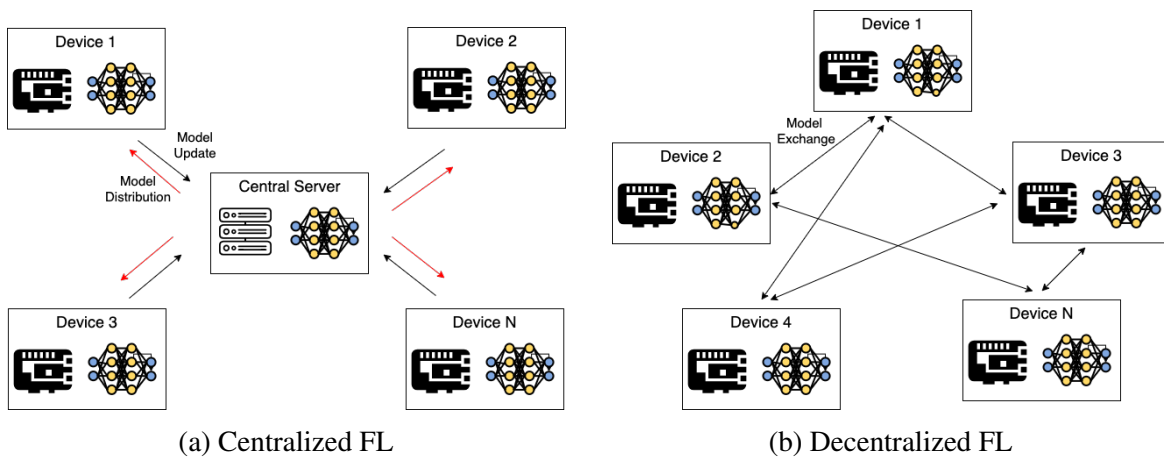


Fig. 2.5 An illustration of centralized and decentralized FL.

Centralized FL and Decentralized FL. FL can be categorized as centralized FL and decentralized FL based on the network topology: These approaches share the common goal of collaboratively training an ML model while keeping data generated on or available on the participating devices private. However, they differ in how devices are connected to one another and how communication occurs during training [161, 10]. As illustrated in Figure 2.5, centralized FL coordinates model updates via a central server, whereas decentralized FL relies on direct communication between devices.

In centralized FL, a central server is the orchestrator for training the global model (as shown in Figure 2.5a). Each participating device trains the model locally on its data, then sends model updates to the central server. The server aggregates these updates and redistributes the global model to the devices for further training. The network is a star topology in which all devices communicate with the central server but not with each other. Centralized FL methods are useful in applications, such as Google’s Gboard for predicting the next-word during typing on an electronic keyboard [186]. User data remains local, and the global server in the cloud facilitates FL training. Additionally, centralized FL is the most common architecture in current FL research [10], making it the primary focus of this thesis.

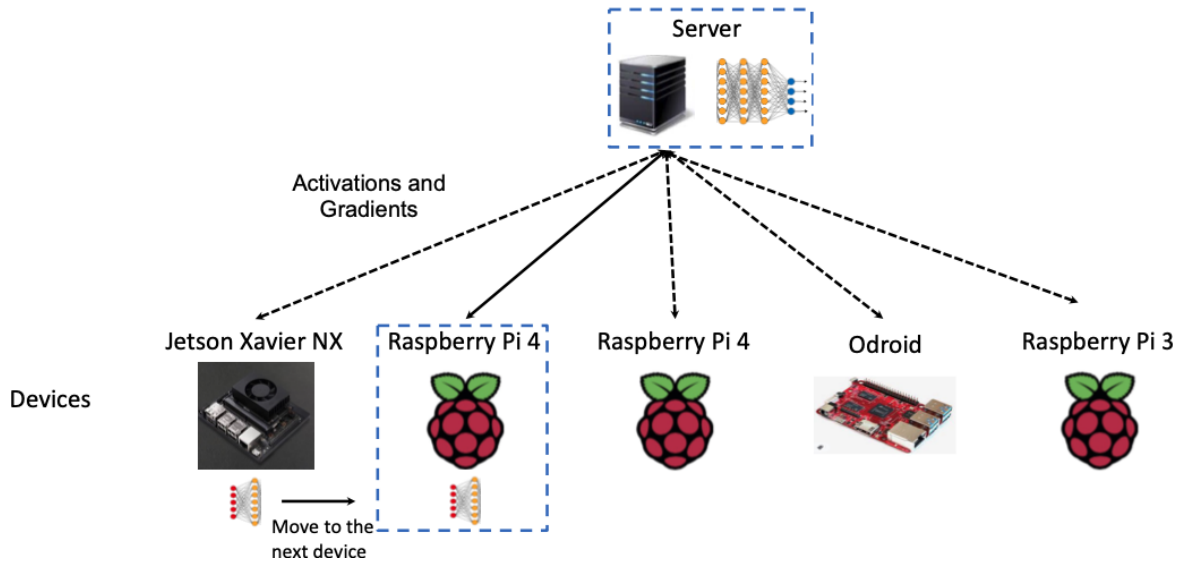


Fig. 2.6 A typical SL architecture in an edge environment.

Decentralized FL, in contrast, eliminates the need for a central server (as shown in Figure 2.5b). Devices communicate directly with each other in a peer-to-peer fashion, sharing and updating model parameters collectively. The network topology resembles a mesh, where devices interact with multiple peers. This paradigm is beneficial in scenarios where relying on a single server is not desirable, such as in smart grid networks and autonomous vehicle networks [10, 141].

2.4.2 Split Learning

SL is another popular privacy-preserving DML paradigm proposed by MIT in 2018 [113]. It was first used to collaboratively train a global model across the data from several hospitals, which is quite sensitive and cannot be shared with each other [169, 133]. Recent research also applies SL on end devices to formulate an efficient DML paradigm in EC [18].

A typical architecture of SL is shown in Figure 2.6 with a global server and 5 end devices. At the beginning of the training, a global model (monolithic DNN) is partitioned into two networks, namely a device-side model and a server-side model. On the device, the DNN is trained up to the layer at which the DNN is partitioned. Then the activation feature map (Activation) of the last layer on the device is sent to the server. The server continues training until the last layer of the DNN. After the training loss is calculated and the gradient is updated, the respective gradients are sent to the device so that the gradients on the device can be calculated and updated. When training in SL with multiple devices, as shown in Figure 2.6, the devices are trained in a sequential round-robin fashion, whereby only one device will

be connected to the server at a time. After a given device completes training, the updated weights are copied onto the next device to continue training.

SL could protect the privacy of data by sending only activations to the server to train the server-side model, leaving the training of the device-side model on the local devices. This avoids the need to transmit raw data from end devices to the server. An advantage of SL is that collaborative training with the help of the server could significantly reduce the computation workloads for the devices compared to FL, in which the entire DNN is trained on the device. Considering the limited computation resources on end devices and the impractical training time of classic FL training on end devices [39], partitioning the model could move the majority of the training workload (most of the DNN layers) to the server. However, when there are a large number of devices, it is inefficient for SL to scale out since all participants need to sequentially connect with the central server and conduct training.

2.4.3 DNN Partitioning-Based FL

There is another line of research that combines FL with SL in the EC setting, namely DPFL. The existing research on DPFL can be divided into two main categories: vanilla DPFL and local loss-based DPFL. SFL [165] is the first work in DPFL, which takes a unique approach by partitioning a DNN between the device and the server in FL. This method combines FL and SL, where device-side models are trained independently by receiving gradients from the server, while server-side models are trained by collecting activations from devices in parallel. By doing so, SFL addresses the computational limitations of resource-constrained devices and accelerates the training process. However, it's worth noting that while SFL efficiently handles the computational burden, it does not account for the additional communication overhead introduced by the DNN partitioning.

Recently, improvements have been made in DPFL approaches, with a focus on optimizing communication costs. These advancements involve computing the local loss on the device, which helps reduce the need for extensive communication [47, 53]. In these methods, the device-side model is trained using local error signals generated by an additional auxiliary network, eliminating the requirement to transfer gradients from the server and utilize them on the device. However, this approach of training the device-side model with local error signals has its limitations, resulting in suboptimal performance and reduced accuracy. Additionally, these local loss-based DPFL methods often overlook the communication costs associated with transferring activations.

A drawback of DPFL methods lies in the vulnerability of activations to hacking, particularly concerning model inversion attacks that can potentially reconstruct the original data [103]. To address this security concern, one potential solution is to incorporate homomor-

phic encryption. By leveraging homomorphic encryption, the activations of the device-side model can be encoded, ensuring secure communication within the DPFL framework [20].

2.4.4 Discussion

Comparison between classic DML, FL, SL and DPFL. Different DML paradigms have unique characteristics influenced by the training algorithms they utilize and the computing environments in which they usually operate in. Table 2.1 presents a comparison of various DML paradigms, including classic DML, FL, SL, and DPFL. Specifically, classic DML refers to the traditional DML system that uses a distributed stochastic gradient descent algorithm [25] and is usually deployed in a cloud-only computing environment. In contrast, FL, SL, and DPFL adopt the training algorithms described in previous sections and are typically implemented across both the cloud and the edge.

As shown in Table 2.1, classic DML, FL, and DPFL can accelerate training, as training on distributed data is carried out simultaneously on each computation node. This process is referred to as data parallelism. However, SL trains the model sequentially, leading to inefficient training, as the number of end devices increases. FL, SL, and DPFL train ML models without the need to transfer the data of end users to servers. In contrast, classic DML in the cloud requires collecting raw data from devices and sent to the cloud. Device heterogeneity in classic DML is low in the cloud, as typically homogeneous devices are deployed when building the cloud infrastructure. However, device heterogeneity in FL, SL, and DPFL at the edge is high, as different end users may possess various types of end devices.

Classic DML in the cloud requires only pre-processing computations on each device before sending the raw data to the server. SL and DPFL reduce computation on devices by offloading the DNN training workload from the end devices to the cloud server. However, FL encounters a bottleneck in on-device computation due to limited resources at the edge. Regarding device-side communication costs, SL, and DPFL have significant communication overhead, due to the need to send forward activations and backward gradients. In contrast, classic DML in the cloud and FL are communication-efficient, since they only need to communicate raw data once or updated weights at the beginning and end of each round, respectively.

To summarize, classic DML executes training on the cloud without considering data privacy. FL requires substantial training on devices with limited resources. SL and DPFL incur significant communication overhead due to the need to send forward activations and backward gradients.

Research gaps. Although FL, SL, and DPFL are emerging paradigms for training ML models locally at the edge, three research gaps are identified as follows:

	Classic DML	FL	SL	DPFL
Execution Environment	Cloud	Cloud and edge	Cloud and edge	Cloud and edge
Training speed up	✓	✓	✗	✓
Data privacy	✗	✓	✓	✓
Device heterogeneity	Low	High	High	High
Computational burden on devices	Low	High	Low	Low
Communication overhead on devices	Low	Low	High	High

Table 2.1 Comparison of classic DML, FL, SL and DPFL. Classic DML refers to the traditional DML system that uses a distributed stochastic gradient descent [25] algorithm and operates in a cloud-only computing environment. In contrast, FL, SL, and DPFL adopt the training algorithms described in Section 2.4.1, Section 2.4.2, and Section 2.4.3, respectively. They are typically implemented across both cloud and edge computing environments.

1. *Existing DML paradigms for the edge require further optimizations to be practical for the edge.* FL has impractical training times due to the computational burden on resource-limited devices on which local models are trained [39]. SL and DPFL reduce the device-side computational workload by offloading parts of it to the server. However, SL does not scale well for a large number of devices, as it requires round-robin style training between devices and the server [169, 133]. DPFL retains the benefits of parallel local training similar to FL but introduces additional communication overheads, such as the exchange of intermediate results between devices and the server, creating a new bottleneck during training. Deploying DML paradigms at the edge, therefore, necessitates further optimization to address the challenges related to impractical training latency, inefficient parallelization, and large communication costs.
2. *Existing solutions for optimizing DML paradigms are often limited in scope.* For instance, techniques of partial training are often adopted to reduce the computational burden on devices in FL training [71, 3, 72, 101]. However, these techniques typically result in significant accuracy degradation [19]. Additionally, while there is research that combines FL with SL to leverage a higher degree of parallelism in classic SL, it often overlooks the issue of device heterogeneity [165, 53]. Furthermore, in DPFL, the communication cost between devices and the server is rarely considered. While a few works [47, 53] adopt local loss to reduce communication costs, they do not account for the communication cost of activations. In summary, existing efforts to

optimize these three DML paradigms are still in their infancy and do not provide effective solutions, as they typically focus on improving only one among many aspects of system performance.

3. *Evaluation on physical devices is necessary to verify the performance of DML paradigms and the accompanying optimization techniques.* Evaluating DML systems on real-world prototypes is essential to identify real issues and to demonstrate the effectiveness of different DML paradigms and the optimization approaches applied to them. For example, lightweight DNN optimization techniques have been proven to effectively reduce DNN inference time in the cloud and on devices [60, 193]. However, evaluation on real IoT devices demonstrates that the optimization techniques within DML have limited training performance on the edge. For instance, a lightweight convolutional neural network (CNN), MobileNetV1 [60], required over 8 hours on Raspberry Pi 3 single-board computers to complete one round of training in an FL training paradigm [39]. Another example is local loss-based DPFL training, which theoretically can reduce half of the communication in DPFL. However, evaluation on real-world network conditions shows only marginal improvement in communication efficiency. The reason is that the actual bottleneck, which is the communication of the activation is not addressed in this approach [178].

The classic DML, FL, SL, and DPFL paradigms cannot be directly applied to build an efficient DML system at the edge. Moreover, existing optimization efforts for these paradigms are ineffective in enhancing overall system performance and are rarely evaluated on real IoT testbeds. Therefore, this thesis proposes three techniques to address the drawbacks of different DML paradigms and fill the gaps in existing work, which will be discussed in Chapter 3, Chapter 4, and Chapter 5. Additionally, a holistic framework integrating the three techniques is presented in Chapter 6 and evaluated on real-world testbeds to demonstrate the overall improvement in system performance when building DML at the edge.

Chapter 3

Offloading Workload from IoT Devices to Edge Nodes

This chapter explores the opportunity of employing DNN partitioning-based offloading techniques to enhance the training speed of FL on IoT devices. Specifically, this chapter addresses three questions: (i) Can DNN partitioning-based offloading accelerate the training time of an FL system at the edge, and to what extent can acceleration be achieved? (ii) How to manage device heterogeneity in the context of DNN partitioning-based offloading (iii) What impact will dynamic operational conditions, such as network bandwidth, have on offloading performance, and how can the DNN partitioning technique adjust to this?

This chapter introduces FedAdapt, a DNN partitioning-based FL framework to tackle these questions. FedAdapt utilizes DNN partitioning to offload part of the DNN training to the edge server, reducing the computation overhead on IoT devices. To address device heterogeneity, FedAdapt integrates an RL agent and device clustering strategy to generate diverse offloading strategies. In addition, the RL agent in FedAdapt makes dynamic offloading decisions in response to changes in network bandwidth during training.

A real-world testbed is built, which comprises five IoT devices and an edge server for testing the performance of FedAdapt. The experimental studies highlight that FedAdapt significantly reduces the total training time by up to 40% while achieving the same accuracy and convergence speed compared to classic FL on two popular DNN models.

3.1 Motivation

In the recent decade, FL has emerged as a machine learning paradigm for privacy-preserving distributed training utilizing private data generated on the network edge [185, 75, 12].

As introduced in Chapter 2.4, in classic FL architecture, the computationally intensive workload of training the DNN is independently carried out on individual devices, which may include IoT devices with limited computational capabilities. In contrast, a server located at the edge of the network or in the cloud is responsible for aggregating the weights sent from these devices, a process that is relatively less computationally demanding. However, the server has more computational resources compared to each IoT device. This naturally raises the question: Can offloading the computational workload from IoT devices to the server improve the performance of the FL system? To answer the above question, this thesis will first explore three challenges associated with a typical FL system.

The classic FL architecture has limitations that hinder its widespread adoption and efficiency in IoT settings. *The first challenge involves the impractical training times encountered on computationally constrained IoT devices.* Due to the resource limitations of such devices in comparison to large servers or specialized clusters designed for centralized ML model training, the time taken to execute FL training on these devices can be prohibitively long, rendering FL less practical for many real-world IoT applications [24, 171, 39]. For instance, experimental studies have shown that training a lightweight convolutional neural network, such as MobileNetV1 [60], on Raspberry Pi3 single board computers, can take over 8 hours for a single round of FL training [39]. Moreover, a typical FL training usually requires hundreds of rounds for the convergence, leading to impractical training times. Consequently, there is a need for novel techniques that can accelerate the training process on IoT devices in FL.

The second key challenge relates to the computational heterogeneity of IoT devices, leading to straggler devices during FL. In an FL system, devices have diverse computation capabilities, with some devices being faster and others slower. The slowest devices in the FL system are called stragglers. In synchronous FL, the stragglers impede the overall training speed since the aggregating server must wait for all devices to complete their training before proceeding with model updates [85, 31, 18]. Stragglers will be commonly present during FL on an edge computing system consisting of diverse IoT devices with varying computational capabilities and heterogeneous hardware architectures. Asynchronous FL approaches have been proposed as a potential solution to mitigate the straggler problem, allowing devices to operate independently without synchronization. However, this approach adversely impacts model accuracy, as not all devices contribute equally to training [184, 46]. As a result, there is a need for innovative methods to minimize the impact of stragglers on FL efficiency and model accuracy.

The third significant challenge lies in addressing the influence of varying operational conditions, such as fluctuations in network bandwidth between devices and the server.

The performance of FL depends on the communication conditions between devices and the central server. Changes in network conditions can create communication bottlenecks and substantially affect the training time and overall efficiency of FL [11, 85]. Adaptive and context-aware strategies are required to dynamically respond to variations in network conditions to optimize training and maintain robust performance across different network conditions.

To address the aforementioned challenges and effectively enhance FL efficiency in IoT environments, this thesis proposes the use of DNN partitioning-based offloading techniques to alleviate the burden of executing computationally intensive workloads on resource-constrained devices. Specifically, considering the challenges in FL systems mentioned above, this chapter aims to present a framework, addressing the following research problems:

1. **RP1:** Can DNN partitioning-based offloading effectively accelerate FL training on resource-constrained IoT devices?
2. **RP2:** How can the impact of computational heterogeneity among IoT devices be identified and minimized automatically in the DNN partitioning-based offloading strategy?
3. **RP3:** What adaptive strategies can respond and adapt to changing network conditions during the FL training process in the DNN partitioning-based offloading technique?

In summary, FL provides an opportunity for enabling training machine learning models in a privacy-preserving manner on IoT devices. However, it requires addressing the challenges of impractical training times on resource-constrained IoT devices, minimizing the impact of computational heterogeneity, and handling varying operational conditions for achieving performance-efficient FL on IoT devices. One solution to this is DNN partitioning-based offloading techniques where partial training workload is offloaded to the server with more computation resources. Therefore, this chapter proposes FedAdapt- an efficient DNN partitioning-based offloading framework for enhancing FL training performance on IoT devices. The next section will provide an overview of FedAdapt and how it is incorporated within the classic FL system.

3.2 FedAdapt Overview

This section presents an overview of FedAdapt and how FedAdapt is incorporated into the conventional FL system. Figure 3.1 presents the overall architecture of FedAdapt, which consists of four key modules: (1) *Pre-processor*, (2) *Clustering Module*, (3) *Trained RL Agent*,

and (4) *Post-processor*. By combining these modules, FedAdapt can effectively accelerate FL training on IoT devices with adaptive DNN partitioning-based offloading strategies.

In particular, following the completion of last FL round (Round $t - 1$), the *Pre-processor* on the server collects essential observations regarding the record of training performance of the devices, denoted as the states of devices. This includes gathering information in terms of the average training latency of each device (denoted as training time per iteration), such as the training time for one batch of training samples, and measuring the network bandwidth between each device and the central server. The training time per iteration is normalized by the *Pre-processor* among devices with varying computation capabilities. The min-max normalization [128] is used in the *Pre-processor* module.

After collecting normalized training times, the *Clustering Module* in FedAdapt groups devices with similar training times into computationally homogeneous clusters. Devices within the same cluster are considered to have comparable computational capabilities and will be assigned the same offloading strategy during training. In addition, the *Clustering Module* accounts for the network bandwidth between each device and the server, ensuring that the grouping takes into consideration the communication time between devices and the central server. The grouping reduces the complexity for training the RL agent since the RL agent only needs to consider groups rather than individual devices. The number of clustering groups is determined using heuristic algorithms, such as the elbow method [77]. For instance, the number of groups is three using the elbow methods in the experiments of this chapter.

To optimize the offloading strategy for each cluster, the *Trained RL Agent* utilizes the group information and observations obtained from the *Pre-processor*. The RL agent takes the state information as input and employs a fully connected neural network to generate offloading decisions (referred to as actions) for each cluster. The training process of the RL agent is a critical aspect and will be conducted offline in advance of FL training, which will be elaborated in Section 3.3.

The *Post-processor* plays a vital role in FedAdapt, as it effectively maps the offloading decisions generated by the *Trained RL Agent* to the devices within each cluster. Consequently, all devices within a cluster execute the same offloading strategy, leading to training speedup of local device training. The offloading strategy indicates which layers of the DNN model will remain on each device and which layers will be offloaded to the central server for the current FL round (Round t).

Among all the aforementioned modules, FedAdapt incorporates three fundamental techniques in the FL training process:

1. **DNN partitioning-based offloading:** FedAdapt adopts a DNN partitioning-based offloading approach, where the DNN model used in FL is horizontally partitioned by layers.

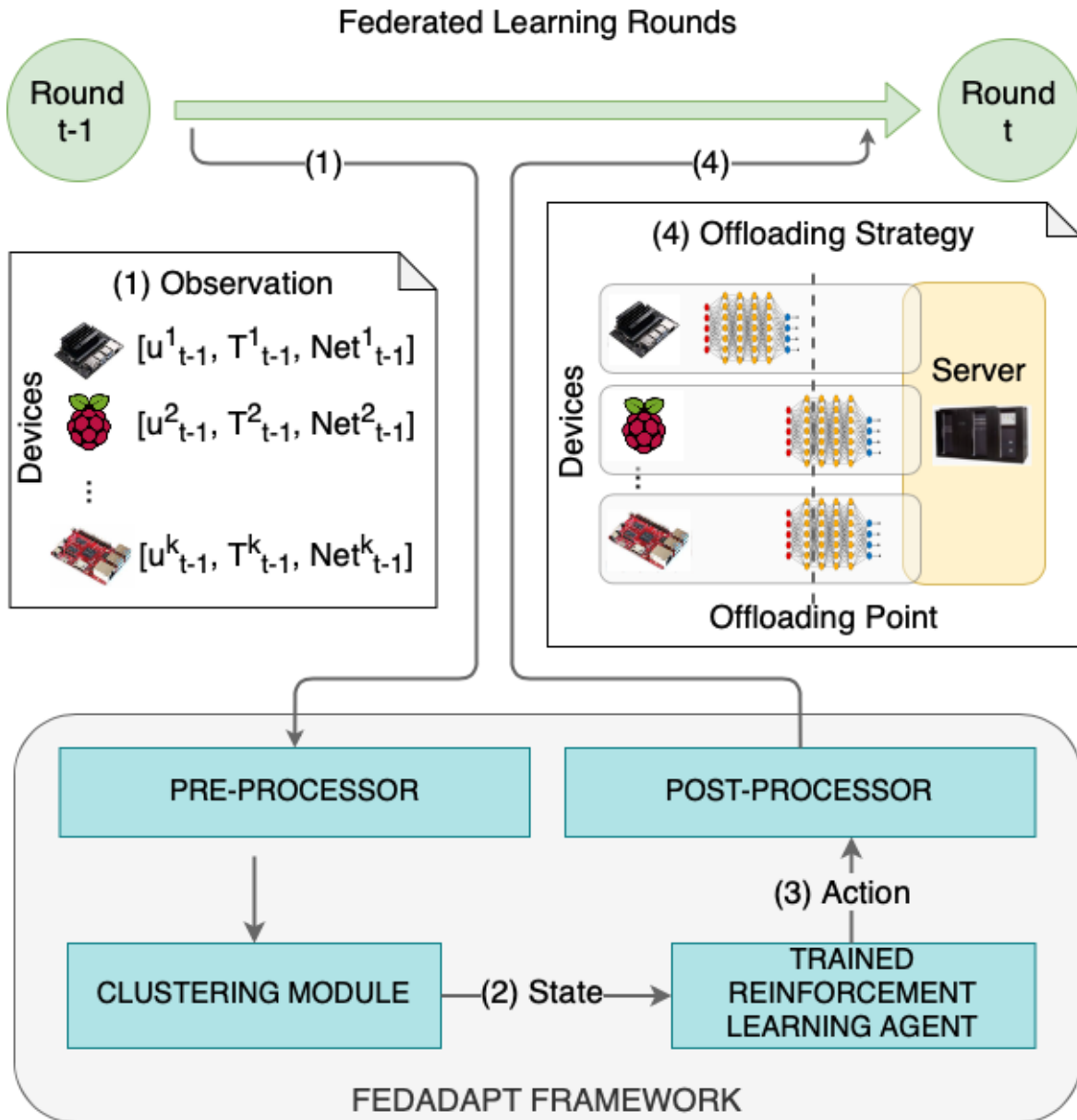


Fig. 3.1 FedAdapt framework and its positioning within FL.

By identifying the PP, certain layers of the DNN can be offloaded from computationally constrained devices to the server while the remaining layers are executed on the devices. This approach accelerates FL training by offloading the computational workload to more capable resources on the server, provided that the network bandwidth is sufficient to transfer the intermediate results efficiently. This technique directly addresses the research problem (RP1) mentioned in Section 3.1.

2. Reinforcement learning technique: FedAdapt employs RL as an automated technique to address the challenge of computational heterogeneity among devices, which often leads to stragglers in FL, as mentioned in RP2 mentioned in Section 3.1. The RL-based approach enables the Post-processor to identify the PP for each individual device before the FL round commences. This results in an optimized offloading strategy being executed for each device, thereby identifying diverse computation capabilities of devices, mitigating the impact of stragglers and enhancing overall training efficiency. To be able to scale up to handle a larger number of devices and determine offloading strategies for each of them, FedAdapt employs a clustering-based approach to group devices with similar computational performance, further aiding in the optimization of RL-based decision making.

3. Optimized RL for changing operational conditions: Operational conditions, such as network bandwidth between devices and the server, can vary during the FL training process. This variation can influence the optimal offloading strategy generated by the RL agent. To account for such dynamic changes and generate optimal offloading strategies, FedAdapt further optimizes the design of the RL agent techniques by considering changes in network bandwidth. This ensures that the offloading decisions dynamically adapt to varying network conditions, addressing RP3 initially raised in Section 3.1.

The benefits for adopting RL: RL is a widely adopted sequential decision optimization technique in various domains, such as automated control and optimization tasks [44, 142, 153, 107]. In the context of FedAdapt, RL presents two advantages. Firstly, RL provides an automated and data-driven mechanism to generate optimal offloading strategies for the DNN partitioning of IoT devices. The objective of the optimization is to maximize the reward, which, in FedAdapt, is represented by the reduction in training time. Unlike existing research that assumes access to detailed hardware configurations of all devices in a white box manner [85, 184], RL eliminates the need for explicit profiling of device hardware by only using training records from the previous rounds. This is valuable in real-world FL applications where obtaining hardware details for all devices might not be feasible. Moreover, the one-time estimation of the training time of a device is often imprecise, as factors such as resource availability and network bandwidth may fluctuate during the training process. However, the RL agent in FedAdapt efficiently adapts to such dynamic conditions and

optimizes the offloading strategy. Secondly, the use of a trained RL agent in FedAdapt offers the advantage of reusability for similar FL tasks. This means that the trained RL agent can be applied to multiple FL scenarios without the need for retraining specifically for each task. Section 3.4.4 verifies the performance of reusing the RL agent without retraining. Without the need for additional training tailored to a specific DNN model, the RL agent achieves a 57% reduction in training time per round. Alternatives to RL in this context include traditional optimization methods, such as rule-based algorithms and heuristic approaches. These methods are computationally lightweight but have reduced flexibility and adaptability compared to RL. Additionally, they require detailed device profiling to gather information of each device participating in FL. In summary, FedAdapt is an adaptive framework designed to optimize the DNN partitioning-based offloading strategy for FL in IoT environments. By integrating DNN partitioning-based offloading techniques, RL-based decision making, and optimized RL strategies for network bandwidth, FedAdapt addresses the challenges posed by resource-constrained devices, computation heterogeneity, and varying network conditions, leading to enhanced training efficiency, reduced straggler issues, and improved adaptivity.

3.3 FedAdapt Modules

This section presents the technical designs of the four key FedAdapt modules, namely, the *Pre-processor*, *Clustering Module*, *Trained RL Agent*, and *Post-processor* considered in the previous section.

3.3.1 Pre-processor

The *Pre-processor* is responsible for collecting essential observations and data regarding the participating devices in the FedAdapt system. It records the training information of the last FL round of each device and processes it to the group states required by the RL offloading agent to inform the subsequent decision-making processes and optimization strategies. In particular, the *Pre-processor* has the following functionalities:

Collecting and normalizing training speed data: The first task of the *Pre-processor* is to record and gather training speeds of each individual device participating in the last round of FedAdapt. Training speed is defined as the time taken by a device to complete one iteration of training for a batch of training samples. This information gives insights into the processing power and computation capability of each device during the last round of training. Devices with more computational resources can complete iterations faster, while resource-constrained devices may take longer. By collecting this data, the *Pre-processor*

provides data for the RL agent to optimize the PP for diverse devices in the next round of FL training.

Measuring network bandwidth: Another key function of the *Pre-processor* is to measure the network bandwidth between each device and the central server. The average available network bandwidth is calculated at the end of each training round. Communication between devices and the server is essential information for optimal DNN partitioning, as data exchange and model updates also contribute to the overall training time. By assessing the network bandwidth for each device-server pair, the *Pre-processor* gains a comprehensive understanding of the communication time for devices. Devices with higher network bandwidth can communicate faster with the server, facilitating faster data exchange and model updates. This information is useful for devising offloading strategies and ensuring that communication overhead is considered in FedAdapt.

Data preparation: Once the training speed and network bandwidth data are collected, the *Pre-processor* prepares the data. First, data is normalized for consistently measuring across devices. In addition, by normalizing the training time per iteration and network bandwidth values, the *Pre-processor* can also ensure that these parameters are standardized during each round of decision-making processes within the entire FL training. It is worth noting that data is prepared at the beginning of each round by analyzing the training records from the previous round.

3.3.2 Clustering

The *Clustering Module* within the FedAdapt framework provides information for the *Trained RL Agent*. The integration of RL techniques and the clustering empowers FedAdapt to make informed decisions regarding the offloading strategy for each device group.

Grouping devices based on training speed: The *Clustering Module* groups devices based on their training speed. Devices that have similar training speeds are categorized into the same cluster, reflecting their comparable computational performance. Specifically, by considering the recorded training time per iteration and the network bandwidth collected by *Pre-processor*, this module groups devices with similar training speed and network bandwidth into a cluster to ensure that one group can capture the general characteristics of computation and communication of a group of devices. In the experiments of this chapter, the number of clustering groups is determined to be three using the heuristic elbow method [77]. The adoption of clustering before the RL agent also significantly reduces the complexity of the application of RL techniques to determine the optimal offloading strategies. This is because the RL agent only needs to optimize the offloading strategies for each groups rather than for each individual device.

Accounting for network bandwidth: Beyond considering training speeds, the *Clustering Module* also takes into account the network bandwidth between each device and the central server. This additional information enables FedAdapt to adapt to the changes in network bandwidth, as it incorporates the communication time of each device-server pair. In the case of changes in network bandwidth during training, devices will be dynamically assigned to different groups with diverse offloading strategies. For example, due to variable network bandwidth, a device may experience a lower network bandwidth at round t . The original partitioning and offloading decision may slow down the overall training time compared to device-native training. To address this, at the end of round t , the *Pre-processor* will calculate the training speed and network bandwidth using the training records from round t . The *Clustering Module* will then reassign the device to groups that utilize device-native training to adapt to the change in the available network bandwidth.

Execution, running frequency, and scalability of the *Clustering Module*: The *Clustering Module* runs on the server, which has access to both the training speeds and network bandwidths of all participating devices. Before training begins, the group center is calculated by the *Clustering Module* on the server measuring the initial device training speed and network bandwidth, and this step is performed only once. Following this, during FL training, the *Clustering Module* reassigns devices to the most suitable group at the end of each round to account for any changes in training speed and network bandwidth.

The information required by the *Clustering Module* is gathered by the *Pre-processor*, which monitors and records the training time per iteration and the network bandwidth between devices and the central server during each round of training. Although the computational complexity increases with the number of devices, clustering is executed only once before FL training. In addition, clustering has a low computation cost as it is based on only two features: training speed and network bandwidth. Theoretically, it is anticipated that with thousands of devices, the overhead in clustering would be low and acceptable in a typical FL scenarios.

3.3.3 RL agent

This subsection details technical designs of the *Trained RL agent* by firstly discussing the reasons why RL is adopted in FedAdapt, followed by the formulation of the RL optimization goals, the method for training the RL agent, and the essential designs of the RL technique: the input state, output action, and the reward function.

Problem Formulation and Optimization Objective: Before considering the details of the design of the RL agent in FedAdapt, the optimization goals are first formulated for the RL agent.

Table 3.1 Notation used in FedAdapt

Notation	Description
K	Number of devices in a FL task
t	Time step (index) of round t
k	Denote a device participating in FL
W^k	Training workload of device k
Net_t^k	Network bandwidth at round t between device k and the server
C_t^k	Training speed at round t of device k
$L(\mu_t^k)$	Total volume of communication of device k at round t , including activations and gradients
s	Server coordinating the FL task
C_t^s	Training speed at round t of server s
μ_t^k	Offloading strategy at round t of device k
μ_t	The static offloading strategy at round t for all devices or groups
T_t^k	Local training time at round t of device k
B^k	Training time of device k without offloading
T_t	FL training time at t
S_t	State at round t in RL
A_t	Action generated by RL at round t
R_t	Reward at round t
G	Total number of groups
g	A representative device in group g
W^g	Training workload of the representative device of g
Net_t^g	Network bandwidth of the representative device of g at round t
C_t^g	Training speed of the representative device of g at round t
μ_t^g	Offloading strategy of the representative device of g at round t
T_t^g	Training time of the representative device of g at round t
f_{norm}	Normalization function used in RL to calculate R_t
γ	a discount factor balancing the short-term and long-term rewards in RL.

In an FL training task, the network bandwidth between the device and server may vary across rounds. However, it is assumed that the network bandwidth does not dynamically change during a single FL round. Consider a scenario where FL training is conducted with K devices, and each device has a training workload W^k for each round. The FL task involving a server s has a training speed of C_t^s at round t , while the set of participating devices $\{k\}_{k=1}^K$ has training speeds denoted by C_t^k . Additionally, the network bandwidth between each device and the server is represented as Net_t^k . The offloading strategy for each device is denoted as μ_t^k , indicating the proportion of computation that is executed on the device at round t . In other words, for a round of FL training on device k , the proportion of workload executed on the device is $\mu_t^k W^k$, while $(1 - \mu_t^k)W^k$ is offloaded to the server.

Let $L(\mu_t^k)$ represent the size of the feature maps transferred between the device and server during the training of round t . It is worth noting that $L(\mu_t^k)$ depends on the offloading strategy μ_t^k , as the strategy determines the size of the transferred feature map. Finally, the training time for device k at round t can be calculated as follows:

$$T_t^k = \frac{\mu_t^k W^k}{C_t^k} + \frac{(1 - \mu_t^k)W^k}{C_t^s} + \frac{L(\mu_t^k)}{Net_t^k} \quad (3.1)$$

where the $\frac{\mu_t^k W^k}{C_t^k}$ and $\frac{(1 - \mu_t^k)W^k}{C_t^s}$ is the training time on the device and on the server, respectively. $\frac{L(\mu_t^k)}{Net_t^k}$ is the communication time during training.

In round t , variables such as W^k , C_t^s , C_t^k , and Net_t^k are either constants or external variables that are not controlled by FedAdapt. However, the offloading strategy μ_t^k for each device is entirely controlled by FedAdapt. Here, μ_t^k represents the PP for each device, and the collection of PPs for all K devices is denoted as $\mu_t = \{\mu_t^k\}_{k=1}^K$. DNN partitioning has also been explored in the literature of SL and SFL. Refer to the details of these techniques and comparison with FedAdapt in Section 3.5. However, it is worth noting that, in these methods, μ_t^k remains uniform across all devices and rounds, denoted as $\bar{\mu}$.

During synchronous training, the server waits for all devices to complete their training before proceeding, resulting in an FL training time for round t denoted as $T_t = \max\{\{T_t^k\}_{k=1}^K\}$. Here, T_t^k represents the training time required for one round of training on device k . A summary of the computational workload for all K devices and the training time required for one round using different methods is presented in Table 3.2. This table provides an overview of the comparative computational complexity and training times for the various techniques.

Table 3.2 Computational workload (on the device) and training time of FL, SL, SFL, and FedAdapt for one round

Methods	Computation	Training Time
FL	$\sum_{k=1}^K W^k$	$\max\{\{\frac{W^k}{C_t^k}\}_{k=1}^K\}$
SL	$\sum_{k=1}^K \bar{\mu}W^k$	$\sum_{k=1}^K \frac{\bar{\mu}W^k}{C_t^k} + \frac{(1-\bar{\mu})W^k}{C_t^s} + \frac{L(\bar{\mu})}{Net_t^k}$
SFL	$\sum_{k=1}^K \bar{\mu}W^k$	$\max\{\{\frac{\bar{\mu}W^k}{C_t^k} + \frac{(1-\bar{\mu})W^k}{C_t^s} + \frac{L(\bar{\mu})}{Net_t^k}\}_{k=1}^K\}$
FedAdapt	$\sum_{k=1}^K \mu_t^k W^k$	$\max\{\{\frac{\mu_t^k W^k}{C_t^k} + \frac{(1-\mu_t^k)W^k}{C_t^s} + \frac{L(\mu_t^k)}{Net_t^k}\}_{k=1}^K\}$

To reduce the training time for all devices in a round, the optimization target is defined as minimizing the average training time for K devices as follows:

$$\begin{aligned} & \underset{\mu_t^k}{\text{minimize}} \quad \frac{1}{K} \sum_{k=1}^K T_t^k \\ & \text{where} \quad T_t^k = \frac{\mu_t^k W^k}{C_t^k} + \frac{(1-\mu_t^k)W^k}{C_t^s} + \frac{L(\mu_t^k)}{Net_t^k} \end{aligned} \quad (3.2)$$

The training time for round t , denoted as T_t , is determined by the maximum training time among all participating devices. However, FedAdapt goes beyond optimizing the maximum training time, which is constrained by the slowest devices (stragglers). Instead, FedAdapt aims to reduce the training time for each individual device. By doing so, it effectively reduces the computational workload on all devices.

In FedAdapt, the objective is to minimize the average training time across all K devices to improve the overall training efficiency. To accomplish this, FedAdapt optimizes the offloading strategy μ_t for each round based on various operational conditions, such as C_t^s , C_t^k , and Net_t^k . By adapting μ_t to these dynamic conditions for each round, FedAdapt ensures efficient and effective offloading decisions, ultimately resulting in reduced average training times for all devices and enhanced performance across all FL rounds.

Optimization under clustering with RL: An initial design strategy could involve generating an individual offloading action for each device. However, this approach has two limitations. Firstly, if the number of participating devices changes during FL training, the RL agent may face challenges in generating offloading actions due to fixed input and output dimensions of the DNN used by the RL agent at the start of FL training. Secondly, as the

number of devices K grows larger, training the RL agent becomes increasingly difficult due to the exponentially expanding action space that needs to be explored.

To address these challenges, a clustering technique is employed at the beginning of each FL round. The clustering process groups homogeneous devices based on training time per iteration and network bandwidth between the device and server into G groups. The number of groups G is determined using heuristic algorithms, such as the elbow method [77]. By forming G groups, the input state and output action dimensions are reduced from K devices to G groups, mitigating the issues arising from the increasing number of devices.

Therefore, the objective is formulated as:

$$\begin{aligned} & \underset{\mu_t^g}{\text{minimize}} && \frac{1}{G} \sum_{g=1}^G T_t^g \\ & \text{where} && T_t^g = \frac{\mu_t^g W^g}{C_t^g} + \frac{(1 - \mu_t^g) W^g}{C_t^s} + \frac{L(\mu_t^g)}{Net_t^g} \end{aligned} \quad (3.3)$$

where g is defined as a representative device in the group that has the maximum training time. In other words, for each group, FedAdapt considers all devices in a group as homogeneous devices which means that they have similar computation capability and network bandwidth. The device with the longest training time is used to represent the group, and its corresponding network bandwidth is also used to represent the group's network bandwidth. Therefore, W^g , Net_t^g , C_t^g , μ_t^g , and T_t^g are bounded by the representative device in each group.

Optimizing training time per round or entire training rounds: In FedAdapt, both Equation 3.2 and Equation 3.3 aim to minimize the average training time across devices for each round, thereby reducing the total training time over all rounds T . However, to align with the design of RL training, which typically seeks to maximize accumulated rewards throughout FL, FedAdapt incorporates a reward mechanism based on a discounted cumulative reward, as shown in Equation 3.4.

$$R = \sum_{t=0}^T \gamma^t r_t \quad (3.4)$$

In this approach, future rewards are discounted using a factor γ , which ranges between 0 and 1. This discounted reward allows the agent to prioritize immediate rewards over future ones. The discount factor γ controls the preference of the agent for short-term or long-term rewards. A higher γ places more weight on future rewards, while a lower γ makes the agent more short-sighted. Notably, when $\gamma = 0$, the optimization focuses on minimizing the training time per round, whereas when $\gamma = 1$, it is equivalent to minimizing the total training time over all rounds.

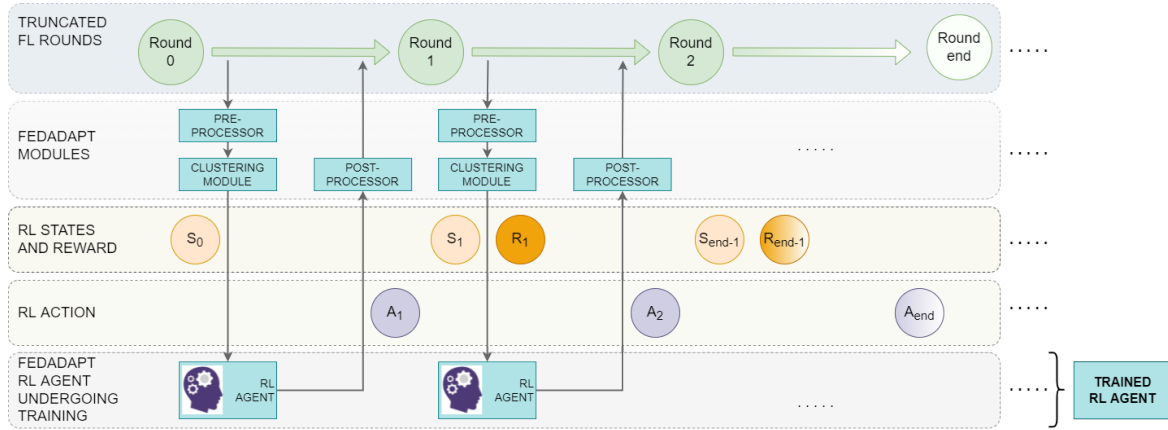


Fig. 3.2 Training of the RL agent used in FedAdapt

Overall training approach: Given the optimization goal as shown in Equation 3.3, the overall training approach of the RL agent in FedAdapt is firstly presented. Figure 3.2 describes one episode of training the RL agent. An episode in this context refers to the entire FL training task, which includes R rounds. Each step represents one round of FL training. The state used in the RL agent is obtained from the *Clustering* Module and consists of normalized values for training time and action.

The RL agent in FedAdapt employs a DNN comprising three layers. It takes the current input state (S_t) as input and produces the offloading action A_t . It is important to note that the offloading action differs from the offloading strategy generated by the *Post-processor*. The action generated by the RL agent is a value between 0 and 1, representing the proportion of training workload executed on a group of devices. In contrast, the offloading strategy produced by the *Post-processor* involves mapping this value to a PP for each device. A trained RL agent is obtained at the end of training. Its objective is to maximize the cumulative rewards over each round, aligning with Equation 3.3. Training commences after the first round, where classic FL training (no offloading) is used to generate the initial state S_0 . Then the RL agent iteratively learns and optimizes its offloading decisions based on observed rewards and the accumulated experience throughout the training episodes.

State and action: The input state for the RL agent includes the maximum local training time of the devices within a group. During each training round, the RL agent generates offloading actions for each group. The Post-processor maps these actions to the corresponding devices in the group, denoted as μ_t^g . For example, in a VGG5 [158] model with three convolutional layers and two fully connected layers, there are five offloading actions. Each output action for a group represented as μ_t^g is designed as a real value ranging from zero to one. This design allows the RL agent to adapt to multiple DNN models. μ_t^g is then mapped

to represent the percentage of the total computational workload of the DNN that should be placed on the device.

Once μ_t^g is obtained, the number of Floating Point Operations (FLOPs) is calculated and set as the target workload for the devices. The PP that is closest to the target workload is selected as the final offloading decision for the group. Equation 3.5 illustrates the input state and output action for the RL agent at round t . This enables the RL agent to dynamically adjust the offloading strategy for each group of devices, taking into account the specific DNN model architecture and workload distribution to achieve efficient and effective offloading decisions.

$$\begin{aligned} S_t &= \{T_t^g, \mu_{t-1}^g\}_{g=1}^G \\ A_t &= \{\mu_t^g\}_{g=1}^G \\ \text{where } \mu_t^g &\in (0, 1] \end{aligned} \quad (3.5)$$

Reward function: At the end of each FL training round, the obtained reward is represented as R_t . To achieve the objective outlined in Equation 3.3, one option is to use the average training time as the reward. However, this approach may lead to biased rewards, as the device with the longest training time could dominate the overall reward. To mitigate this issue, a normalization function denoted as f_{norm} is employed to calculate the reward. The baseline training time for each device when no DNN model is offloaded is denoted as B^k . The training time of device k (T_t^k) is then normalized with respect to B^k using Equation 3.6.

$$\begin{aligned} R_t &= \sum_{k=1}^K f_{norm}(T_t^k, B^k) \\ f_{norm} &= \begin{cases} 1 - \frac{T_t^k}{B^k} & T_t^k \leq B^k \\ \frac{B^k}{T_t^k} - 1 & T_t^k > B^k \end{cases} \end{aligned} \quad (3.6)$$

Choice of RL algorithm: For training the RL agent, various RL algorithms are available, such as DQN [163, 114] and REINFORCE [125]. However, in this research, Proximal Policy Optimization (PPO) [147] is chosen as the preferred method.

PPO is considered a popular RL algorithm that offers several advantages. Firstly, it is relatively straightforward to use and has demonstrated superior performance on standard RL benchmarks [147]. Unlike DQN, which determines the optimal action by evaluating all possible actions using the Q-network [114], PPO generates explicit actions using a policy network [147]. In FedAdapt, the action space is continuous, represented as $\mu_t^g \in (0, 1]$,

making it impossible to enumerate all possible actions for each group if DQN were to be adopted.

Furthermore, PPO is an off-policy RL algorithm, which utilizes trajectory data from previous explorations (interactions between the agent and the environment) for training. This feature improves training efficiency, especially considering that interactions between the agent and the environment can be time-consuming. Thus, PPO is chosen as the RL algorithm in FedAdapt, enabling the RL agent to efficiently learn and optimize offloading strategies that minimize training time across all devices during FL training.

RL training methodology: Finally, the specific training methodology used in training the RL agent in FedAdapt is described. The RL agent comprises two fully connected networks, namely the actor and critic networks. Both networks have the same architecture comprising three layers. When the RL agent is trained, the critic network is adopted to assist in the training of the actor network. The actor network is trained to output the offloading action. After completing training, only the actor network will be used to provide the offloading action. Ideally, the RL agent should be trained online during an FL task. However, if the RL agent is trained online during an FL task, the overall training time for the RL agent is the time for all rounds of FL training. The RL agent will need to wait until the completion of each round to obtain the training speed required for calculating the reward, which is time-consuming. To address huge overheads in training the RL agent, it is trained in an offline manner before FL tasks. In addition, to accelerate the training of the RL agent, the number of batches used for each round in RL training is significantly reduced, referred to as truncated FL rounds in Figure 3.2. The training time per batch for each device is recorded instead of the training time of a round as an element of the input state. The FL model will be trained again with normal rounds (beyond the truncated rounds) after the Trained RL agent is obtained.

3.3.4 Post-processor

The role of the *Post-processor* in the FedAdapt framework is that it takes the output of the Trained RL agent, which consists of offloading decisions for each group, and effectively maps these decisions onto the individual devices within each group. Since all devices in a group are considered computationally homogeneous, they will execute the same offloading strategy. This strategy specifies which layers of the DNN model will be placed on each device for the specific FL Round t .

In detail, the *Post-processor* ensures that the offloading decisions generated by the RL agent are translated into practical actions for each device. By implementing these strategies, FedAdapt dynamically partitions and distributes the DNN model across the

devices, optimizing the computational workload on each device and adapting to changing network conditions.

3.4 Evaluation

This section conducts a series of experiments to empirically validate the performance of FedAdapt. The experiments are organized in response to the research problems posed in Section 3.1 with each subsection addressing a specific aspect of FedAdapt.

Section 3.4.1 provides the results obtained from examining the acceleration performance of applying DNN partitioning-based offloading in FL, addressing the first research problem (RP1). The experiments in this section focus on evaluating how offloading certain layers of the DNN model from devices to the server reduces the overall training time.

Section 3.4.2 presents the outcomes of applying the RL technique in FedAdapt, tackling the challenge of identifying diverse IoT devices and minimizing the impact of computational heterogeneity among devices (RP2). By grouping devices with similar computational capabilities using the *Clustering Module*, FedAdapt optimizes offloading strategies tailored to each group, thus reducing the impact of stragglers during FL training.

Section 3.4.3 analyzes the results obtained when optimizing the RL technique to account for changing network bandwidth, addressing the third research problem (RP3). By adapting offloading decisions based on varying network conditions for each round, FedAdapt further enhances training efficiency.

To demonstrate the benefits of FedAdapt, this section conducts comparative analyses with classic FL. In Section 3.4.4, experiments showing how FedAdapt outperforms traditional FL approaches in terms of training time reduction and overall efficiency are conducted, highlighting the effectiveness and practical applicability of the proposed framework.

3.4.1 Acceleration of DNN partitioning-based Offloading in FL

The validation of the assumption that DNN partitioning-based offloading can accelerate FL training on computationally limited IoT devices, such as Raspberry Pi single-board computers, is conducted through an empirical study. The Raspberry Pi device serves as the testbed in this chapter due to its popularity as an IoT device and widespread usage in existing works [39, 184, 48]. The study involves two shallow CNNs, namely VGG5 [158] and VGG8 [158] which is similar to the existing works [173]. In addition, this subsection carried out the experiments under various network bandwidth conditions, including Wi-Fi

(75 Mbps and 50 Mbps with equal uplink and downlink bandwidth), 4G+ (25 Mbps uplink and 50 Mbps downlink), and 4G (10 Mbps uplink and 20 Mbps downlink) connections ¹.

The study aims to demonstrate the following key findings: (i) DNN partitioning-based offloading from a device to a server leads to a significant reduction in FL training time when compared to classic FL, where all layers of the DNN are executed on the device. Existing research has already highlighted that computational time on resource-constrained devices is a major bottleneck in FL [53, 171, 39]. By offloading certain layers to the server with more resources, FedAdapt efficiently mitigates this computational limitation and enhances the overall training performance on IoT devices.

(ii) The performance gain achieved through DNN partitioning-based offloading is substantial, effectively offsetting the communication overhead incurred during the transfer of activation and gradient feature maps between the device and the server. Despite the additional communication cost, the offloading strategy in FedAdapt proves highly beneficial in terms of reducing the overall FL training time.

Concerns about information leakage by offloading: The partitioning and offloading techniques used in FedAdapt require the transfer of intermediate activations and gradients, which may expose hidden information about the raw data. However, as the partitioning layer moves from the lower to the higher layers of the network, the information becomes less sensitive due to the filtering effect of non-linear layers (e.g. ReLU layer), which retain only high-level semantic details [63]. Therefore, to account for privacy, the potential for information leakage should be taken into account when selecting the partitioning point. For instance, sensitive data might be more vulnerable before a certain partition, and therefore, it is important to choose partition points that not only improve training speed but also align with privacy requirements set by a user. Furthermore, encryption techniques, such as differential privacy, can be utilized to encode intermediate activations and gradients before transmission [1, 195]. However, this area is not the focus of the thesis, but related differential privacy techniques can be incorporated into FedAdapt.

Setup: An experimental testbed is built comprising an edge server equipped with a 2.5GHz dual-core Intel i7 CPU laptop and an IoT device, specifically a Raspberry Pi 4 Model B featuring a 1.5GHz quad-core ARM Cortex-A72 CPU. To evaluate the performance gain of FL training with DNN partitioning-based offloading, a single IoT device is employed in this subsection. The Wi-Fi connectivity for the device is facilitated through the Virgin Media Super Hub 3 router. To emulate different network bandwidths between the device and the server, the Linux built-in network traffic control module `tc` is utilized. This enables the

¹<https://www.4g.co.uk/how-fast-is-4g/>

simulation of various network conditions to assess the impact of changing bandwidth on FL training performance.

For the experimental model, two popular DNN models are used, namely VGG5 [158] and VGG8 [158] models as outlined in Table 3.3. To simplify the presentation, batch normalization and non-linear layers (ReLU) are not explicitly shown in the table. The layers indicated with PP represent the empirical PPs in this study, wherein all layers after the PP can be offloaded to the server. Based on the literature on FL, the widely used CIFAR-10 dataset [79] is employed for the experiments, and a batch size of 100 is maintained consistently across all experiments.

Table 3.3 Architecture of the models used for evaluating FedAdapt. Convolution layers are denoted by C followed by the number of filters; filter size is 3×3 for all convolution layers, MaxPooling layer is MP, Fully Connected layer is FC with a given number of neurons, and the PP is followed with index.

Model	Architecture
VGG5	C32-MP(PP1)-C64-MP(PP2)-C64(PP3)-FC128-FC10(PP4)
VGG8	C32-C32-MP(PP1)-C64-C64-MP(PP2)-C128-C128(PP3)-FC128-FC10(PP4)

Table 3.4 and Table 3.5 present the training time per iteration of FL when using DNN partitioning-based offloading for all PPs of VGG5 and VGG8. VGG5 and VGG8 both have four possible PPs. In addition, the last PP in each model (PP4) corresponds to device native execution of the DNN as in classic FL. The results are an average of five independent runs. The best result for each value of network bandwidth is shown in bold.

Table 3.4 Training time per iteration when layer offloading is used in FL for VGG5 under different network bandwidths.

PP	75 Mbps (Wi-Fi)	50 Mbps	25 Mbps (4G+)	10 Mbps (4G)
PP1	2.38 ± 0.18	2.7 ± 0.03	3.52 ± 0.08	6.07 ± 0.21
PP2	3.61 ± 0.4	3.9 ± 0.48	4.36 ± 0.27	5.31 ± 0.24
PP3	5.24 ± 0.45	5.26 ± 0.19	5.42 ± 0.38	6.73 ± 0.71
PP4 (device native)	4.36 ± 0.66	4.36 ± 0.66	4.36 ± 0.66	4.36 ± 0.66

The optimal values for the training time per iteration for VGG5 and VGG8 models are found to be 2.38 seconds and 4.75 seconds, respectively, compared to 4.36 seconds and 10.61 seconds for classic FL in a Wi-Fi network. These results demonstrate a significant reduction in training time when employing DNN partitioning-based offloading into FL.

Table 3.5 Training time per iteration when layer offloading is used in FL for VGG8 under different network bandwidths.

PP	75 Mbps (Wi-Fi)	50 Mbps	25 Mbps (4G+)	10 Mbps (4G)
PP1	4.75 ± 0.24	5.29 ± 0.34	6.08 ± 0.35	8.84 ± 0.28
PP2	7.52 ± 0.55	8.37 ± 0.61	8.32 ± 0.69	9.95 ± 0.33
PP3	10.74 ± 1.3	11.98 ± 0.71	12 ± 1.05	15.93 ± 1.12
PP4 (device native)	10.61 ± 1.47	10.61 ± 1.47	10.61 ± 1.47	10.61 ± 1.47

In both VGG5 and VGG8 models, the best PP is PP1. This indicates that the most efficient performance is achieved when the majority of layers are offloaded from the resource-constrained device to the server. The training time is reduced by more than 45% for VGG5 and over 55% for VGG8 when compared to the classic FL approach. In addition, the best PP for both models is the pooling layer, where all layers beyond the pooling layer are offloaded. This choice is motivated by the fact that pooling layers are computationally less intensive and can reduce the volume of data (activation feature maps) that needs to be transferred between the device and the server. This strategy further contributes to the overall performance improvement in FedAdapt.

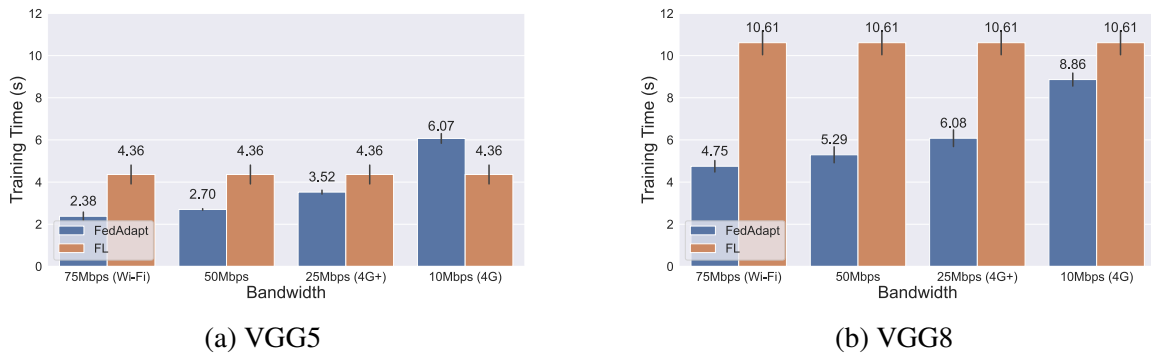


Fig. 3.3 Comparing training time per iteration in FedAdapt and classic FL

The results of the best PP achieved for the VGG5 and VGG8 models under four different network bandwidths are further presented in Figure 3.3a and Figure 3.3b, respectively. It is observed that employing layer offloading in FedAdapt leads to a reduction in the training time per iteration, with over 45% and 55% reductions achieved for the VGG5 and VGG8 models, respectively, in a Wi-Fi connection scenario.

However, as the network bandwidth decreases, the performance acceleration achieved by offloading diminishes. For instance, with a network bandwidth of 25 Mbps, which represents

the typical bandwidth of real-time 4G+ mobile networks, the training time is reduced by 19% and 43% for the VGG5 and VGG8 models, respectively. At an even lower bandwidth of 10 Mbps, offloading negatively impacts the performance of the VGG5 model, and the best PP is determined to be device-native execution. For the VGG8 model, offloading continues to be effective, reducing the training time by 17% at 10 Mbps bandwidth. This indicates that the effectiveness of offloading is influenced by the network bandwidth between the device and the server since frequent communication takes place during training. For the VGG5 model, device-native execution is more efficient when the bandwidth is limited to 10 Mbps. However, offloading the layers at PP1 from the device to the server becomes more effective for higher bandwidths. In contrast, for the VGG8 model, offloading consistently outperforms device-native execution across various bandwidths.

In summary, FedAdapt is envisioned to be highly beneficial in scenarios where distributed IoT devices possess limited computational resources and need to offload the FL training workload onto a server. Potential examples include home security cameras utilizing Wi-Fi and leveraging computational resources on a home hub, as well as wearable visual auxiliary equipment operating in both indoor and outdoor environments, utilizing both Wi-Fi and mobile networks. Potential learning tasks for these applications include object detection for security events in home cameras [104] or real-time scene recognition and object identification for visually impaired users in wearable devices [95].

3.4.2 RL Optimization for Heterogeneity

The challenge of computational heterogeneity among devices in FL often results in the presence of stragglers. Consequently, the straggler device can significantly impact the overall training time. In contrast, some IoT devices with more computation resources (e.g., Jetson Xavier) might not require the assistance of DNN partitioning-based offloading. To address this challenge, FedAdapt incorporates an innovative offloading strategy that can effectively account for device heterogeneity and minimize the impact of stragglers. The key feature enabling this capability is the adoption of an RL agent, responsible for selecting different PP for the participating devices in FL. The training process of the RL agent is guided by a reward function, as shown in Equation 3.6. For this particular experiment, the focus is solely on addressing device heterogeneity, and as such, changing network bandwidth is not considered.

By utilizing the RL-based offloading strategy, FedAdapt aims to optimize the training of each device, taking into account its computational capabilities. As a result, different devices can be assigned diverse offloading strategies tailored to their computation capabilities.

Setup: The IoT-edge server environment considered in this experiment consists of one server and five IoT devices. The server is the same as the one used in Section 3.4.1, equipped with a 2.5GHz dual-core Intel i7 CPU. The five devices include:

1. Two Raspberry Pi 4 devices denoted as Pi4¹ and Pi4², which have a 1.5GHz quad-core ARM Cortex-A72 CPU each.
2. Two Raspberry Pi 3 devices denoted as Pi3¹ and Pi3² Model B, equipped with a 1.2GHz quad-core ARM Cortex-A53 CPU.
3. One Jetson Xavier NX device denoted as Jetson, featuring embedded GPUs for accelerated processing.

To create a straggler device for experimental purposes, the running CPU frequency of Pi4² is manually set to 0.7GHz. All Raspberry Pi devices run the Raspbian GNU/Linux 10 (Buster) operating system, with Python version 3.7 and PyTorch version 1.4.0. Both the Jetson and the server have the same version of Python and PyTorch, and the Jetson also has the CuDNN library installed to utilize the GPU during training. All devices are connected to the server through a Wi-Fi network, using the router described in Section 3.4.1, with an average available bandwidth of 75 Mbps.

The DNN model used for the experiments is the VGG5, and it will be demonstrated in Section 3.4.4 that the RL agent trained for VGG5 can also be effectively employed for VGG8. The CIFAR-10 dataset is used for training and testing, containing 50,000 training samples and 10,000 testing samples. The training samples are evenly divided among the five devices without any overlaps, while the entire test dataset is available on the server. The FL task comprises 100 rounds, and the standard FedAvg [111] aggregation method is applied on the server. For data augmentation, the horizontal flip technique is used with a probability of 0.5, and SGD is utilized as the optimizer for updating the model parameters. The learning rate is set to 0.01 at the beginning of the FL task and reduced to 0.001 at the start of the 50th round. The experimental setup simulates a real-world environment with five IoT devices, reflecting common testbed configurations used in peer-reviewed research on FL [39, 191, 171].

Results of training the RL agent: The RL agent used is first trained and then deployed as a trained agent to generate offloading strategies during each round of FL training. To reduce the number of iterations in one round of RL training, the iteration is reduced from 100 to 5 iterations as described in Section 3.3. The training schedule for the RL agent comprises 50 rounds, and the PPO algorithm is used as the RL algorithm. The RL agent consists of two neural networks: an actor network and a critic network. Both networks share the same architecture, consisting of fully connected layers with two hidden layers, containing 64 and 32 neurons, respectively.

Table 3.6 Clustering devices into groups when using VGG5.

Devices	Training time (s)	Group number	Group center
Jetson	0.07	1	0.07
Pi4 ¹ 1.5GHz	3.58	2	3.7
Pi3 ¹ 1.2GHz	3.75	2	3.7
Pi3 ² 1.2GHz	3.77	2	3.7
Pi4 ² 0.7GHz	5.14	3	5.14

During training, a discount factor of $\gamma = 0.9$ is set for the RL agent to determine the importance of using rewards from future states [147]. The learning rates for the actor and critic networks are configured to be 10^{-4} , which is a standard value used in the literature for RL training [147]. The actor and critic networks are updated every 10 rounds. For each update, the data collected in the previous 10 rounds are used for 50 times. To ensure a balance between exploration and exploitation during training, the standard deviation of the actor network is set as 0.5 at the beginning of the RL training and exponentially decayed with a decay rate of 0.9 after 200 rounds of training. This allows the RL agent to have more freedom to explore the action space in the first 200 rounds and then decreases the standard deviation to encourage the agent to produce stable actions after 200 rounds. These hyperparameters remain consistent throughout all the experiments, ensuring better generalization of FedAdapt and consistent performance evaluation of the RL agent for various FL tasks and scenarios.

Clustering: The initial state S_0 is gathered by *Pre-processor* under the execution without any offloading. The results of the initial round are also used for calculating the group centers for the subsequent FL rounds. Table 3.6 shows the results of clustering the five devices in the testbed. The Jetson has a faster training speed due to GPU acceleration. Pi4² is the straggler due to the lower CPU frequency (0.7GHz). Using the values of the training time per iteration of each device the k-means clustering algorithm divides all devices into G groups based on the results from the first round of training. It is assumed that the training speed of devices will not change substantially in subsequent rounds. In this experiment, G is calculated using the elbow method [77] and set to $G = 3$. The Jetson and Pi4² (0.7GHz) are individually allocated to a group, whereas the Pi4¹, Pi3¹, and Pi3² are clustered into one group.

To verify the offloading actions produced by the RL agent in FedAdapt, an empirical study is conducted with VGG5 using all potential PP for each device. The results are shown in Table 3.7, and the best performance result is indicated in bold. The optimal offloading actions for each group are determined to be $[\mu_{G=1} > 0.96, \mu_{G=2} < 0.38, \mu_{G=3} < 0.38]$, and the boundaries between adjacent PPs are identified as 0.38, 0.79, and 0.96. The proportion of workload (FLOPs) on the device for each PP is calculated as 0.1, 0.66, 0.94, and 1. Therefore,

the PP closest to the action generated by the RL agent is chosen as the selected offloading strategy. The results indicate that the best performance is achieved when the layers after PP1 are offloaded to the server for all Raspberry Pis, while the Jetson performs optimally for device-native execution.

Table 3.7 Training time per iteration in seconds for each device for all possible PPs in VGG5.

PP	Jetson	Pi4 ¹ 1.5GHz	Pi3 ¹ and Pi3 ² 1.2GHz	Pi4 ² 0.7GHz
PP1	0.51 ± 0.02	2.38 ± 0.18	2.99 ± 0.06	2.63 ± 0.07
PP2	0.28 ± 0.01	3.61 ± 0.4	3.97 ± 0.04	4.68 ± 0.29
PP3	0.27 ± 0.02	5.24 ± 0.45	4.93 ± 0.05	5.88 ± 0.3
PP4 (device native)	0.17 ± 0.04	4.36 ± 0.66	4.47 ± 0.17	5.15 ± 0.1

Given the optimal offloading strategies shown in Table 3.7, to validate if the RL agent in FedAdapt can learn the optimal offloading strategies for VGG5, the action of the RL agent over 500 rounds training is shown in Figure 3.4. The results represent the average of five independent runs with different random seeds and are displayed for three different groups, G_1 , G_2 , and G_3 . The horizontal lines represent the boundaries for each PP. At the start of RL training, the RL agent produces similar offloading actions (around 0.5) for each group. However, as the RL agent optimizes the offloading actions based on rewards, the mean actions of G_1 , G_2 , and G_3 become optimal after the 80th, 30th, and 40th rounds, respectively. After these rounds, the mean actions of all three groups align with the optimal offloading actions, indicating that the RL agent successfully learns to produce efficient offloading strategies.

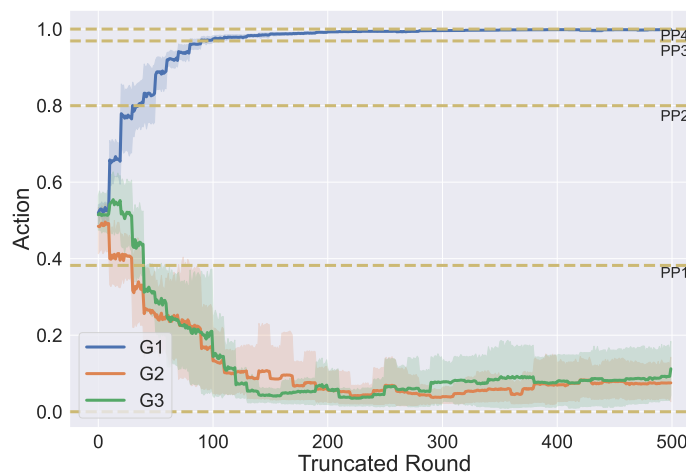


Fig. 3.4 Actions produced for each group chosen by the RL agent during training for VGG5. Action is the proportion of training workloads that remain on the devices within the group.

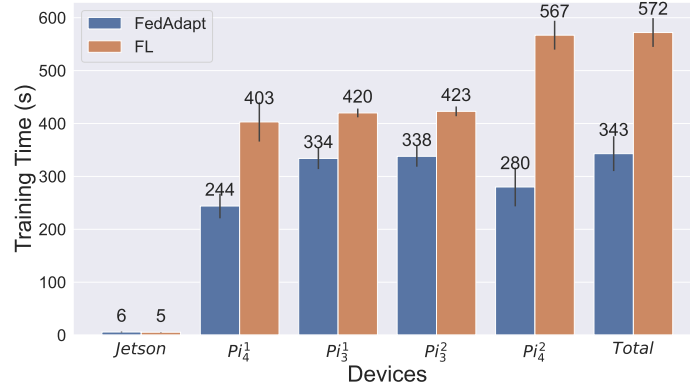


Fig. 3.5 Total training time per round in seconds for different devices in FedAdapt and classic FL using VGG5.

Once the training of the RL agent is completed, the trained actor network is deployed to guide the offloading strategies during FL training. The average training time for one round on each device using VGG5 is displayed in Figure 3.5. The training times for all Raspberry Pis are reduced and any negative impact of offloading on the Jetson is minimized. The maximum performance gain is observed for the straggler Pi4², with a 50% reduction in training time per round. Overall, FedAdapt saves 40% of the total training time for one round compared to classic FL. In this experiment, the network bandwidth remains unchanged during the FL rounds.

3.4.3 Adapting to Changing Network Bandwidth

This section evaluates FedAdapt in terms of addressing RP3, which focuses on the impact of changing network bandwidth on performance when DNN partitioning-based offloading is employed (Table 3.4). This section investigates scenarios with devices that operate under limited bandwidth conditions, such as 10 Mbps. This may lead to potential changes in the optimal PPs and offloading strategies. The experimental setup is similar to that presented in Section 3.4.2.

Clustering configuration: FedAdapt incorporates an additional grouping mechanism to address the impact of low network bandwidth on training time. This approach is similar to the clustering process described in Section 3.4.2, but it takes into account both the training time per iteration and the network bandwidth. For instance, devices with a lower bandwidth of 10 Mbps are treated as a separate group to minimize any negative effects on training time.

In this experiment, the upload bandwidth of Pi3² is manually set to 10 Mbps for the entire RL training, while other devices are connected via the 75 Mbps Wi-Fi network. Three groups ($G = 3$) are utilized. The grouping results are shown in Table 3.8 show the grouping

Table 3.8 Example of clustering into groups for five devices with one low bandwidth device when using VGG5.

Device	Training time (s)	Bandwidth	Group number	Group center (s)
Jetson	0.07	75 Mbps	1	0.07
Pi4 ¹ 1.5GHz	3.58	75 Mbps	2	4.16
Pi3 ¹ 1.2GHz	3.75	75 Mbps	2	4.16
Pi3 ² 1.2GHz	3.77	10 Mbps	3	3.7
Pi4 ² 0.7GHz	5.14	75 Mbps	2	4.16

results after the first round of training in FedAdapt. The Jetson, Pi4¹, Pi4², and Pi3¹ devices are clustered into two groups based on their training time. In addition, Pi3² is assigned to another group since its network bandwidth is much lower than other devices.

Figure 3.6 presents the actions generated by the RL agent for each group (G_1 , G_2 , and G_3) over 500 rounds during the training of VGG5.

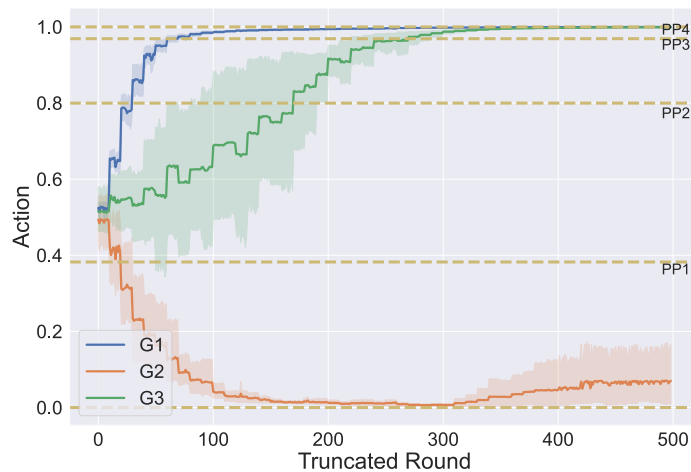


Fig. 3.6 Actions produced for each group chosen by the RL agent during training of VGG5 that accounts for devices with low network bandwidth to the server. Action is the proportion of training workloads that remain on the devices within the group.

At the initial stages of training, the RL agent rapidly learns optimal actions for G_1 and G_2 . After the 20th round, the mean action of G_1 becomes optimal, and similarly, after the 60th round, the mean action of G_2 becomes optimal. However, finding the optimal action for G_3 requires more exploration, and it is only after the 240th round that the RL agent

determines the best action for this group. This delay in optimizing the action for G_3 is because, initially, the rewards from G_1 and G_2 dominate the total reward, leading the agent to focus on optimizing the actions for those groups first. Once the actions for G_1 and G_2 become optimal, the RL agent gradually shifts its focus to optimizing the action for G_3 . As a result, the mean actions of all three groups eventually align with the optimal offloading actions after the 240th round.

3.4.4 Comparing FedAdapt and Classic FL

The performance of FedAdapt is compared to classic FL in terms of training time and accuracy. The evaluation is conducted on the same five devices used previously for FL training on the CIFAR-10 dataset over 100 rounds. For the first 50 rounds, all devices are connected with Wi-Fi, providing a network bandwidth of 75 Mbps. However, during the remaining 50 rounds, the network bandwidth is intentionally lowered to 10 Mbps for specific devices in a controlled manner. The network bandwidth for the devices will be reduced to 10 Mbps for 10 rounds, then recover to 70 Mbps. The sequence of devices is as follows: Jetson (rounds 50 to 59), Pi4¹ (rounds 60 to 69), Pi4² (rounds 70 to 79), Pi3¹ (rounds 80 to 89), and Pi3² (rounds 90 to 99).

The FedAdapt trained RL agent, trained as discussed in Section 3.4.3, is deployed to generate the offloading actions for each device during the FL rounds using VGG5. On the other hand, for classic FL, training on devices is conducted without any offloading. The comparison between FedAdapt and classic FL is then performed based on the training time and accuracy achieved during these 100 rounds among the heterogeneous devices and changing network conditions.

Figure 3.7 presents the training time of VGG5 for each round of both FedAdapt and classic FL. Up to the 50th round, FedAdapt demonstrates a significant reduction of 40% in the average training time compared to classic FL.

For the last 50 rounds, the network bandwidth varies for different devices. FedAdapt effectively responds to these changes by utilizing observations from the previous round to determine a suitable offloading strategy for the current round. As a result, FedAdapt adapts to the changing network conditions and reassigns devices to appropriate groups. For instance, since the optimal action for the Jetson is PP4 (device native), there is limited impact on training time from Round 50 to 59. However, for other devices, the change in bandwidth makes their previous offloading action invalid. However, FedAdapt promptly adjusts the offloading strategy in the subsequent round by reassigning the devices to G_3 . Overall, throughout the 100 rounds, FedAdapt manages to reduce the training time by nearly 30% compared to classic FL.

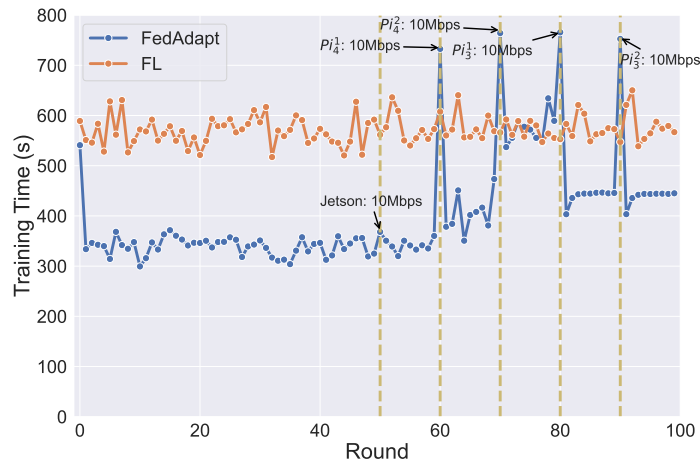


Fig. 3.7 Comparing the training time per round in FedAdapt and classic FL for VGG5. Vertical lines define five time slots after the 50th round. At the beginning of each slot, the highlighted device is limited to under 10 Mbps bandwidth.

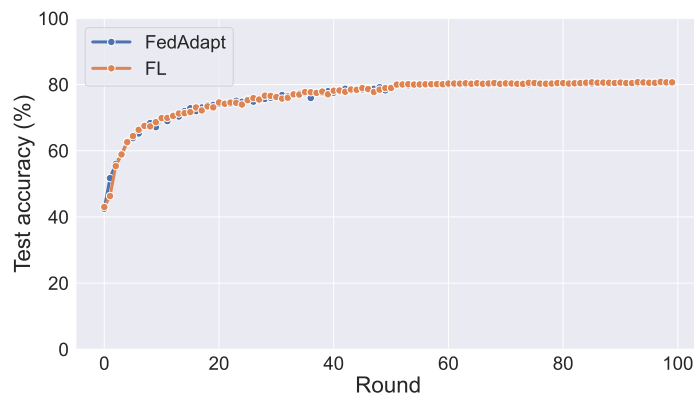


Fig. 3.8 Test accuracy per round of FedAdapt and classic FL for VGG5.

Figure 3.8 compares the test accuracy of VGG5 using both FedAdapt and classic FL over the 100 rounds. For image classification tasks, accuracy is the most commonly used metric to evaluate the quality of a model [54, 158]. Test accuracy is defined as the ratio of correctly classified images to the total number of images in the test dataset. It is observed that both approaches achieve similar accuracy since FedAdapt also utilizes the same FedAvg algorithm as classic FL, it attains the same convergence speed and final accuracy.

Regarding the overhead incurred by FedAdapt, which includes the time required to run the actor network of RL agent and the time taken to redeploy models on each device, it was measured to be an average of 1.6s per round (equivalent to 0.5% of the time for one round of training). In terms of memory consumption, the majority of the memory used by the RL agent is allocated to the actor and critic networks, which are lightweight networks. This is

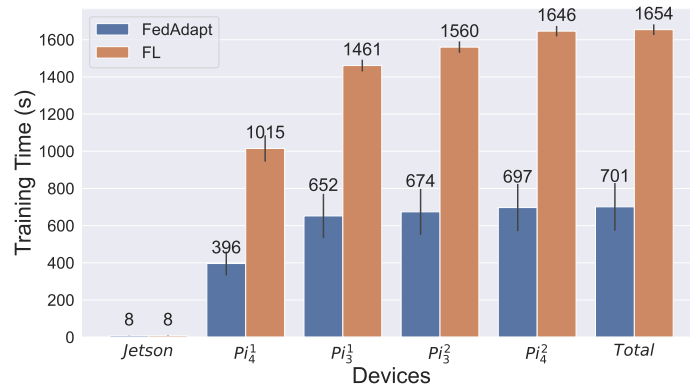


Fig. 3.9 Device and total training time per round in seconds in FedAdapt and classic FL for VGG8 when using the RL agent trained for VGG5.

significantly smaller than the VGG5 model, accounting for only 0.92% of the size of the latter.

Reusing the RL agent of FedAdapt: The performance of FedAdapt, utilizing the RL agent that was originally trained for VGG5, was evaluated on VGG8 without custom retraining. Figure 3.9 shows the average training time of one round for each device. The maximum performance gain is achieved for the straggler Pi₄², where a 57% reduction in training time per round is observed.

However, it should be noted that as the network bandwidth changes, FedAdapt generates sub-optimal offloading strategies for VGG-8 after the 70th round. The RL agent, which was originally trained for VGG5, tends to select device-native strategies instead of offloading-based strategies for devices with low bandwidth in the context of VGG8. As a result, the overall performance gain is minimized due to the differences in model architectures.

Despite this limitation, the overall training time is still reduced by nearly 40% when compared to classic FL, as shown in Figure 3.10. This highlights the effectiveness of FedAdapt even when applied to a different model without custom retraining.

3.5 Existing Solutions

This section discusses the existing solutions related to FedAdapt and highlights the differences between these approaches and FedAdapt.

Split learning: SL [169] was introduced to address the challenges of training computation-intensive DNN on resource-constrained devices. It involves partitioning the DNN into two networks: a device-side network and a server-side network. The device-side network is responsible for training the DNN up to the PP. The activation feature map of the partitioning

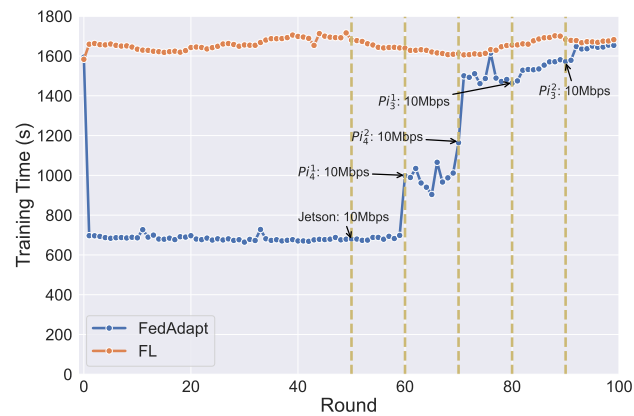


Fig. 3.10 Comparing the training time per round in FedAdapt and classic FL for VGG8 when using the RL agent trained for VGG5. Experimental setting is presented in Section 3.4.3.

layer on the device is then sent to the server, which continues training the rest of the DNN. After calculating the training loss and updating the gradient, the respective gradients are sent back to the device for further updates. When using SL with multiple devices, training occurs in a sequential round-robin fashion, where only one device connects to the server at a time, and the updated weights are then copied to the next device for continued training. This approach reduces computation on the device compared to traditional FL, where the entire DNN is trained on the device. However, SL becomes inefficient for a large number of devices due to the sequential training process.

Splitfed Learning: Recent research has explored the combination of FL and SL to overcome the limitations of each approach, giving rise to SFL [165]. SFL aims to achieve both parallel training in FL and accelerated device training in SL. Additionally, the synergy between SFL and knowledge distillation has been proposed to enhance the convergence rate of large models (e.g., ResNet56) on resource-limited devices [53]. However, existing SFL methods often lack optimal offloading strategies, and they may require hardware configuration data to manually determine the PPs for all devices before training. Moreover, the use of a static partitioning strategy may become suboptimal when operational conditions change during the training.

Computation Offloading in Edge Computing: Computation offloading has been extensively explored in the context of edge computing, where resource-constrained devices offload computationally expensive tasks to nearby servers [196]. This approach helps alleviate the computational burden on the devices. Research efforts have focused on optimizing the offloading strategy to maximize performance, such as reducing training time while minimizing energy consumption [4]. While there has been research on determining optimal offloading

Table 3.9 Comparison of FL, SL, SFL, and FedAdapt

	FL	SL	SFL	FedAdapt
Independent (parallel) training	✓	✗	✓	✓
Limited computational resources	✗	✓	✓	✓
Heterogeneous devices	✗	✗	✗	✓
Changing network bandwidth	✗	✗	✗	✓
Optimizing offloading strategy	✗	✗	✗	✓

for inference tasks, the application of computation offloading to training tasks such as FL training is still in its initial stages. Specifically, determining an adaptable and dynamic offloading strategy that can respond to changes in operational conditions is required, given that FL training is computationally intensive and requires more time than inference queries in machine learning applications.

How does FedAdapt differ from prior work? Table 3.9 presents a comparison of FL, SL, SFL, and FedAdapt. As in FL, FedAdapt independently trains on the local device and, similar to SL, accounts for the limited computational resources on the device. Although DNN partitioning-based offloading is also utilized in SFL, the key differences are that FedAdapt accounts for heterogeneous devices and adapts to the change of network bandwidth which is not considered within classic SFL. In addition, FedAdapt requires no prior knowledge of the devices but uses an automated approach based on RL to identify the optimal offloading strategy for each device. It worth noting that most FL, SL, and SFL implementations in current literature are simulation-based and do not focus on real testbeds. The benefits of FedAdapt on the other hand are demonstrated on a physical lab-based testbed.

3.6 Summary

The limitations of classic FL in IoT-edge environments, such as limited computational capacity on IoT devices, device heterogeneity, and varying network bandwidth between devices and servers, make it impractical for efficient training. To address these challenges, this chapter proposes FedAdapt, a comprehensive framework that incorporates three techniques to accelerate FL, mitigate the impact of stragglers, and adapt to varying network bandwidth.

Through FedAdapt, the training time of straggler devices is significantly reduced by more than 50% compared to classic FL. In addition, when facing both stragglers and changing network bandwidth, FedAdapt outperforms classic FL by achieving up to 40% reduction in training time while maintaining the same accuracy and convergence speed. The execution

time overhead incurred by FedAdapt is negligible, making it a practical and efficient solution for FL in IoT-edge environments.

It is worth noting that although FedAdapt employs DNN partitioning-based offloading to accelerate classic FL training, the new communication overhead of transferring intermediate results (both activation and gradient) cannot be ignored. In the worst-case scenario, the communication time can offset the time savings gained by DNN partitioning-based offloading. The next chapter will investigate this communication overhead and introduce methods to mitigate it.

Chapter 4

Reducing Communication between IoT Devices and Edge Nodes

Chapter 3 proposed using DNN partitioning-based offloading to accelerate DNN training on IoT devices. The FedAdapt framework was presented to support an adaptive offloading strategy and demonstrates the acceleration of training time. However, it is worth noting that offloading also introduces new communication overheads, specifically the communication of activation and gradient of the partitioning layer. This can be large since the data (i.e., activation and gradient) needs to be frequently transferred during training. As a result, communication becomes a new bottleneck for training time when the network bandwidth is low (e.g., 3G).

This chapter proposes techniques to reduce the communication overhead incurred, particularly in DNN partitioning-based offloading approaches, to further boost speedup. EcoFed is introduced as a communication-efficient DNN partitioning-based framework. EcoFed significantly reduces the communication incurred by DNN partitioning-based offloading by considering both the reduction of volume and frequency for activation and gradient. In particular, EcoFed eliminates the transmission of the gradient by developing pre-trained initialization of the DNN model on the device for the first time. In addition, EcoFed incorporates a novel replay buffer technique and implements a quantization-based compression to reduce the transmission of the activation.

Experiments on diverse datasets and models demonstrated that EcoFed can reduce the communication cost by up to $133\times$ and accelerate training by up to $21\times$ when compared to classic FL. Compared to vanilla DNN partitioning-based FL, EcoFed achieves a $16\times$ communication reduction and $2.86\times$ training time speedup.

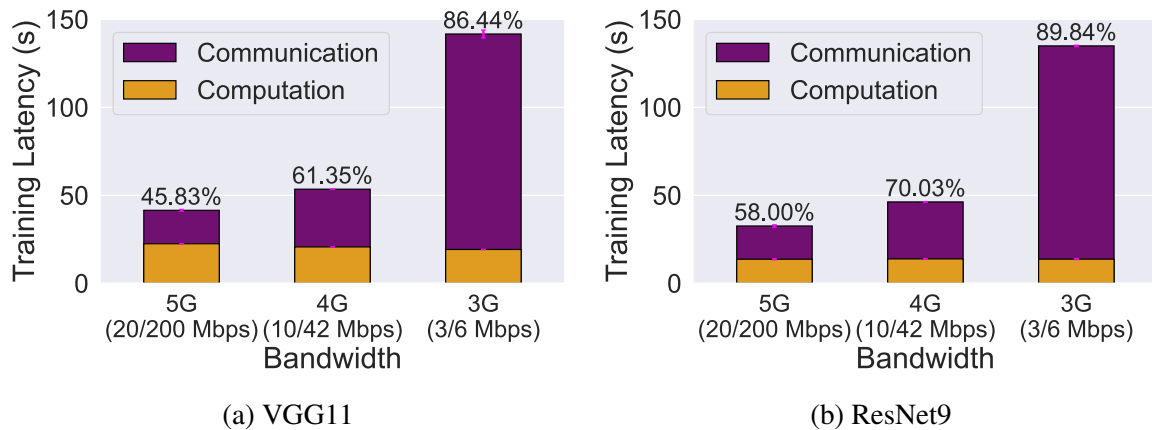


Fig. 4.1 Computation and communication time in DPFL training under typical (upload/download) network bandwidth. Numerical value above the bars is the percentage of communication time.

4.1 Motivation

FL training on resource-constrained IoT devices poses significant challenges, especially when training DNNs that are computationally intensive, as the burden of training the entire DNN is on the device [66, 119, 39, 171, 184]. To address this challenge, *DPFL* has been developed as a solution [177, 47, 53] to reduce the computation burden on IoT devices.

In DPFL, the DNN model is partitioned into two parts: a device-side model and a server-side model. The initial layers of the DNN are deployed and trained on the device, while the remaining layers are offloaded to a server with more computational resources. Chapter 3 verifies that these partitioning and offloading techniques can significantly reduce the training time on resource-constrained devices as it only needs to train a subset of the entire DNN model.

Huge communication cost in a vanilla DPFL system. Although DPFL reduces the training time on devices, it introduces additional communication overheads between the device and the server. This is because during training, the outputs of the activation generated by the device-side model in a forward pass, and the corresponding gradient calculated during backward propagation, need to be transferred between the devices and the server. The communication overheads of vanilla DPFL systems are critical for the following three reasons:

(i) *Communication time becomes a new bottleneck in the DPFL system.* Figure 4.1 shows the computation and communication latency incurred with DPFL training for the PP with minimum latency¹. The communication overhead in DPFL is nearly 46% (58% for ResNet9)

¹Please refer to Section 4.5 for the experiment configuration

of the overall training time under 5G conditions and around 86% (90% for ResNet9) for 3G bandwidth;

(ii) *The volume of communication is substantial.* The total volume of communication in a DPFL system is not negligible since it is proportional to the sum of all training samples across devices. Moreover, current DNN training uses stochastic gradient descent, which continuously iterates through the dataset to compute gradients and update the model. This results in the repeated traversal of the training dataset. In a DPFL system, this incurs additional communication costs in each iteration for transferring activation and gradient.

(iii) *The communication frequency is extremely high.* Traditional FL only requires communication twice per round, but DPFL requires communication twice for every batch of sample inference. This works well in a controlled and stable network setting. However, in real-world environments where network conditions can vary, such high communication demands may become impractical, leading to potential performance issues due to unstable connections or limited bandwidth.

Limitations of Local loss-based DPFL. Recent DPFL methods [47, 53], which are denoted as *local loss-based DPFL*, use local loss generated by an auxiliary network to train the device-side model instead of transferring and using the gradient from the server. Local loss-based DPFL can reduce half of the communication cost since only the activation needs to be sent from the devices to the server. However, these methods are ineffective due to two issues (Section 4.5).

Firstly, *the volume of communication required to achieve the target accuracy increases due to poor learning performance, as indicated by lower final accuracy and convergence speed.* The poor learning performance of local loss-based DPFL methods is referred to as ‘accuracy degradation’ and is caused by training using local loss instead of end-to-end training. It is also demonstrated that the use of local error signals will lower the final accuracy and convergence speed. As a result, the communication cost to achieve the target accuracy is similar to or even higher than vanilla DPFL.

Secondly, *under low network bandwidth conditions, the training time does not significantly decrease since the upload bandwidth is usually lower than the download bandwidth.* Local loss-based DPFL methods do not consider the transfer of activation, which is significant - half of the communication volume - and has a high frequency per iteration. Therefore, when network bandwidth is limited, as typically seen in resource-constrained environments, it is not feasible to accelerate training using current local loss-based DPFL methods.

To reduce the communication in the DPFL system, this chapter proposes EcoFed, a communication-efficient framework for DPFL systems. Figure 4.2 illustrates the training pipeline of classic FL, vanilla DPFL, local loss-based DPFL, and EcoFed. EcoFed signif-

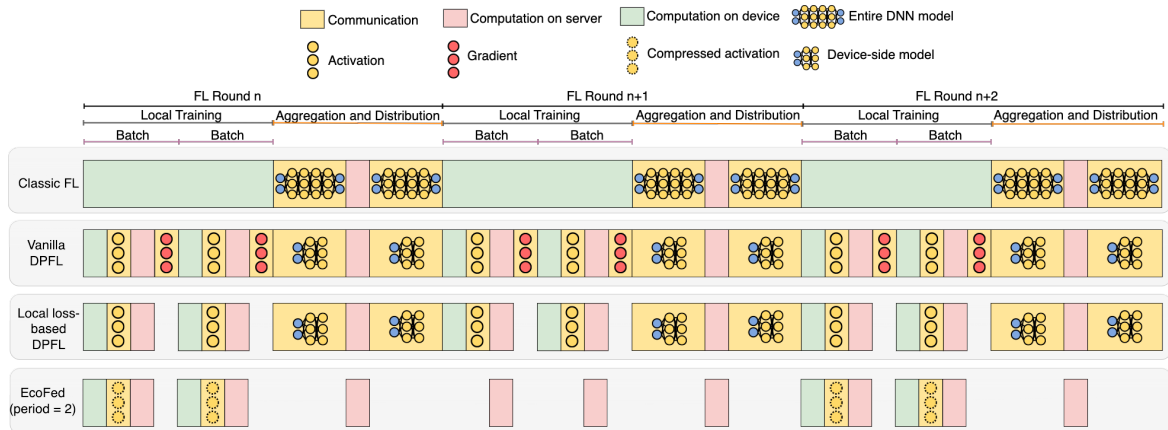


Fig. 4.2 The training pipeline of classic FL, vanilla DPFL, local loss-based DPFL, and EcoFed for three rounds of training. Classic FL transfers the entire model from the devices to the server at the end of each round. Vanilla DPFL only needs to upload the device-side model at the end of each round. However, vanilla DPFL transfers the activation and gradient for each batch sample. Local loss-based DPFL reduces the communication by half since the gradient is computed locally. EcoFed reduces communication further as it transfers the activation only periodically (for example, once in two rounds) and further compresses the size of the activation.

icantly reduces the communication cost (shown as yellow blocks in Figure 4.2) by only transferring the activation periodically (for example, once every two rounds) and further reduces the size of the activation. Pre-trained initialization of the device-side model is employed in EcoFed to reduce accuracy degradation caused by local loss-based methods. The frequency and size of transferring activation are reduced by proposing a replay buffer with a lightweight quantization technique. The next section will provide an overview of EcoFed and how it is integrated into the DPFL system.

4.2 EcoFed Overview

This section presents an overview of EcoFed and how it is incorporated into the DPFL system. Figure 4.3 provides an overview of the modules in the EcoFed framework, which operates across resource-constrained devices and servers (cloud or edge servers). A brief description of these modules is discussed below.

1. *Initializer*: This module is first configured on the server and, at the start of FL training, it is sent to each device. It determines the initial training scheme, configures the DNN models, and initializes the model weights.

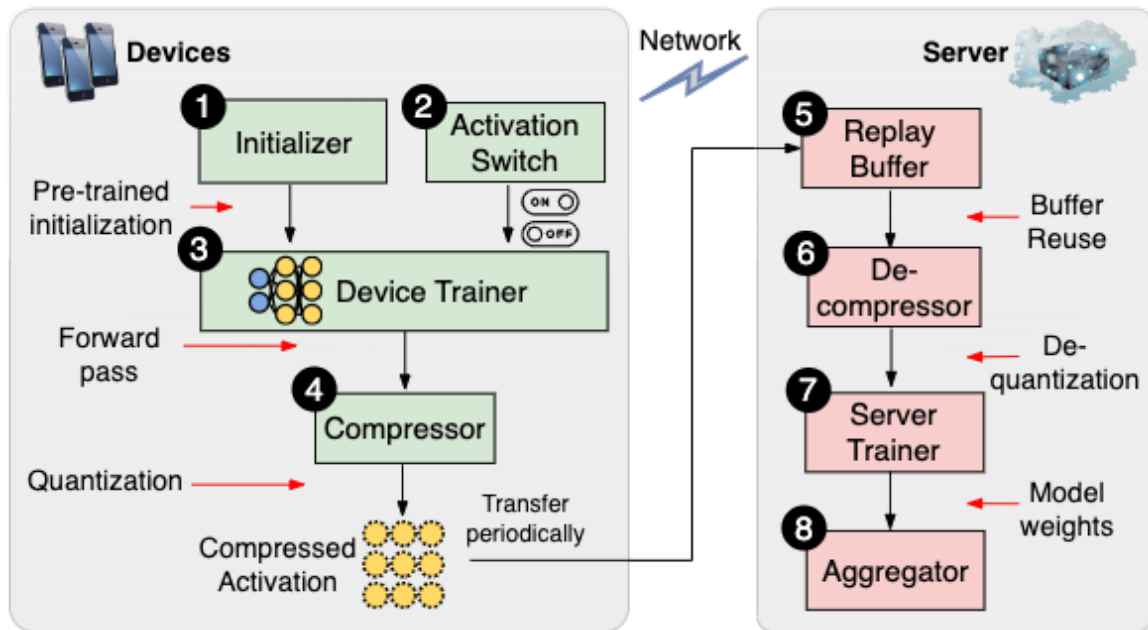


Fig. 4.3 EcoFed modules on the devices and server.

2. *Activation Switch:* The *Activation Switch* on the device side decides whether the activation outputs of the device-side model need to be sent to the server.
3. *Device Trainer:* This is the training engine that resides on devices.
4. *Compressor:* The *Compressor* module is located on the device and compresses the activation outputs before sending them to the server.
5. *Replay Buffer:* The *Replay Buffer* on the server is the cached activation so that devices can reduce the frequency of sending activation to the server.
6. *Decompressor:* This module operates on the server-side and decompresses the compressed activation.
7. *Server Trainer:* The *Server Trainer* is the training engine on the server.
8. *Aggregator:* The *Aggregator* module collects the updated weights from each device and aggregates them using the FedAvg algorithm [111].

At the beginning of a DPFL training, the *Initializer* (1) determines the training configurations (the configurations of the DNN models and partitioning deployments) and initializes the weights. The device-side models are sent to the devices and the corresponding server-side

models are kept on the server. Both vanilla DPFL and EcoFed require a similar initialization phase. However, the training processes differ, as outlined below.

In vanilla DPFL, after initialization, the training starts on the device-side model independently for each device. The *Device Trainer* (③) will first generate activation outputs of the device-side model. The outputs and labels of the corresponding data samples are sent to the server. The *Server Trainer* (⑦) trains the server-side model using the activation outputs received from the device and generates corresponding gradients. The gradients are sent back to each device to update the device-side model. The above steps are repeated for each batch sample in vanilla DPFL training.

In contrast, in EcoFed, before generating and sending the activation outputs of the device-side model to the server, the *Activation Switch* (②) will determine whether the outputs of the device-side model are required to be sent to the server or if the server should use the buffer with the cached activation to train the server-side model. This is accomplished in a period-controlled manner in EcoFed, as detailed in Section 4.3. If the activation outputs are required to be sent, then they will be further compressed using the quantization technique implemented by the *Compressor* (④). The compressed activation and labels of the corresponding samples will then be sent to the server. On the server, the compressed data will be firstly used to update the *Replay Buffer* (⑤) and reconstructed by the *Decompressor* (⑥) for training the server-side models. The *Server Trainer* only needs to calculate and update the gradients of the server-side models without sending the gradient back to each device for training the device-side models.

After completing the above steps, the *Aggregator* (⑧) will collect updated weights from each device for aggregation and for generating new global weights for next round using the FedAvg algorithm [111].

In summary, the DPFL training in EcoFed differs from vanilla DPFL training in the communication of activations and gradients. EcoFed eliminates the need for transferring gradients from the server to devices by employing the *Initializer* module. Regarding activation upload, EcoFed utilizes the *Activation switch* module to control communication frequency and the *Compressor* module to further reduce data size. By integrating these modules, EcoFed achieves a communication-efficient DPFL system that has lower communication overheads.

4.3 Technical Design

This section presents two detailed techniques that support the implementation of modules in EcoFed: pre-trained initialization and replay buffer.

4.3.1 Pre-trained Initialization

The current local loss-based DPFL approach incorporates local error signals to train the device-side model using an auxiliary network [47, 53, 48]. The benefit of this approach is that it eliminates the need for transferring gradients from the server to the devices. However, the decoupling of the device-side model and the server-side model, each trained by different error signals (local and global error signals), adversely affects accuracy and convergence speed. Section 4.5 empirically demonstrates this by showing higher loss in local loss-based DPFL approaches. As a result, the reduction in communication required to achieve the target accuracy is not effectively achieved and may even increase due to the lower accuracy and slower convergence speed of local loss-based methods (Section 4.5).

To address this challenge, EcoFed proposes a solution that adopts pre-trained initialization for the device-side model, denoted as \mathbf{w}_C . This means initializing the device-side models with pre-trained weights that represent partial weights of the entire model previously trained on a large dataset, such as ImageNet [27]. Furthermore, to investigate the impact of local loss on generating activation outputs used to train the server-side model, empirical studies are conducted to show the impact of pre-trained initialization (Section 4.5). The results indicate that local loss alone is not sufficient to determine a “good” activation output for training the server-side model, especially when pre-trained initialization is utilized. Therefore, during FL training with EcoFed, the weights of the device-side model on device k , denoted as $\mathbf{w}_{C,k}$, are frozen. The insight of pre-trained initialization in EcoFed is that the device-side model handles the initial layers of the overall model, which learn general features that are applicable to a range of tasks. This aligns with transfer learning, in which the initial layers are task-agnostic, and the weights of a pre-trained model can be effectively transferred to another model [188]. In transfer learning, the initial layers, which extract patterns such as edges, textures, or simple shapes in the case of images, are considered task-agnostic. They are not specific to the task the model was originally trained for and can be used on new tasks that may have different objectives. For instance, a model pre-trained on a large dataset like ImageNet can be used for another domain, such as object detection, by reusing the initial layers and retraining only the later layers for the new task [188]. In EcoFed, this transfer of weights is applied to initialize the device-side model, enhancing the training process.

Benefits of pre-trained initialization in EcoFed:

1. *Reduced communication during training:* By adopting pre-trained initialization and freezing the weights of the device-side model $\mathbf{w}_{C,k}$, the need for devices to receive gradients from the edge server is eliminated.

2. *Mitigated accuracy degradation using local error signals:* The use of local error signals in local loss-based DPFL approaches can negatively impact accuracy and convergence speed. However, with pre-trained initialization, the device-side model $\mathbf{w}_{C,k}$ starts with knowledge gained from a larger dataset (e.g., ImageNet), leading to reduced accuracy loss compared to using local error signals alone.
3. *Reduced computational workload on resource-constrained devices:* Pre-trained initialization allows the gradients of $\mathbf{w}_{C,k}$ to remain frozen on the devices. As a result, devices with limited computational capabilities do not need to calculate and update these gradients during the training process, alleviating the computational burden on such resource-constrained devices.
4. *Staleness of cached activation:* This refers to when the activation outputs that are stored in the cache do not accurately reflect the current state of the model parameters since they were updated. Since the device-side model $\mathbf{w}_{C,k}$ and its corresponding activation outputs remain relatively unchanged during each training round, it becomes feasible to use cached activation from a buffer. The persistence in activation facilitates more efficient training of the server-side models, as the cached activation can be utilized effectively without frequent updates.

In summary, pre-trained initialization and freezing of the weights of the device-side model offer multiple advantages in EcoFed, including reduced communication, improved accuracy, lower computational demands on devices, and the ability to leverage cached activation for more efficient server-side model training.

4.3.2 Replay Buffer

In EcoFed, a novel module called the *Replay Buffer* is introduced on the edge server. This buffer stores activation outputs obtained from a previous training round. During the current training round, the server utilizes these cached activation outputs to train the server-side model, thereby reducing the need for transmission of activation from the devices to the server.

The buffer is periodically updated with the new activation, ensuring that the stored information remains up-to-date. When the transmission of activation from devices to the server is turned off in a particular round, EcoFed switches to using the cached activation from the buffer. As a result, two distinct modes of training are implemented: one with the buffer, where cached activation is utilized, and another without the buffer, where activation is transmitted from devices in the current round.

Periodic transfer in Activation Switch: The *Activation Switch* module in EcoFed controls the frequency of transferring activation outputs from the device-side to the server-side model. The transfer frequency is determined by an interval denoted as ρ . For instance, if $\rho = 2$, it means that activation outputs are transferred every second round, while during the other rounds, the edge server uses cached activation from the buffer to train the server-side models. It is worth noting that, when the devices participate in training, their activation outputs (\mathbf{a}_k^t) will vary in each round due to different data batches and data augmentations. Thus, the buffer on the edge server needs to be periodically updated during the training.

The hyper-parameter ρ plays a crucial role in EcoFed as it affects both model performance and communication cost. A higher value of ρ reduces the frequency of activation transfers, leading to decreased communication between devices and the server. However, this may also impact the learning performance of the model, as the server-side models will be trained using cached activation that might be stale. The trade-off between communication cost and model performance due to different values of ρ is further discussed in Section 4.5.

Reducing buffer size using compression: To address the potential issue of a large buffer requirement due to the storage of activation, EcoFed implements strategies to efficiently reduce the buffer's storage. The maximum size of the buffer required will be the size of all activations transferred from all devices. However, it is practical to establish a maximum buffer size. Therefore, EcoFed sets a practical maximum buffer size determined based on factors such as available storage resources and implements periodic updates to the buffer to efficiently manage the size. In addition, instead of storing the original activation (\mathbf{a}_k^t), EcoFed further reduces the buffer size by storing compressed activation outputs (\mathbf{z}_k^t). The compression is performed by the *Compressor* module of EcoFed using a lightweight 8-bit linear quantization technique [176], represented as function $Q(\cdot)$. The original activation, which is typically represented using 32 bits, is quantized to 8 bits before being sent to the server. This quantization reduces the size of each activation, effectively reducing the memory footprint required by the *Replay Buffer*.

Computational complexity of the buffer replay: The computational complexity of using the buffer replay in EcoFed can be approximated as $O(n)$, where n represents the number of activations retrieved from the buffer. By employing 8-bit linear quantization to compress activations, EcoFed reduces buffer storage, enabling efficient memory management. Compression minimizes the overhead associated with storing large volumes of activations while adding only limited overheads to decompression during replay. Setting a maximum buffer size in EcoFed can also optimize memory usage and prevent overloads caused by an increasing number of connected devices. This can improve buffer scalability over the current implementation and is discussed in Chapter 7, Section 7.2.1.

4.3.3 EcoFed Training Pipeline

Algorithm 1: DNN partitioning-based training in EcoFed

```

1 Input: Pre-trained  $\mathbf{w}_C^*$  and data  $\{D_k\}_{k=1}^K$ 
2 Output:  $\mathbf{w}^*$ 
3 for each device  $k \in K$  in parallel do
4   | Download  $\mathbf{w}_C^*$  to device  $k$ ;
5   | Initialize and freeze  $\mathbf{w}_{C,k}^*$ ; //Pre-trained initialization
6 end
7 for each device and corresponding worker  $k \in K$  in parallel do
8   | Initialize  $\mathbf{w}_{S,k}$  on the server;
9   for each round  $t \in T$  do
10    | if Activation switch is on ( $t \bmod \rho == 0$ ) then
11      | Forward on  $\mathbf{w}_{C,k}^*$  to get  $\mathbf{a}_k^t$ ;
12      |  $\mathbf{z}_k^t \leftarrow Q(\mathbf{a}_k^t)$ ; //Quantization
13      | Send  $\mathbf{z}_k^t$  to the server;
14      | update buffer with  $\mathbf{z}_k^t$ ;
15    end
16    else
17      |  $\mathbf{z}_k^t \leftarrow$  buffer; //Replay Buffer
18    end
19    |  $\hat{\mathbf{a}}_k^t \leftarrow Q^{-1}(\mathbf{z}_k^t)$ ; //Dequantization
20    | Forward on  $\mathbf{w}_{S,k}^t$ ;
21    | Calculate loss  $\ell_{S,k}^t$  and gradients  $\nabla \ell_{S,k}^t(\mathbf{w}_{S,k}^t)$ ;
22    |  $\mathbf{w}_{S,k}^t \leftarrow \mathbf{w}_{S,k}^t - \eta \nabla \ell_{S,k}^t(\mathbf{w}_{S,k}^t)$ ;
23  end
24  |  $\mathbf{w}_{S,k} \leftarrow \mathbf{w}_{S,k}^t$ ;
25  |  $\mathbf{w}_S \leftarrow \sum_{k=1}^K \frac{|D_k|}{|D|} \mathbf{w}_{S,k}$ ; //FedAvg
26 end
27  $\mathbf{w}_S^* \leftarrow \mathbf{w}_S$ ;
28  $\mathbf{w}^* \leftarrow \{\mathbf{w}_C^*, \mathbf{w}_S^*\}$ ; //Concatenation of  $\mathbf{w}_C^*$  and  $\mathbf{w}_S^*$ 
29 return  $\mathbf{w}^*$ 

```

Algorithm 1 presents the DNN partitioning-based training process integrated with EcoFed, specifically utilizing pre-trained initialization and a replay buffer with quantization.

The set of K devices is denoted as $\{k\}_{k=1}^K$. Each device generates its data D_k , and the combined data from all devices is denoted as D : $\{D_k\}_{k=1}^K$. The number of samples in D_k is represented as $|D_k|$, and the total number of samples is $|D|$. The entire model is denoted as \mathbf{w} , which is partitioned into the device-side model (\mathbf{w}_C) and server-side model (\mathbf{w}_S). The models for the k^{th} device are represented as $\mathbf{w}_{C,k}$ and $\mathbf{w}_{S,k}$. The superscript t denotes the training

round t out of a total of T rounds. \mathbf{a}_k represents the intermediate activation generated by $\mathbf{w}_{C,k}$, and $\ell_{S,k}(\cdot)$ is the loss function of the server-side model of the k^{th} device.

The training process with EcoFed starts by preparing the pre-trained weights of the device-side model (\mathbf{w}_C^*), which are then frozen to avoid transferring gradient from the server-side to the device-side model \mathbf{w}_C (Lines 4 and 5). Subsequently, each server-side model ($\mathbf{w}_{S,k}$) is independently trained with K parallel workers.

In each round t , if the activation transfer is switched on (i.e., $t \bmod \rho == 0$), the activation output of $\mathbf{w}_{C,k}^*$, denoted as \mathbf{a}_k^t , is generated and compressed to 8 bits, represented as \mathbf{z}_t^k . This compressed activation is then uploaded to the server and used to update the corresponding buffer (Lines 10-15). Otherwise, the server retrieves \mathbf{z}_t^k directly from the buffer (Line 17). EcoFed dequantizes \mathbf{z}_t^k to obtain $\hat{\mathbf{a}}_k^t$ and provides this output to the server-side model for the training of $\mathbf{w}_{S,k}$ (Lines 19-22).

After each edge server completes a training round, it sends its respective $\mathbf{w}_{S,k}$ to the cloud, where the cloud aggregates a global server-side model (Lines 24-26). Finally, after T rounds of training, the cloud obtains the optimal server-side model (\mathbf{w}_S^*) and concatenates it with \mathbf{w}_C^* to create the complete model \mathbf{w}^* (Lines 28-29). Table 4.1 lists the notations used in this chapter.

4.4 Convergence and Cost Analysis

This section conducts a theoretical analysis of EcoFed focusing on two main aspects: convergence behavior and computation and communication costs on the device, comparing it to other relevant methods, namely classic FL, vanilla DPFL, and local loss-based DPFL.

4.4.1 Convergence Analysis

This section analyzes the convergence of EcoFed (Algorithm 1) following the convergence analysis presented in the literature [47, 9, 172]. Several assumptions to facilitate the analysis:

Assumption 1 – *The server-side objective functions are L -smooth, i.e., $\|\nabla F_S(\mathbf{u}) - \nabla F_S(\mathbf{v})\| \leq L\|\mathbf{u} - \mathbf{v}\|, \forall \mathbf{u}, \forall \mathbf{v}$.*

Assumption 2 – *The squared norm of the stochastic gradient has an upper bound for the server-side object function, i.e., $\|\nabla F_S(\mathbf{a}_k^t; \mathbf{w}_{S,k}^t)\|^2 \leq G, \forall k, \forall t$.*

Assumption 3 – *The learning rate η_t satisfies $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$.*

Assumption 1 to Assumption 3 are the standard assumptions widely used in FL literature that can be applied when training DNN models. [47, 9, 172]. To analyze the impact of *Replay Buffer* and quantization on the convergence, two types of errors are further

Table 4.1 Notation used in EcoFed

Notation	Description
K	The total number of devices participating in the training
k	Index of the k -th device, where $k \in \{1, 2, \dots, K\}$
D_k	The dataset generated on the k -th device
D	The combined dataset from all devices, i.e., $D: \{D_k\}_{k=1}^K$
$ D_k $	The number of data samples on the k -th device
$ D $	The total number of data samples from all devices
\mathbf{w}	The entire model to be trained
\mathbf{w}_C	The device-side model
\mathbf{w}_S	The server-side model
$\mathbf{w}_{C,k}$	The device-side model on the k -th device
$\mathbf{w}_{S,k}$	The server-side model on the k -th device
\mathbf{a}_k	The intermediate activation generated by the device-side model on the k -th device
$\ell_{S,k}(\cdot)$	The loss function of the server-side model on the k -th device
\mathbf{z}_t^k	The compressed activation from the k -th device at training round t
$\hat{\mathbf{a}}_k^t$	The dequantized activation from the k -th device at training round t
t	The index of the training round
T	The total number of training rounds
ρ	The frequency of buffer updates
δ_k	Buffer error for the k -th device
ε_k	Quantization error for the k -th device
H_1	Upper bound on buffer error
H_2	Upper bound on quantization error
η_t	Learning rate at round t
$F_S(\cdot)$	The server-side objective function
G	Upper bound on the squared norm of the stochastic gradient
L	Smoothness constant of the server-side objective function
Γ_T	Sum of learning rates, i.e., $\Gamma_T = \sum_{t=0}^{T-1} \eta_t$
$p_k^t(\mathbf{a})$	Original distribution of activations at round t for the k -th device
$q_k^t(\mathbf{a})$	Buffer distribution of activations at round t for the k -th device
d_k^t	Distance between activation distributions $p_k^t(\mathbf{a})$ and $p_k^*(\mathbf{a})$

defined: buffer and quantization errors. Buffer error is defined as the distance of gradients between the original distribution $p_k^t(\mathbf{a})$ and buffer distribution $q_k^t(\mathbf{a})$, denoted as $\delta_k \triangleq \int \|\nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S)\| \|(q_k^t(\mathbf{a}) - p_k^t(\mathbf{a}))\| da$. The $\ell(\cdot)$ is the loss function (e.g. cross-entropy loss for classification). Quantization error is defined as $\int \|\nabla \ell(\hat{\mathbf{a}}_k^t; \mathbf{w}_S) - \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S)\| \|q_k^t(\mathbf{a})\| da$, which is the sum of gradient errors over $q_k^t(\mathbf{a})$ due to quantization. A further assumption specific to EcoFed is as follows:

Assumption 4 – The buffer error and the quantization error have upper bounds, i.e., $\delta_k^t \leq H_1$ and $\epsilon_k^t \leq H_2, \forall k, \forall t$.

Convergence of device-side model: Since \mathbf{w}_C is fixed during training, its convergence is not considered.

Convergence of server-side model: Let $\Gamma_T \triangleq \sum_{t=0}^{T-1} \eta_t$. Following on from Assumption 1, Assumption 2, and Assumption 4, Algorithm 2 ensures the following:

$$\begin{aligned} \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \eta_t \mathbb{E}[\|\nabla F_S(\mathbf{w}_S^t)\|^2] &\leq \frac{4(F_S(\mathbf{w}_S^0) - F_S(\mathbf{w}_S^*))}{3\Gamma_T} \\ &+ \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \left(\eta_t (\sqrt{G} + 1)(H_1 + H_2) + \frac{L}{2} \eta_t^2 G \right) \end{aligned} \quad (4.1)$$

where $\nabla F_S(\mathbf{w}_S^t) \triangleq \frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t)$. $F_{S,k}(\cdot)$ is the objective function of the k^{th} server-side model. \mathbf{w}_S^0 is the initial server-side weights and \mathbf{w}_S^* is the optimal server-side weights. Based on Assumption 3, it is noted that with increasing T , the right-hand side of Equation 4.1 converges to zero. Thus, Equation 4.1 guarantees that the proposed algorithm of EcoFed converges to a stationary point with increasing T . The complete proof is provided in Appendix A.

Differences between the convergence of local loss-based DPFL and EcoFed: Equation 4.1 is similar to the convergence analysis presented in the literature [47]. The server-side model converges as follows:

$$\begin{aligned} \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \eta_t \mathbb{E}[\|\nabla F_S(\mathbf{w}_S^t)\|^2] &\leq \frac{4(F_S(\mathbf{w}_S^0) - F_S(\mathbf{w}_S^*))}{3\Gamma_T} \\ &+ G \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \left(\eta_t \frac{1}{K} \sum_{k=1}^K (d_k^t) + \frac{L}{2} \eta_t^2 \right) \end{aligned} \quad (4.2)$$

where $d_k^t \triangleq \int \|p_k^t(\mathbf{a}) - p_k^*(\mathbf{a})\| d\mathbf{a}$ which is defined as the distance between the probability distribution of activation \mathbf{a}_k^t and \mathbf{a}_k^* . It is worth noting that $p_k^t(\mathbf{a})$ keeps changing during FL training since $\mathbf{w}_{C,k}^t$ is updated by the local error signals. Therefore, in local loss-based DPFL, the changing distance (d_k^t) of the probability distribution of \mathbf{a}_k^t caused by updating $\mathbf{w}_{C,k}^t$ affects the convergence of the server-side model as shown in Equation 4.2. However, in EcoFed, $d_k^t = 0$ since $\mathbf{w}_{C,k}^t$ is fixed during training.

Impact of ρ and quantization error on convergence: The frequency of buffer updates, ρ , and quantization error influence convergence by controlling the buffer and the errors introduced by quantization (δ_k^t and ϵ_k^t), which are bounded by constants H_1 and H_2 . A higher

Table 4.2 Computation and communication costs on the device for each round.

Methods	Computation	Communication
FL	$ \mathbf{w} $	$2 \mathbf{w} $
Vanilla DPFL	$ \mathbf{w}_C $	$2 \mathbf{w}_C + 2 D_k \mathbf{a}_k $
Local loss-based DPFL	$ \mathbf{w}_C $	$2 \mathbf{w}_C + D_k \mathbf{a}_k $
EcoFed w/o buffer	$\frac{1}{2} \mathbf{w}_C $	$ D_k \mathbf{z}_k $
EcoFed w buffer	0	0

ρ (i.e., more frequent updates) reduces the value of H_1 , which results in faster convergence but increases the cost of communication. Conversely, a lower ρ reduces communication costs but allows errors to accumulate, which will slow convergence. Furthermore, the quantization techniques employed by the **Compressor** module affect the value of H_2 , where a larger H_2 results in slower convergence of EcoFed.

4.4.2 Cost Analysis

Table 4.2 presents a comparison of the computation and communication costs of EcoFed against classic FL, vanilla DPFL, and local loss-based DPFL. The notation $|\cdot|$ represents either the computation or communication workload of a given model or activation.

In classic FL, the entire model \mathbf{w} is computed on each device ($|\mathbf{w}|$). At the end of each round, \mathbf{w} is uploaded to the cloud, and then the newly aggregated \mathbf{w} is downloaded to the device ($2|\mathbf{w}|$).

For vanilla DPFL, each device only needs to train the device-side model \mathbf{w}_C ($|\mathbf{w}_C|$). \mathbf{w}_C is transferred at the end of each round ($2|\mathbf{w}_C|$ for both uploading and downloading). In addition, the activation and gradient of each data sample are communicated with an overhead of $2|D_k||\mathbf{a}_k|$.

Similarly, for local loss-based DPFL, each device trains \mathbf{w}_C ($|\mathbf{w}_C|$). \mathbf{w}_C is uploaded and downloaded at the end of each round ($2|\mathbf{w}_C|$), but only the activation is transferred during each round's training ($|D_k||\mathbf{a}_k|$).

However, for EcoFed without using the replay buffer (indicated as EcoFed w/o buffer), the device only computes the forward pass on \mathbf{w}_C ($\frac{1}{2}|\mathbf{w}_C|$). Additionally, the communication overhead is reduced to $|D_k||\mathbf{z}_k|$, where \mathbf{z}_k represents the compressed activation². When using the buffer in a given training round, there is no computation on device or communication between the device and server.

²There is one exception in the first round of training, devices need to download \mathbf{w}_C . Therefore, the communication cost for the first round is $|\mathbf{w}_C| + |D_k||\mathbf{z}_k|$.

In summary, the convergence of EcoFed is bounded by H_1 and H_2 , representing the upper bounds of the *Replay Buffer* and quantization error as shown in Equation 4.1. These bounds are influenced by different ρ values and quantization techniques used by the *Compressor* module. Regarding the device-side cost of EcoFed, when the buffer is not used, the device only needs to compute the forward pass of \mathbf{w}_C , and the communication cost is reduced to only transferring the compressed activation $|D_k||\mathbf{z}_k|$. When using the buffer, there are no computation or communication costs during training.

4.5 Evaluation

This section evaluates EcoFed by comparing it against four baselines, using two sets of metrics. It is demonstrated that EcoFed can achieve higher accuracy compared to the baseline methods and mitigates the accuracy degradation caused by local error signals in local loss-based DPFL. In addition, EcoFed significantly reduces the amount of data transmitted during the training process and accelerates the training.

4.5.1 Test Environment

Datasets and Models: The evaluation utilizes two image classification datasets, CIFAR-10 [79] and CIFAR-100 [79]. For data partitioning, a similar method is followed as in the literature [111]. For an independent and identically distributed (I.I.D.) setting, the training set is initially divided into 500 shards for CIFAR-10 and 5000 shards for CIFAR-100. Each device is then randomly assigned 5 shards for CIFAR-10 and 50 shards for CIFAR-100. In contrast to the dataset utilized in Chapter 3, we now incorporate a more complex dataset, CIFAR-100, for further analysis. CIFAR-100 has 100 classes instead of the 10 in CIFAR-10. The total number of training samples are the same for both CIFAR-100 and CIFAR-10. Since these training samples are distributed across all 100 classes, each class has fewer samples than in CIFAR-10. In terms of the non-independent and identically distributed (Non-I.I.D.) setting, the dataset is first sorted based on their labels, dividing it into 500 shards for CIFAR-10 and 5000 shards for CIFAR-100. Then, each device is randomly allocated to 5 shards and 50 shards for CIFAR-10 and CIFAR-100, respectively. As a result, each device only has non-I.I.D. training samples with up to half of all available classes. The test dataset remains on the server and is used to evaluate model performance after each training round.

Two popular convolutional neural networks are trained, VGG11 [158] (a plain convolutional neural network) and ResNet9 [54] (a residual convolutional neural network). In this chapter, a deeper VGG model, namely VGG11, is selected as the benchmark model

Table 4.3 Models used for evaluation. Convolution layers denoted as C followed by the no. of filters; filter size is 3×3 for all convolution layers except for downsampling convolution (filter size is 1×1); MaxPooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers, a max pooling, and downsampling convolution layers; number followed is no. of output channels.

Model	Device	Server
VGG11	C64-MP-C128-MP	C256-C256-MP-C512-C512-MP-C512-C512-FC4096-FC4096-FC10
ResNet9	C64-MP-C128-MP	RB256-RB512-RB512-FC10
Auxiliary Network	FC10	N/A

compared to the VGG5 and VGG8 models in Chapter 3. This decision is driven by the utilization of a cloud-simulated testbed, enabling us to assess the accuracy of a deeper model. Moreover, for measuring communication and training latency, we solely need to report the results for one round in this chapter. The architectures of VGG11, ResNet9, the auxiliary network, and the device-side and server-side model partitions are shown in Table 4.3.

In terms of different DNN PPs, EcoFed does not optimize PPs in DPFL to minimize training latency, but its efficient communication techniques can enhance the performance over a vanilla DPFL system for all PPs. Table 4.4 shows four different PPs used in the evaluation. In particular, a fixed PP is adopted, i.e., PP2, in the majority of experiments to align with the configuration of LGL [47]. However, the latency performance of EcoFed for all PPs is also investigated.

FL Training Hyperparameters: For each FL round, the server randomly selects 20 devices out of 100 devices, with a sampling ratio of 0.2, to participate in the current training round. For the aggregation of model updates, the standard FedAvg [111] method is used, which is employed by the *Aggregator*. The same data augmentation techniques and learning schedules are adopted for all methods. Specifically, the horizontal flip with a probability of 0.5 and the SGD optimizer with a learning rate of 0.01 are used. The total number of training rounds for all datasets is set to 500. These hyperparameters are consistent across all methods, ensuring a fair comparison of EcoFed against other baselines.

Testbed: To evaluate the system performance, a prototype using an edge server and five resource-constrained devices is developed. The edge server is equipped with a 2.5GHz Intel i7 8-core CPU and 16 GB RAM. Five Raspberry Pi 4 Model B single-board computers, each featuring a 1.5GHz quad-core ARM Cortex-A53 CPU, are used to represent resource-constrained devices. All devices and the server run the same training engine, PyTorch. The server and devices are connected to a router. To emulate different network conditions and

Table 4.4 Different PP of VGG11 and ResNet9 used for evaluation. Convolution layers denoted as C followed by the no. of filters; filter size is 3×3 for all convolution layers except for the downsampling convolution where filter size is 1×1 ; Max Pooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers, a max pooling layer, and downsampling convolution layers; number followed is no. of output channels. Communication size is represented as *channels* \times *width* \times *height*

PP	Device		Server		Communication size	
	VGG11	ResNet9	VGG11	ResNet9	Activation	Gradient
PP1	C64-MP	C64-MP	C128-MP- C256- C256-MP- C512- C512-MP- C512- C512- FC4096- FC4096- FC10	C128-MP- RB256- RB512- RB512- FC10	$64 \times$ $64 \times$	$64 \times$ 16×16 16×16
PP2	C64-MP- C128-MP	C64-MP- C128-MP	C256- C256-MP- C512- C512-MP- C512- C512- FC4096- FC4096- FC10	RB256- RB512- RB512- FC10	$128 \times$ 8×8	$128 \times$ 8×8 8×8
PP3	C64-MP- C128-MP- C256- C256-MP	C64-MP- C128-MP- RB256	C512- C512-MP- C512- C512- FC4096- FC4096- FC10	RB512- RB512- FC10	$256 \times$ 4×4	$256 \times$ 4×4 4×4
PP4	C64-MP- C128-MP- C256- C256-MP- C512- C512-MP	C64-MP- C128-MP- RB256- RB512	C512- C512- FC4096- FC4096- FC10	RB512- FC10	$512 \times$ 2×2	$512 \times$ 2×2 2×2

Table 4.5 Typical network bandwidths available in the UK.

Network type	Upload bandwidths (Mbps)	Download bandwidths (Mbps)
3G	0.4	3
3G HSPA+	3	6
4G LTE	5	20
4G LTE-Advanced	10	42
Home Broadband Wi-Fi	11	60
5G	20	200

evaluate the impact of varying upload and download bandwidths, the Linux `tc` command is used to emulate different network bandwidths. In particular, real-world data is referred to as shown in Table 4.5. Tests on network conditions of 3G HSPA+ (3G), 4G LTE-Advanced (4G), and 5G are carried out.

A simulation-based test environment for evaluating the learning performance is also employed. The simulation is conducted on a high-performance system equipped with a 2 GHz AMD EPYC 7713P 64-Core CPU, 252 GB RAM, and two Nvidia A6000 GPUs. The reason for conducting simulations on high-end GPUs is due to practical considerations for taking into account the time required for running the experiments. For each accuracy experiment, tests with three different random seeds are needed to mitigate the impact of randomness. This would take several months to complete all the tests. To expedite the collection of results, accuracy-related experiments are conducted on the simulated testbed.

However, both test environments used the same code and experimental configurations. In addition, it is verified that the accuracy of results obtained in the simulation environment is the same as that of experiments conducted in the physical testbed.

4.5.2 Evaluation Setup

Baselines: The evaluation includes four DML baseline methods. The centralized ML method is excluded from comparison because it requires transferring raw data from devices to the cloud for training. These baselines are as follows:

1. **Classic FL:** This approach involves training the entire model locally on each device.
2. **SFL (vanilla DPFL):** SFL is the first DPFL approach. It partitions the model into device-side and server-side models, which are trained on devices and the server, respectively. The activation and gradient of the output of the device-side model are

transferred between devices and the server for each data batch. SFL does not consider any communication optimization.

3. **Local generated loss (LGL):** This method, proposed in [47], introduces locally generated loss on devices to produce gradients for training the device-side model. This reduces the need for transmitting gradients from the server.
4. **FedGKT:** FedGKT [53] also incorporates local loss on devices. In addition, probabilistic predictions, known as soft labels [57], of the server-side models are periodically transferred between the devices and the server. Soft labels are used to distill the training of both device-side models and server-side models. A variant of FedGKT based on vanilla DPFL is implemented to compare it with other baselines using the same aggregation algorithm (FedAvg).
5. **EcoFed:** The proposed method, EcoFed, is implemented with the following configuration. Pre-trained initialization is conducted from a pre-trained model on the ImageNet dataset [27], and these weights are frozen during training. The *Replay Buffer* uses $\rho = 2$, meaning the buffer is updated every two rounds. For the *Compressor*, a linear 8-bit quantization implementation is adopted.

Evaluation Metrics: The evaluation of EcoFed and the baseline methods consists of two sets of metrics: **Metrics on learning performance:** *Accuracy:* The accuracy of each approach is evaluated on the global test dataset. For each baseline, the highest test accuracy achieved during the entire rounds of training is recorded. The results are averaged over three independent runs with different random seeds.

Metrics on system performance: System performances in terms of communication cost and training latency are reported: (i) *Communication cost:* This metric measures the communication overhead for one round of training. In addition, the communication cost versus test accuracy curve is generated to show the efficiency of communication to achieve a target accuracy. (ii) *Training latency:* Training latency is defined as the wall-clock time taken for one training round for each baseline under different network bandwidths.

4.5.3 Accuracy

The test accuracy curves for EcoFed and the four baseline methods, obtained using two different datasets (I.I.D. and Non-I.I.D. settings) and two DNN models (VGG11 and ResNet9), are illustrated in Figure 4.4. In addition, Table 4.6 presents the highest test accuracy achieved by each method.

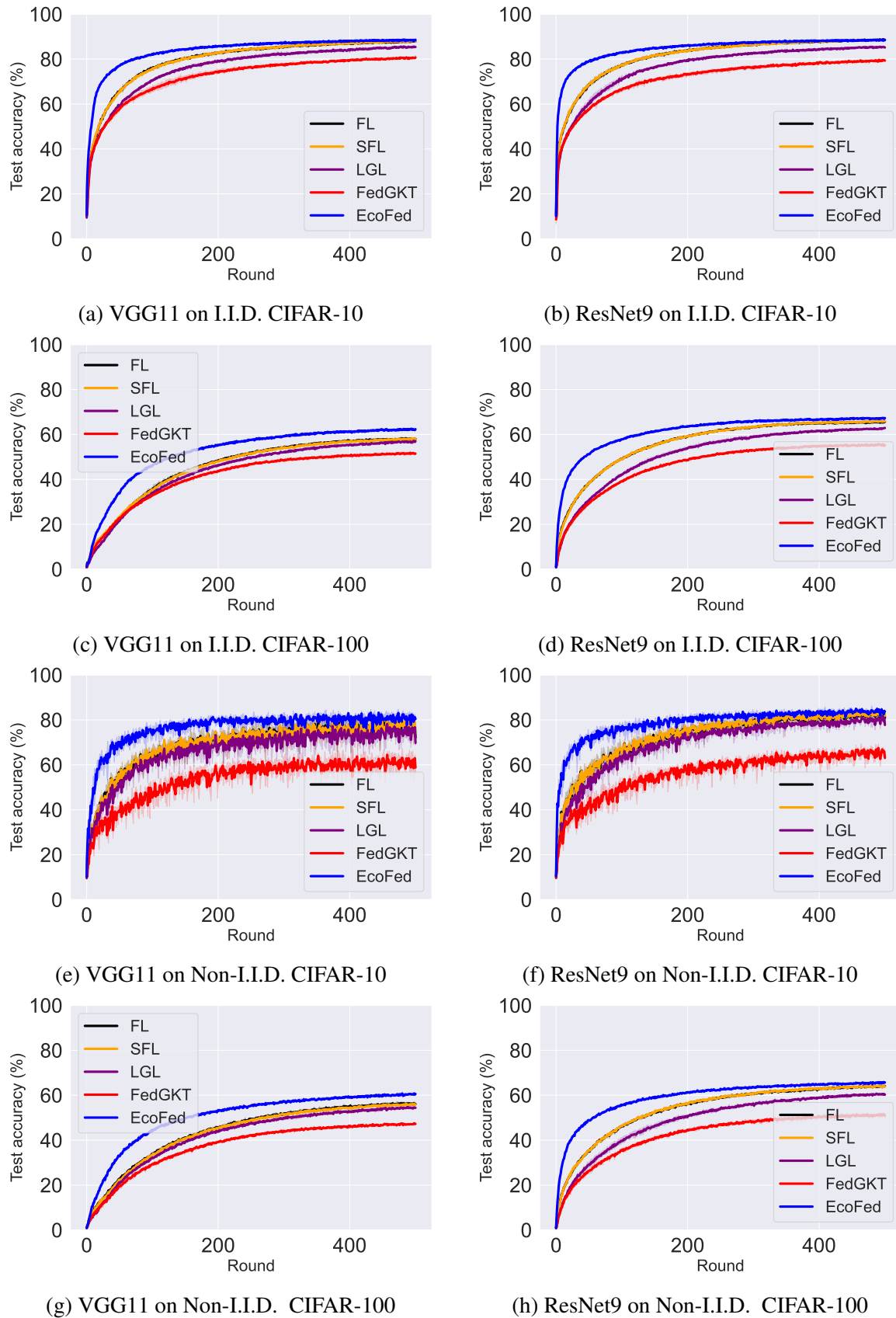


Fig. 4.4 Test accuracy curves of EcoFed and the baselines using VGG11 and ResNet9 in I.I.D. and Non-I.I.D. settings for CIFAR-10 and CIFAR-100 datasets.

Table 4.6 The test accuracy of EcoFed compared to the baselines for VGG11 and ResNet9 on two datasets on I.I.D. and Non-I.I.D. distribution. The results are an average of three independent runs with different random seeds.

Methods	CIFAR-10				CIFAR-100			
	I.I.D.		Non-I.I.D.		I.I.D.		Non-I.I.D.	
	VGG11	ResNet9	VGG11	ResNet9	VGG11	ResNet9	VGG11	ResNet9
FL	88.29 ± 0.18%	88.95 ± 0.08%	82.01 ± 0.34%	84.52 ± 0.28%	58.48 ± 0.28%	65.87 ± 0.13%	56.79 ± 0.42%	64.29 ± 0.06%
SFL	88.31 ± 0.29%	88.81 ± 0.11%	81.38 ± 0.16%	84.56 ± 0.07%	58.5 ± 0.12%	66.32 ± 0.12%	56.47 ± 0.13%	64.42 ± 0.11%
LGL	85.76 ± 0.27%	85.65 ± 0.26%	79.66 ± 0.48%	82.18 ± 0.35%	57.22 ± 0.26%	63.01 ± 0.14%	54.88 ± 0.28%	61 ± 0.05%
FedGKT	81.09 ± 0.39%	79.82 ± 0.19%	66.39 ± 0.64%	68.71 ± 0.63%	52.03 ± 0.51%	55.92 ± 0.26%	47.74 ± 0.23%	51.81 ± 0.47%
EcoFed	88.87 ± 0.11%	88.81 ± 0.09%	84.47 ± 0.17%	85.82 ± 0.31%	62.81 ± 0.11%	67.61 ± 0.15%	61.1 ± 0.31%	66.08 ± 0.17%

The results demonstrate that EcoFed mostly outperforms all baselines across all datasets and model architectures, except for FL on I.I.D. CIFAR-10 for ResNet9. Specifically, EcoFed achieves a maximum improvement of 4.63% in accuracy on Non-I.I.D. CIFAR-100 for VGG11 compared to SFL (and 4.31% compared to FL). In addition, EcoFed improves accuracy by up to 6.22% on Non-I.I.D. CIFAR-100 for VGG11 compared to LGL, and by up to 18.08% on Non-I.I.D. CIFAR-10 for VGG11 compared to FedGKT.

SFL shares similar accuracy and learning curves as FL since they fundamentally have the same training algorithm. On the other hand, LGL and FedGKT introduce local error signals to reduce communication. However, this approach leads to a significant loss in accuracy and slower convergence speed compared to FL and SFL. In contrast, EcoFed does not suffer from the accuracy degradation observed in LGL and FedGKT while achieving a more substantial reduction in communication costs compared to local loss-based DPFL methods.

Impact of pre-trained initialization and freezing weights on the device-side model:

To obtain a better understanding of accuracy achieved in Figure 4.4 and Table 4.6, accuracy degradation in local loss-based DPFL methods and the effects of using pre-trained initialization and frozen weights on w_c in EcoFed are investigated to answer the following three questions ³:

Can local error signals generated on the device degrade accuracy? From Table 4.6 and Figure 4.4, it is evident that local loss-based DPFL methods suffer a higher accuracy loss compared to other methods. Even when compared to FL and SFL, LGL and FedGKT

³The same conclusions were observed for CIFAR-100.

Table 4.7 The test accuracy of FedGKT (bi-direction) and FedGKT (uni-direction) on CIFAR-10. The results are an average of three independent runs with different random seeds.

FedGKT	I.I.D.		Non-I.I.D.	
	VGG11	ResNet9	VGG11	ResNet9
Bidirection	81.09 ±	79.82 ±	66.39 ±	68.71 ±
	0.39%	0.19%	0.64%	0.63%
Unidirection	83.18 ±	83.42 ±	75.89 ±	77.68 ±
	0.26%	0.27%	0.55%	0.53%

consistently achieve lower accuracy. In addition, local loss-based DPFL demonstrates a slower convergence speed, as shown in Figure 4.4, resulting from separately optimizing the device-side and server-side models with inconsistent gradient signals.

It is also observed that FedGKT exhibits much lower accuracy compared to LGL. In FedGKT, knowledge distillation is conducted on both device-side and server-side models. The results demonstrate that distilling device-side soft labels during server-side training negatively impacts accuracy. Table 4.7 presents the accuracy results of bidirectional FedGKT and unidirectional FedGKT (where only server-side soft labels are used for distilling the device-side model). An accuracy improvement is observed when removing distillation from device-side soft labels during server-side model training, as shown in unidirectional FedGKT. This further underscores that local training leads to accuracy degradation of the server-side model.

Can pre-trained initialization on the device-side model also improve accuracy for other baselines? The impact of pre-trained initialization on the device-side model for FL, SFL, and local loss-based DPFL methods is investigated. Table 4.8 presents the highest test accuracy achieved by each method when pre-trained initialization is applied to the device-side model. The results indicate that pre-trained initialization can significantly improve the accuracy of FL and SFL, particularly in the Non-I.I.D. setting. This finding has also been observed in recent research [17, 120].

Surprisingly, the results demonstrate that pre-trained initialization can also mitigate accuracy degradation caused by local error signals in local loss-based DPFL methods, which, has never been reported before. When all methods adopt pre-trained initialization, EcoFed still outperforms all local loss-based DPFL methods. Compared to FL and SFL, EcoFed achieves competitive accuracy performance (less than 1% loss).

Does training device-side models with local loss in the context of pre-trained initialization improve the accuracy of the server-side model? In LGL and FedGKT, the device-side model is trained using local error signals to avoid receiving gradients. However, after applying

Table 4.8 The test accuracy of EcoFed compared to the baselines on CIFAR-10 with pre-trained initialization on the device-side model. The results are an average of three independent runs with different random seeds.

Methods	I.I.D.		Non-I.I.D.	
	VGG11	ResNet9	VGG11	ResNet9
FL	89.48 ± 0.1%	89.55 ± 0.17%	84.81 ± 0.17%	86.06 ± 0.22%
SFL	89.44 ± 0.04%	89.57 ± 0.14%	85.02 ± 0.39%	86.19 ± 0.09%
LGL	88.06 ± 0.06%	88.29 ± 0.16%	83.44 ± 0.03%	85.04 ± 0.22%
FedGKT	83.59 ± 0.16%	82.35 ± 0.19%	72.73 ± 0.58%	74.36 ± 0.41%
EcoFed	88.87 ± 0.11%	88.81 ± 0.09%	84.47 ± 0.17%	85.82 ± 0.31%

pre-trained initialization, it is not clear whether training the device-side model using local loss can enhance the training of the server-side model.

To this end, the test accuracy of the device-side model (\mathbf{w}_c) and server-side model (\mathbf{w}_s) of LGL are recorded on CIFAR-10. The test accuracy of LGL (with pre-trained initialization) with trainable \mathbf{w}_c (device-side model) and frozen \mathbf{w}_c^* are compared. Table 4.9 shows that the trainable \mathbf{w}_c has significantly higher test accuracy than the frozen \mathbf{w}_c^* across I.I.D. and non-I.I.D. settings. However, it is surprising that the server-side model (\mathbf{w}_s), when the device-side model is frozen (denoted as \mathbf{w}_s under \mathbf{w}_c^*), achieves higher accuracy compared to the server-side model with a trainable device-side model (denoted as \mathbf{w}_s under \mathbf{w}_c).

The results suggest that training the device-side model can significantly enhance its accuracy, but it does not necessarily improve the accuracy of the server-side model and may even degrade it. However, achieving high accuracy with the server-side model is the ultimate goal. There is conjecture that local training on the device-side model is not necessary for enhancing the accuracy performance of the server-side model. The local error signals generated by the local auxiliary network may not be optimal for training the server-side models. Therefore, given these observations, EcoFed freezes the device-side weights when pre-trained initialization is adopted.

Impact of ρ and quantization on accuracy The accuracy performance of EcoFed under different hyper-parameter settings is further investigated. In particular, two hyperparameters: ρ are examined, which control the update frequency of the *Replay Buffer*, and quantization, which is used to reduce the size of transferred data. The impact of ρ and quantization on test accuracy for CIFAR-10 is shown in Figure 4.5. The accuracy gradually decreases as

Table 4.9 The test accuracy of device-side model of trainable w_c and frozen w_c^* in LGL (on CIFAR-10) and the highest test accuracy of respective server-side models under w_c and w_c^* . The results are an average of three independent runs with different random seeds.

LGL		I.I.D.		Non-I.I.D.	
		VGG11	ResNet9	VGG11	ResNet9
Device	Trainable w_c	78.61 ± 0.06%	78.68 ± 0.1%	76.78 ± 0.37%	76.77 ± 0.1%
	Frozen w_c^*	70.6 ± 0.2%	70.72 ± 0.06%	69.46 ± 1.35%	67.89 ± 0.59%
Server	w_s under w_c	87.82 ± 0.16%	88.04 ± 0.05%	83.05 ± 0.68%	84.9 ± 0.13%
	w_s under w_c^*	88.97 ± 0.06%	89.34 ± 0.18%	84.4 ± 0.37%	85.77 ± 0.12%

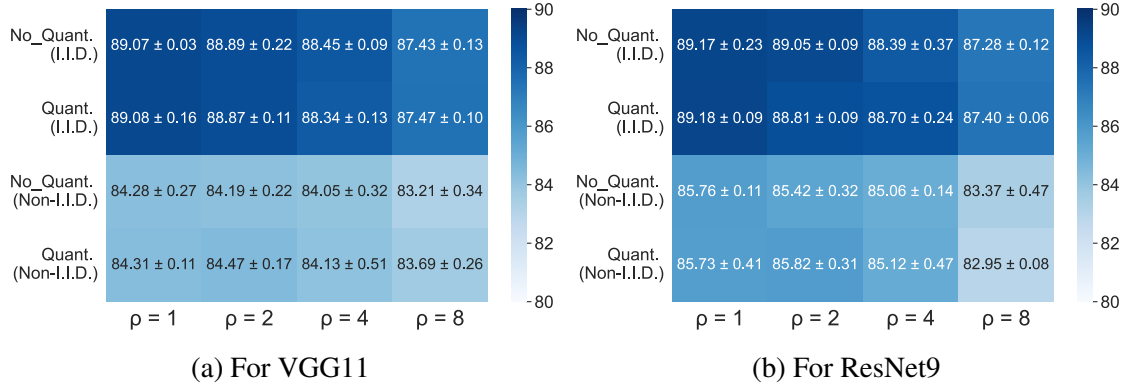


Fig. 4.5 Accuracy for different ρ values with or without quantization in EcoFed. The results are an average of three independent runs with different random seeds.

ρ increases since the update frequency is reduced. However, accuracy is less sensitive to quantization, as similar accuracy is achieved with or without quantization for the same ρ value. It is worth noting that when $\rho = 1$, EcoFed updates the cached buffer every round.

Dynamic control strategy for ρ : The dynamic control of ρ due to its larger impact on accuracy is further explored. A heuristic-based strategy to control the value of ρ is designed. In detail, accuracy improvements after every 40 rounds are recorded, denoted by Δ_{acc} . The value of ρ for the subsequent 40 rounds is then determined based on predefined heuristic rules outlined in Table 4.10. Figure 4.6 presents the highest accuracy and the corresponding change in ρ during training. In general, ρ gradually decreases as training accuracy approaches a plateau in the later stages. The average value of ρ with dynamic control is $\rho = 3.5$ throughout the entire training process. In addition, it is observed that the highest accuracy using dynamic

Table 4.10 Heuristic rules for dynamic ρ .

Period	$\Delta_{acc} \in (10^{-1}, \infty)$	$\Delta_{acc} \in (10^{-2}, 10^{-1}]$	$\Delta_{acc} \in (10^{-3}, 10^{-2}]$	$\Delta_{acc} \in (-\infty, 10^{-3}]$
ρ	8	4	2	1

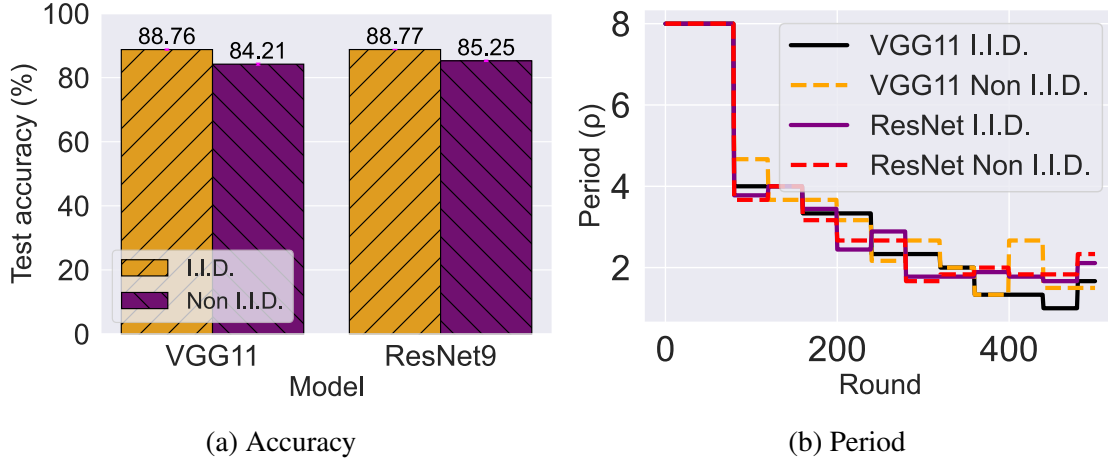


Fig. 4.6 Dynamic ρ values based on Table 4.10 in EcoFed. The results are an average of three independent runs with different random seeds.

control is achieved between fixed periods with $\rho = 2$ and $\rho = 4$. The results show that the dynamic control has a similar performance to the predefined $\rho = 2$.

Impact of pre-training dataset. Pre-trained weights for the device-side model, obtained from large-scale datasets (e.g., ImageNet) can be readily downloaded from open-source repositories such as Hugging Face⁴ and PyTorch Model Zoo⁵. This approach reduces pre-training time when using pre-trained initialization in EcoFed. However, pre-trained models on larger datasets are not always available for public access. Therefore, the impact on accuracy when using pre-trained weights that are trained from scratch on a small pre-training dataset, e.g., Tiny-ImageNet [80] is explored. In addition, given the challenges of collecting user data for pre-training, synthetic data is used for pre-training, such as CIFAR-5m [118] and SIP-17 [200].

Table 4.11 provides detailed descriptions of these datasets.

1. *ImageNet* [27] is a popular large-scale dataset (1.2M samples with 1000 classes) that has been widely used for pre-training on various tasks [63].

⁴<https://huggingface.co/>

⁵https://pytorch.org/serve/model_zoo.html

Table 4.11 Summary of pre-training datasets and training costs. The costs for pre-training the VGG11 model as pre-training time in seconds and the proportion relative to the total training time of EcoFed in brackets are reported.


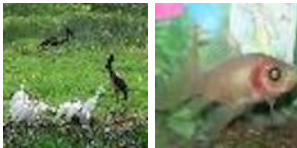
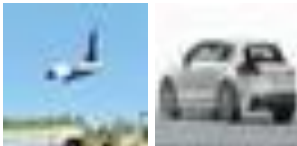

Dataset	#Class	#Samples	Visual examples	Pre-training cost
ImageNet	1000	1.2M		n/a
Tiny-ImageNet	200	100K		1117s (8%)
CIFAR-5m	10	50K		684s (5%)
SIP-17	15	18K		463s (3%)

Table 4.12 The test accuracy of EcoFed on CIFAR-10 with different pre-training datasets. **L** and **S** are used to denote large-scale or small-scale datasets, respectively. **Nat** and **Syn** are used to denote whether the data is natural or synthetic. The results are an average of three independent runs with different random seeds.

Pre-training datasets	I.I.D.		Non-I.I.D.	
	VGG11	ResNet9	VGG11	ResNet9
ImageNet (L, Nat)	88.87 ± 0.11%	88.81 ± 0.09%	84.47 ± 0.17%	85.82 ± 0.31%
Tiny-ImageNet (S, Nat)	89.32 ± 0.08%	88.99 ± 0.15%	85.66 ± 0.4%	85.86 ± 0.17%
CIFAR-5m (S, Syn)	87.92 ± 0.18%	87.94 ± 0.05%	83.61 ± 0.43%	83.6 ± 0.38%
SIP-17 (S, Syn)	86.81 ± 0.01%	86.98 ± 0.08%	81.45 ± 0.11%	83.49 ± 0.24%

2. *Tiny-ImageNet* [80] is a sampled version of ImageNet, containing 100K images from 200 classes, each with 500 training images.
3. *CIFAR-5m* [118] is a dataset of 5 million synthetic images. It was generated by sampling from the denoising diffusion probabilistic model (DDPM) [118]. 50K samples, with 5,000 samples per class, are further sampled from the generated CIFAR-5m dataset to construct a small-scale synthetic dataset that is equal to the size of CIFAR-10.
4. *SIP-17* [200], consists of 15 individual objects and 2 assembly objects such as crosses and gears, each with 1,200 synthetic images. All 15 of the single objects are used for the pre-training. The SIP-17 dataset is not only synthetic but also exhibits greater domain shift compared to CIFAR-10. The domain shift in the SIP-17 dataset refers to the difference in data distribution between the training and testing sets, which is more pronounced compared to CIFAR-10. The domain shift in SIP-17 compared to CIFAR-10 is because it contains images of industrial components, which are different to images of natural objects in CIFAR-10.

The pre-training cost in terms of the training time of VGG11 model using a single A6000 GPU is also reported. For instance, the CIFAR-5m dataset with the same size as CIFAR-10 took 684 seconds which is equivalent to 5% of the total training time of EcoFed on the Raspberry Pi prototype.

Table 4.12 presents the accuracy results on CIFAR-10 when using pre-trained weights from four different types of pre-training datasets. In general, EcoFed demonstrates robust generalization across various pre-training datasets. It achieves a higher level of accuracy

Table 4.13 Communication cost for one training round. It is worth noting that the communication cost is determined independently of different runs.

Methods	Communication cost	
	VGG11	ResNet9
FL	5.13 GB	1.4 GB
SFL	0.62 GB	0.62 GB
LGL	0.33 GB	0.33 GB
FedGKT	0.33 GB	0.33 GB
EcoFed w/o buffer	0.077 GB	0.077 GB
EcoFed w buffer	0 GB	0 GB

on natural datasets and maintains similar accuracy, even on small-scale datasets like Tiny-ImageNet. EcoFed has a slight decrease in accuracy compared to SFL (up to 0.96%) for CIFAR-5m. However, it still outperforms local loss-based methods such as LGL (by up to 3.95%). When dealing with the more challenging SIP-17 dataset with domain shifts, EcoFed still achieves high performance with a small accuracy loss (up to 1.83%) compared to SFL. In addition, it outperforms local loss-based methods such as LGL (up to 1.79%).

4.5.4 Communication Cost

Communication cost for one training round: The communication overhead of EcoFed and the four baseline methods are recorded and compared for one training round on the CIFAR-10 dataset as shown in Table 4.13⁶.

Compared to classic FL, other DPFL methods have a smaller communication overhead; for example, the communication cost is reduced by $8.27\times$ (SFL) and $15.55\times$ (LGL and FedGKT) when training VGG11. This reduction is attributed to the fact that DPFL methods only need to transfer the device-side model between the devices and the server. EcoFed (with buffer) achieves a further reduction in communication cost of $66.62\times$ on VGG11 and $18.18\times$ on ResNet9. When using the buffer, EcoFed eliminates the need for communication. In terms of the overall communication costs for all rounds of FL, EcoFed ($\rho = 2$) can reduce communication by $133.25\times$ on VGG11 and $36.36\times$ on ResNet9 compared to classic FL.

Compared to the other DPFL methods, EcoFed also significantly reduces the communication cost. EcoFed without buffer reduces the communication cost by $8.05\times$ and $4.29\times$ on VGG11 and ResNet9, respectively. LGL and FedGKT can halve the communication cost compared to SFL as they only require the activation to be transferred from devices to the

⁶For evaluation of communication cost and training latency of one training round, only results on the CIFAR-10 dataset is reported as system performance is independent of datasets.

server. However, due to the replay buffer technique in EcoFed, the average communication per round is further reduced by a factor of ρ . For instance, in the experiments with $\rho = 2$, the average communication per round is 0.0385 GB, which is $16.1\times$, $8.57\times$, and $8.57\times$ lower than SFL, LGL, and FedGKT, respectively.

Communication cost vs test accuracy: The cumulative communication cost for all training rounds on the two datasets is considered as the communication cost after each training round is recorded. Figure 4.7 highlights the communication cost incurred to achieve a target test accuracy.

Table 4.14 presents the communication costs incurred when achieving specific target accuracies. For instance, given a test accuracy target of 80% on I.I.D. CIFAR-10 for VGG11, FL, SFL, LGL, and FedGKT will require 729 GB, 90 GB, 71 GB, and 145 GB of data transfers, respectively. Similarly, for a 50% test accuracy target on I.I.D. CIFAR-100 the communication costs for VGG11 are 1121 GB, 139 GB, 110 GB, and 156 GB. However, EcoFed will only require 4 GB on I.I.D. CIFAR-10 and 7 GB on I.I.D. CIFAR-100 to achieve the same target test accuracy. For ResNet9, FL, SFL, and LGL will require 169 GB, 78 GB, and 67 GB of data to be transferred (150 GB, 65 GB, 64 GB, and 100 GB on I.I.D. CIFAR-100), respectively. FedGKT fails to achieve 80% test accuracy on I.I.D. CIFAR-10 and requires 100 GB on I.I.D. CIFAR-100 for 50% test accuracy. In contrast, EcoFed has a low volume of data transfer (4 GB on I.I.D. CIFAR-10 and 3 GB on I.I.D. CIFAR-100) to achieve the same target test accuracy.

	CIFAR-10 (I.I.D.)		CIFAR-100 (I.I.D.)	
	VGG11 @ 80%	ResNet9 @ 80%	VGG11 @ 50%	ResNet9 @ 50%
FL	729 GB	169 GB	1121 GB	150 GB
SFL	90 GB	78 GB	139 GB	65 GB
LGL	71 GB	67 GB	110 GB	64 GB
FedGKT	145 GB	N/A	156 GB	100 GB
EcoFed	4 GB	4 GB	7 GB	3 GB

Table 4.14 Communication costs incurred for different methods to achieve specific target accuracies.

For the Non-I.I.D. setting on both datasets, EcoFed shows higher communication efficiency in reaching a target test accuracy compared to all baselines. Notably, although LGL and FedGKT reduce communication by half per round (due to eliminated gradient transfers) compared to SFL, the cumulative volume of communication required to achieve the same level of accuracy does not significantly decrease. In some cases, such as ResNet9 on CIFAR-

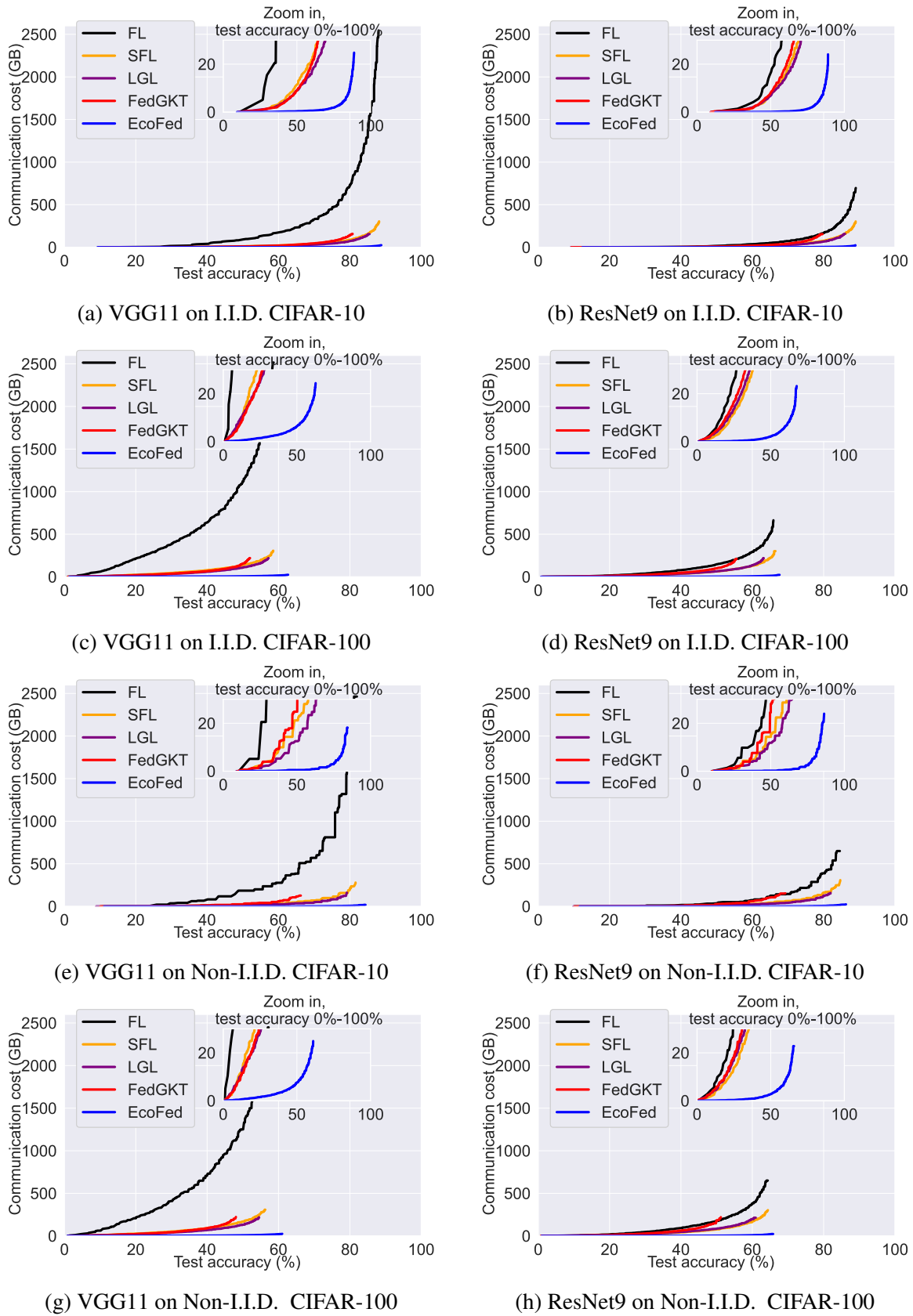


Fig. 4.7 The cumulative communication costs versus test accuracy of VGG11 and ResNet9 for the I.I.D and non-I.I.D. settings on CIFAR-10 and CIFAR-100 datasets.

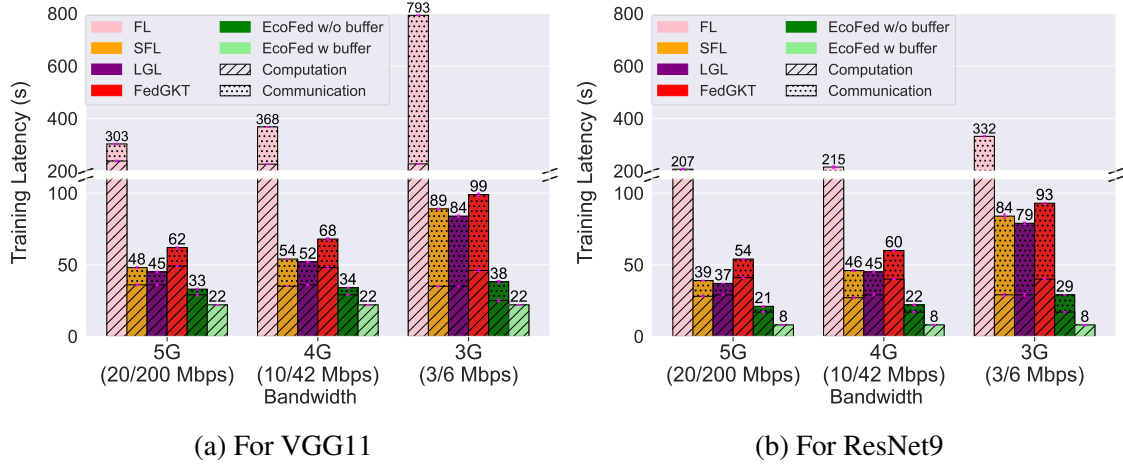


Fig. 4.8 Latency of one training round for VGG11 and ResNet9 under different network conditions for PP2. The results are an average of three independent runs.

100, it even increases. This is because of accuracy degradation and slower convergence using local error signals.

EcoFed reduces communication cost by $133.25\times$ on VGG11 ($36.36\times$ on ResNet9) compared to classic FL. Compared to SFL (vanilla DPFL), EcoFed improves communication by up to $16.1\times$. In short, classic FL, SFL (vanilla DPFL), and local loss-based DPFL incur substantial communication costs. In contrast, EcoFed has significantly lower communication costs and higher communication efficiency.

4.5.5 Training Latency

EcoFed enhances communication efficiency and eliminates the computation of gradients on the device, leading to a reduction of the overall training latency. The average training time for one round of EcoFed is compared against the baseline methods to quantify this benefit. Figure 4.8a and Figure 4.8b highlight that, under 5G conditions, EcoFed achieves a $9.33\times$ and $10.06\times$ speedup for VGG11 and ResNet9 without using buffer, respectively. In addition, compared to SFL (vanilla DPFL), EcoFed achieves a speedup of $1.47\times$ and $1.9\times$ on VGG11 and ResNet9 without using the buffer. When compared to local loss-based DPFL methods (LGL and FedGKT), there is an improvement of training latency of about $1.38\times$ and $1.91\times$ on VGG11 and $1.83\times$ and $2.62\times$ on ResNet9 without using the buffer.

The impact of network bandwidth is further considered, which will influence the training latency for devices operating in environments with limited network capabilities (e.g. mobile phones with 4G or 3G signal). Therefore, EcoFed and each baseline are evaluated under different network bandwidth conditions. When the network bandwidth is limited to 4G (10

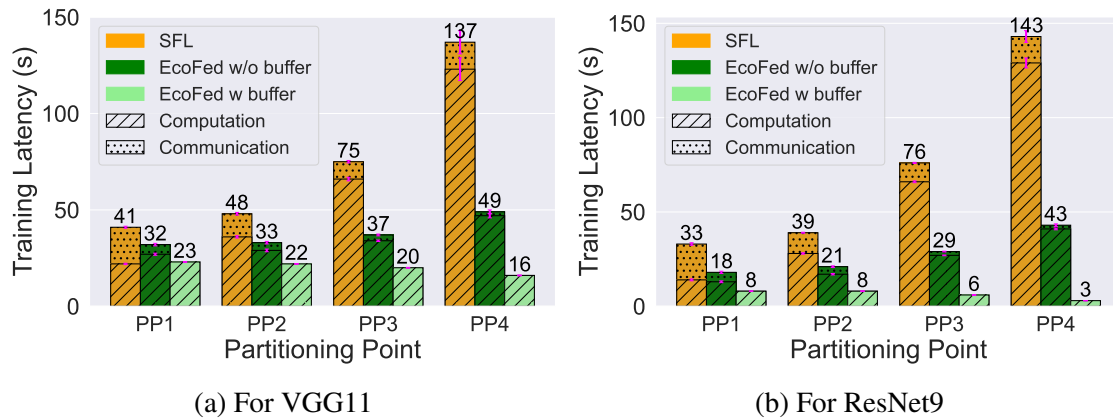


Fig. 4.9 Latency of one training round for VGG11 and ResNet9 under different PPs in 5G conditions. The results are an average of three independent runs.

Mbps and 42 Mbps for upload and download) and 3G (3 Mbps and 6 Mbps for upload and download), the training latency of FL, SFL, LGL, and FedGKT increases due to the higher communication costs for transferring the model and intermediate activation and gradient. It is worth noting that local loss-based methods (i.e. LGL and FedGKT) have a similar training latency compared to non-optimized vanilla DPFL (i.e. SFL) since they still incur large communication costs due to transferring activation. In particular, the upload bandwidth for sending activation is typically much lower than the download bandwidth used for sending gradient, and it has not been considered in local loss-based DPFL. This highlights the importance of reducing communication costs when transferring activation.

In contrast, EcoFed shows only a marginal increase in training latency, resulting in a $21.08\times$ and $2.38\times$ speedup on VGG11 and $11.26\times$ and $2.86\times$ speedup on ResNet9 in a 3G network, compared to FL and SFL, respectively. In addition, when EcoFed employs the buffer for training (half of the entire rounds with $\rho = 2$), the training latency is fundamentally independent of network bandwidths. This highlights the minimal bandwidth requirements of EcoFed.

Latency under different PPs: There exist variants of vanilla DPFL, which dynamically adjust the PP to minimize training latency [30, 55, 177]. The impact on training latency for different PPs in EcoFed is further investigated. Four PPs, i.e., PP1, PP2, PP3, and PP4 are considered. EcoFed and SFL for different PPs using 5G network bandwidth are compared. The overall latency and the components (computation and communication) that contribute to the latency are reported.

Figure 4.9 illustrates the latency of SFL and EcoFed under different PPs. As the partition point is placed deeper in the model, the overall training latency gradually increases since more computations are performed on the device. In addition, the communication latency

Table 4.15 The test accuracy of EcoFed on CIFAR-10 for different PPs using pre-trained ImageNet weights. The results are an average of three independent runs with different random seeds.

PPs	I.I.D.		Non-I.I.D.	
	VGG11	ResNet9	VGG11	ResNet9
PP1	88.36 ± 0.12%	88.59 ± 0.14%	83.26 ± 0.19%	84.34 ± 0.31%
PP2	88.87 ± 0.11%	88.81 ± 0.09%	84.47 ± 0.17%	85.82 ± 0.31%
PP3	92.6 ± 0.08%	92.26 ± 0.12%	90.74 ± 0.23%	90.42 ± 0.07%
PP4	92.34 ± 0.26%	91.96 ± 0.11%	90.88 ± 0.13%	89.44 ± 0.15%

becomes smaller as activation and gradient in the later layers are smaller in size. Overall, EcoFed achieves significant acceleration across all partition points without using the buffer. It results in speedups of $1.31\times$, $1.47\times$, $2.03\times$, and $2.83\times$ for VGG11 (and $1.8\times$, $1.9\times$, $2.6\times$, and $3.36\times$ for ResNet9) at PP1, PP2, PP3, and PP4, respectively. When the partition point is at the initial layers of the model (PP1 and PP2), EcoFed minimizes latency by reducing network communication latency. When the partition point is at the later layers of the model (PP3 and PP4), EcoFed reduces latency by eliminating gradient computation on the device. The lowest overall latency is achieved for PP1, where EcoFed achieves $1.8\times$ speedup compared to SFL. When the buffer is used there is no computation on the device. In this case, the training latency of EcoFed is further reduced and gradually decreases when the PP is set at later layers since less computation is offloaded to the server. In summary, EcoFed can further accelerate training latency for different PPs.

The accuracy of EcoFed for different PPs as shown in Table 4.15 was considered. Overall, EcoFed achieves high accuracy for different PPs. It is worth noting that there is a significant accuracy increase with EcoFed when the partition point is moved to later layers (i.e., PP3 and PP4). This indicates that initializing and freezing more layers of pre-trained weights significantly improves the training accuracy of federated learning, which is also observed in prior literature [17].

4.6 Existing Solutions

This section explores the existing solutions related to EcoFed on four aspects including communication reduction techniques in FL, DPFL methods, pre-training in FL, and layer-wise learning.

Communication reduction in FL: Communication reduction techniques in FL can be grouped into two categories: those that reduce communication frequency and those that compress the size of transferred data. Under the first category, a key technique involves increasing the interval between model aggregations, thus effectively reducing communication frequency between devices and the server [173, 180, 179]. Under the second category, compression approaches such as quantization and sparsification are used to minimize the size of updated model weights during each communication round [139, 50]. Another approach to address this issue is to incorporate distillation into training, which can eliminate the communication overhead of transferring model parameters in traditional FL [67]. However, the existing research in this area mainly focuses on the communication of updated models at the end of each FL round and does not consider the communication costs introduced in the DPFL systems.

DPFL methods: The literature on DPFL can be divided into two main categories: vanilla DPFL and local loss-based DPFL. SFL [165] is the first DPFL work, where a DNN is partitioned between the devices and the server. It adopts the advantages of both SL [169] and FL. In addition, dynamic partitioning strategies for DPFL [55, 30, 177] and pipeline scheduling [195] have been considered to optimize performance. However, these methods overlook the communication overhead introduced by DNN partitioning-based offloading.

Recently, DPFL approaches have been optimized by computing the local loss on the device to reduce communication costs [47, 53]. In these approaches, the device-side model is trained with local error signals generated by an additional auxiliary network. This eliminates the need to transfer gradient from the server. However, training the device-side model with local error signals may lead to sub-optimal accuracy performance. In addition, these approaches do not account for communication costs associated with transferring activation. The distinctions between classic FL, vanilla DPFL, local loss-based DPFL, and EcoFed are highlighted in Table 4.16.

Pre-training in FL: Pre-training a model is rarely investigated in the literature on FL. Instead, models are usually trained from random weights. Recent work has shown that pre-training can narrow the accuracy gap between FL and centralized learning, especially in non-IID settings [17, 120]. EcoFed introduces the use of pre-training on the device as an approach to reduce the accuracy loss associated with local loss-based DPFL methods, which is, for the first time, considered in DPFL.

Table 4.16 Comparing FL, vanilla DPFL, local loss-based DPFL, and EcoFed.

	FL	Vanilla DPFL (e.g., [165, 55, 30, 177])	Local loss-based DPFL (e.g., [47, 53])	EcoFed
Offers device-side acceleration	✗	✓	✓	✓
Reduces communication cost for DNN partitioning	n/a	✗	✓	✓
Optimizes communication arising from activation transfers	n/a	✗	✗	✓
Has low accuracy degradation	n/a	n/a	✗	✓

Layer-wise Learning: Another relevant research direction is layer-wise learning, where each layer of a DNN is independently trained using auxiliary networks [122, 8, 9]. However, these approaches are designed for resource-rich environments with centralized servers. As a result, the computation overhead of each layer and the communication costs to transfer intermediate data are rarely taken into consideration. EcoFed can be considered as a special case of parallel block-wise learning. However, the device-side model is deployed in a resource-constrained environment on devices with limited computation and communication resources, which is challenging.

4.7 Summary

This chapter presents EcoFed, a communication-efficient framework for DPFL. EcoFed introduces pre-trained initialization to eliminate the need for transferring gradient from the server to the devices in DPFL. In addition, EcoFed incorporates a replay buffer technique with quantization-based compression to further reduce the communication cost associated with transferring activation. In summary, EcoFed offers a promising solution for reducing both forward and backward communication in DPFL systems, especially tailored to resource-limited IoT devices. Evaluation of EcoFed demonstrates its ability to mitigate the accuracy degradation observed in state-of-the-art local loss-based DPFL methods. In addition, EcoFed significantly improves communication efficiency and training speed compared to classic FL and other state-of-the-art DPFL methods.

Both FedAdapt (Chapter 3) and EcoFed are based on a variant of the FL system, specifically DPFL, which integrates offloading-based training. However, it is worth noting that although a portion of the computation workload is offloaded from devices and the server, the total computation workload, including that of devices and the server, is not reduced compared

to classic FL. In the next chapter, the layer freezing technique will be investigated, which can further reduce the overall computation demand.

Chapter 5

Accelerating On-Device Training by Layer Freezing

Chapter 3 introduced FedAdapt, which adaptively offloads DNN training from devices to a server to create a DPFL system. The previous chapter presented EcoFed to decrease the communication costs in the DPFL system. This chapter will focus on on-device training within classic FL. In particular, the opportunity for incorporating layer freezing into the FL training process is explored. The approaches used in this chapter can also be adapted for both the devices and the server in a DPFL system. Therefore, it can be integrated with FedAdapt and EcoFed. The integration of these works will be presented in Chapter 6.

Layer freezing is a computation-saving technique recently utilized in centralized training [174, 98], where the parameters of specific layers are frozen during training to skip the gradient computation for those layers. Existing layer freezing methods can be categorized into two types: early-stage and accuracy-guaranteed. Early-stage layer freezing emphasizes aggressive acceleration in the initial stages, while accuracy-guaranteed methods prioritize achieving high accuracy. This chapter proposes FedFreeze - an effective layer freezing framework for FL combining features of both early-stage acceleration and accuracy-guaranteed layer freezing. There are dual-phase layer freezing techniques in FedFreeze, i.e., local layer freezing and global layer freezing. In particular, FedFreeze develops a novel regularization-based layer freezing approach on the device to apply early-stage layer freezing during the initial stages of local training for improving training speedup. In addition, a convergence-based layer freezing approach is also adopted in FedFreeze on the global server to ensure a high final accuracy.

Experiments conducted across various testbeds, datasets, models, and initialization methods demonstrate that FedFreeze can achieve a training speedup of up to $1.3\times$, while achieving comparable final accuracy to vanilla FL. In addition, FedFreeze demonstrates a

better balance between final accuracy and speedup performance compared to state-of-the-art early-stage and accuracy-guaranteed layer freezing techniques.

5.1 Motivation

FL training on IoT devices with limited resources has substantial challenges, as they must locally execute computationally intensive DNN training, often leading to impractical training latency [66, 119, 177].

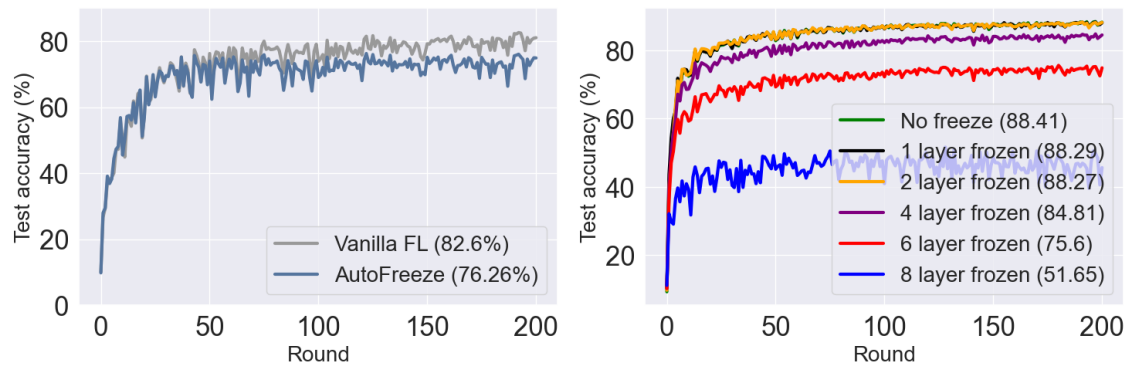
Why layer freezing? Various approaches have been integrated into FL to alleviate device-side computation overhead, e.g., pruning [71], partial training [58, 3], and offloading [165, 177, 178]. These approaches assume all parameters of the model need to be trained with the same workload. However, recent research [137, 116] has shown that different layers in DNN require a varying number of training rounds to converge, indicating a diverse training workload across different layers. Building on this insight, layer freezing has emerged as a valuable technique for reducing computation costs [174, 13, 45]. By freezing specific layers during training, the computation required for calculating gradients for those layers is eliminated. This highlights layer freezing as a distinctive method to identify unnecessary computation for layers during the training process.

In general, there are two types of layer freezing in the current literature, i.e., early-stage layer freezing and accuracy-guaranteed layer freezing. Early-stage layer freezing involves the early freezing of layers during the initial training stages to achieve acceleration [98]. Another variant of early-stage layer freezing is transfer learning where layers are permanently frozen and initialized with pre-trained weights before training commences [166]. As for accuracy-guaranteed layer freezing, the convergence behavior of the layers is continuously monitored during training, and a layer is frozen only upon convergence of its weights [174].

5.1.1 Learning Quickly or Learning Effectively

When applying layer freezing techniques to improve FL training, existing methods of early-stage layer freezing and accuracy-guaranteed layer freezing can only either reduce training time or achieve high final accuracy. In other words, existing methods do not find a balance between improving accuracy and training time across different settings.

Early-stage layer freezing suffers significant accuracy loss: Early-stage layer freezing methods aggressively freeze certain layers of the DNN in the initial training rounds, which can lead to inadequate updates of these layers, resulting in a significant loss of accuracy. Figure 5.1 shows the test accuracy results of two different early-stage layer freezing methods



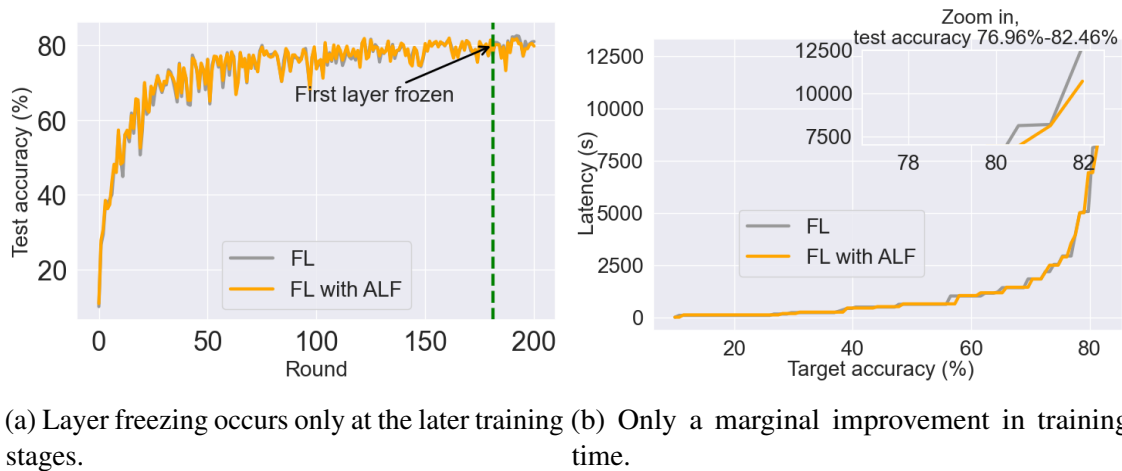
(a) Accuracy loss when using aggressive early- (b) FL test accuracy reduces as the number of stage layer freezing, AutoFreeze, compared to frozen layers increases using transfer learning-based early-stage freezing.

Fig. 5.1 Accuracy of FL training using early-stage layer freezing approaches for the VGG11 model on CIFAR-10 dataset.

applied to FL for the VGG11 model [158] on the CIFAR-10 dataset [79]. Specifically, Figure 5.1a shows the test accuracy of vanilla FL with AutoFreeze [98], an aggressive early-stage layer freezing method that freezes layers based on the lowest N percentile of the change rate of gradient, regardless of whether the layer has converged or not [98]. This approach results in a significant accuracy loss of 6.34% due to premature freezing of layers in the early stages.

Another early-stage layer freezing method is based on transfer learning where the layers are frozen before training happens and are initialized with pre-trained weights, known as pre-training initialization [166]. Figure 5.1b shows the test accuracy curves of transfer training-based layer freezing with FL training for a varying number of frozen layers. The results demonstrate that the final accuracy decreases as the number of frozen layers increases when using weights from the pre-trained ImageNet [27] model. When the number of frozen layers exceeds four layers, there is an accuracy loss of more than 3%. This observation emphasizes the importance of freezing only a limited number of layers to avoid significant accuracy loss when applying pre-training layer freezing. However, determining the maximum number of layers to freeze while staying within an acceptable accuracy threshold is a challenge. Furthermore, the domain shift between the target dataset and the pre-trained dataset can further degrade final accuracy.

Training speedup for accuracy-guaranteed layer freezing is marginal: Accuracy-guaranteed layer freezing observes the convergence of layers during training to decide whether to freeze them. A typical method for assessing layer convergence involves analyzing



(a) Layer freezing occurs only at the later training stages. (b) Only a marginal improvement in training time.

Fig. 5.2 Test accuracy for different rounds and latency incurred for a target accuracy in FL training with accuracy-guaranteed layer freezing for VGG11 on CIFAR-10.

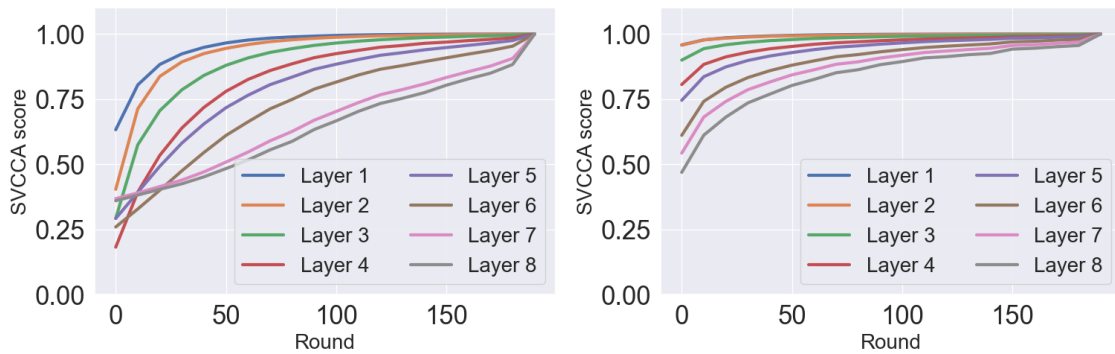
the gradient change. If the change for a layer is minimal, falling below a predefined threshold, it indicates that the layer can be frozen for the remainder of training.

The accuracy-guaranteed layer-freezing is empirically evaluated by implementing a state-of-the-art automatic layer freezing method (ALF) [105] method. Figure 5.2 shows the test accuracy curves of the global model versus FL rounds and the training latency against the target accuracy of FL and ALF for the VGG11 model on the CIFAR-10 dataset. As shown in Figure 5.2a, ALF (accuracy-guaranteed layer freezing) achieves the same highest accuracy as classic FL. However, it is noteworthy that the first instance of layer freezing occurred only at round 181 out of a total of 200 rounds, resulting in minimal speedup, as indicated in Figure 5.2b. Moreover, there are no perceived benefits from ALF during the initial FL rounds, as no layers are frozen. Figure 5.2b details the latency incurred to attain a target accuracy during training. Across a wide range of target accuracy, ALF does not contribute to any training acceleration.

5.1.2 Opportunity for Applying Early-stage Layer Freezing

Existing layer freezing approaches face the dilemma of prioritizing learning quickly or effectively but seldom achieve both simultaneously. This raises a fundamental question: *Can layer freezing be employed in the early stages of FL to speedup training while ensuring that final accuracy is not compromised?* Two key observations that indicate the feasibility of early-stage layer freezing are presented to address the question.

Observation 1: The bottom-up learning dynamic highlights that the bottom layers of a DNN require less training to converge. Recent research using canonical correlation



(a) Layers converge in a bottom-up manner in FL (b) Pre-trained initialization accelerates convergence, especially for the bottom layers.

Fig. 5.3 The SVCCA score of each layer during FL training for VGG11 on CIFAR-10. Layers are numbered bottom-up (i.e., from the initial to later layers).

analysis (CCA) has revealed that the initial (or bottom) layers of a DNN converge faster than the later (or top) layers, referred to as the bottom-up learning dynamic in the literature [137, 116]. This observation enables bottom-up layer freezing during training to reduce computation [174, 98].

To verify the bottom-up dynamic in the context of FL, a post-hoc layer-wise convergence analysis is conducted using the singular vector canonical correlation analysis (SVCCA) technique after training a VGG11 model on the CIFAR-10 dataset [137]. Specifically, the intermediate model weights after each round are preserved and utilized to produce output activation on a shared set of data samples. The activation of each layer in every round will then be employed to compute the SVCCA score, which is a normalized value ranging from 0 to 1 that quantifies the correlation between the in-training parameters and the final parameters of a layer. A higher score indicates a higher degree of convergence.

Figure 5.3 shows the SVCCA scores of each VGG11 layer during FL training. It is evident from Figure 5.3a that the bottom-up dynamic holds in the context of FL training with random initialization. In addition, the use of pre-trained initialization improves the convergence speed, particularly for bottom layers as shown in Figure 5.3b. These findings underscore the first observation: bottom layers require fewer updates through training to attain final parameters compared to top layers.

Observation 2: Multiple local learning updates can result in over-fitting. In each FL training round, the global model downloaded from the server is independently trained on each device using local data. Unlike traditional distributed stochastic gradient descent (DSGD) where only one or a small number of local updates are executed on devices [160, 189], the number of gradient updates in FL involves more steps that iterate over the entire local

samples multiple times. This is necessary for FL as the communication between devices and the server is a bottleneck compared to a cloud cluster [78, 75]. However, it has been demonstrated that multiple local updates lead to over-fitted models on local datasets [17, 89]. Moreover, in FL, the local dataset is typically non-I.I.D. [75], which can result in a more biased fitting to the local distribution.

Solution: Based on the aforementioned observations, there is an opportunity to apply early-stage layer freezing by reducing the ‘*redundant*’ training for the bottom layers even in the early-stage of FL training. This motivates the design of an ‘*aggressive*’ but ‘*temporary*’ layer freezing strategy during initial training on the device to mitigate redundant training of the bottom layers. Meanwhile, a more ‘*conservative*’ but ‘*permanent*’ layer freezing strategy is employed on the server to achieve a higher final accuracy.

To this end, FedFreeze an effective layer freezing framework for FL is proposed to accelerate early-stage device training and achieve a high global model accuracy. FedFreeze decouples how layer freezing is adopted for FL into two phases - firstly, device-side layer freezing, and then, server-side layer freezing. This separation allows different freezing strategies to be used on the device and server, thereby accelerating local training and achieving a highly accurate global model.

5.2 FedFreeze Overview

This section first formalizes the basic steps in FL training and the concept of parameter freezing in FL. The formulation will facilitate the design of the modules in FedFreeze. Then the FedFreeze architecture is presented followed by the FL training workflow using FedFreeze.

5.2.1 FL Steps and Layer Freezing Formulation

FL training steps: In FL, training data is distributed across M devices, and in each round of FL training, K devices ($K \leq M$) participate in training with their respective datasets $\mathcal{D} := \{\mathcal{D}_k\}_{k=1}^K$. The goal of FL is to optimize the following:

$$\min_{\boldsymbol{\theta}} \mathcal{F}(\boldsymbol{\theta}) = \sum_{k=1}^K \frac{|\mathcal{D}_k|}{|\mathcal{D}|} \mathcal{F}_k(\boldsymbol{\theta})$$

where $\mathcal{F}_k(\boldsymbol{\theta}) = \frac{1}{|\mathcal{D}_k|} \sum_{\zeta^t \sim \mathcal{D}_k} \mathcal{F}_k(\boldsymbol{\theta}; \zeta^t)$ (5.1)

where $\boldsymbol{\theta}$ is the model parameters; \mathcal{F} is the objective function on the server; \mathcal{F}_k is the objective function on device k (e.g., cross entropy loss [26]); ζ^t is a sampled mini-batch of data from \mathcal{D}_k at iteration t .

A FL round r can be divided into two phases: local learning and global aggregation. For each device k , the learned parameters $\boldsymbol{\theta}_k^r$ are optimized from the initial parameters $\boldsymbol{\theta}^{r-1}$ using an SGD algorithm, namely local learning.

$$\boldsymbol{\theta}_k^r = \arg \min_{\boldsymbol{\theta}} \mathcal{F}_k(\boldsymbol{\theta}) \quad \text{where } \boldsymbol{\theta} \xleftarrow{\text{init.}} \boldsymbol{\theta}^{r-1} \quad (5.2)$$

Thereafter, global aggregation is executed on the server:

$$\boldsymbol{\theta}^r = \sum_{k=1}^K \frac{|\mathcal{D}_k|}{|\mathcal{D}|} \boldsymbol{\theta}_k^r \quad (5.3)$$

Local learning and global aggregation are repeated for multiple rounds until the global model ($\boldsymbol{\theta}$) converges or achieves the desired accuracy.

Parameter freezing in FL: One method for solving Equation 5.2 is to employ the mini-batch gradient descent algorithm that updates the model parameters [56] using:

$$\boldsymbol{\theta}_k^r = \boldsymbol{\theta}_k^{r-1} - \gamma \nabla \mathcal{F}_k(\boldsymbol{\theta}_k^{r-1}) \quad (5.4)$$

with learning rate γ . By substituting Equation 5.1 to Equation 5.4, the update rule for the global model is derived as:

$$\begin{aligned} \boldsymbol{\theta}^r &= \boldsymbol{\theta}^{r-1} - \gamma \nabla \mathcal{F}(\boldsymbol{\theta}^{r-1}) \\ \nabla \mathcal{F}(\boldsymbol{\theta}^{r-1}) &= \sum_{k=1}^K \frac{|\mathcal{D}_k|}{|\mathcal{D}|} \nabla \mathcal{F}_k(\boldsymbol{\theta}_k^{r-1}) \end{aligned} \quad (5.5)$$

where $\nabla \mathcal{F}_k$ is the gradient on device k and $\nabla \mathcal{F}$ is the aggregated gradient on the server. To freeze parameters of the global model $\boldsymbol{\theta}$, a mask $\mathcal{M} \in \{0, 1\}^{|\boldsymbol{\theta}|}$ with the same size as $\boldsymbol{\theta}$ is applied to the global gradient $\nabla \mathcal{F}(\boldsymbol{\theta}^{r-1})$. This results in the following update rule:

$$\boldsymbol{\theta}^r = \boldsymbol{\theta}^{r-1} - \gamma (\mathcal{M} \odot \nabla \mathcal{F}(\boldsymbol{\theta}^{r-1})) \quad (5.6)$$

where \odot denotes the entry-wise (Hadamard) product.

Layer freezing: The sparsity within \mathcal{M} can either be structured or unstructured. In structured sparsity, entire components like neurons, filters, or layers are frozen. Unstructured

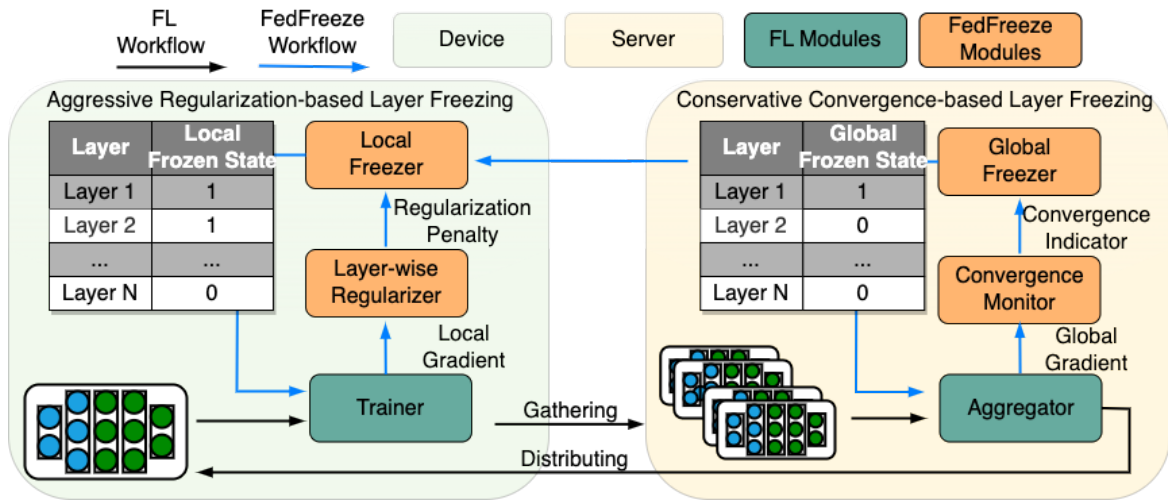


Fig. 5.4 The FedFreeze architecture.

sparsity, on the other hand, randomly prunes individual weights across the network, resulting in irregular patterns. Consequently, structured or unstructured parameter freezing techniques can be applied. In practice, structured layer freezing with regular sparsity at the layer has more computation and communication benefits compared to unstructured freezing. This is attributed to the fact that structured layer freezing can reduce the computation and communication costs of the frozen layer without requiring sparse optimization software and hardware.

5.2.2 FedFreeze Architecture

Figure 5.4 illustrates the architecture of FedFreeze on both devices and the server, highlighting both the FL modules and FedFreeze modules. On each device, the local *Trainer* updates the global model over the local dataset for several epochs¹. FedFreeze adopts an *aggressive regularization-based layer freezing* strategy to accelerate local learning, even in the initial FL rounds. A two-step process is developed for generating a list of frozen layers, referred to as the ‘Local State List’ which is required for implementing local layer freezing: (i) FedFreeze uses a local *Layer-wise Regularizer* module for generating a layer-wise regularization scheme based on the analysis of the local layers’ gradients provided by the local *Trainer* (Section 5.3.1); (ii) The *Local Freezer* combines the local regularization scheme with the list of the frozen layer from the *Global Freezer*, referred to as the ‘Global State List’ to create the Local State List (Section 5.3.2). The state of the layers from the list is used by the *Trainer* to freeze the required layers in the next training iterations.

¹One epoch is denoted as the complete training over all data points.

On the server, upon receiving the local model updates from all devices, the global *Aggregator* aggregates local gradients using aggregation algorithms, such as FedAvg [111]. Unlike on the device, FedFreeze adopts a *conservative convergence-based layer freezing* strategy to preserve the final accuracy. The *Convergence Monitor* module assesses the convergence behavior of the global model and sends the convergence metrics of each layer to the *Global Freezer*. The *Global Freezer* generates the ‘Global State List’ that is used by the *Aggregator* to freeze the layers of the global model. The Global State List is also sent to the local devices as it is utilized by the *Local Freezer*. By employing distinct freezing strategies on devices and servers, FedFreeze can accelerate local training while enhancing global accuracy.

Notation	Description
M	Total number of devices in FL
K	Number of devices participating in each FL round ($K \leq M$)
\mathcal{D}	The total dataset, distributed across devices: $\mathcal{D} := \{\mathcal{D}_k\}_{k=1}^K$
\mathcal{D}_k	Local dataset on device k
θ	Model parameters (global model)
$\mathcal{F}(\theta)$	Objective function on the server to optimize
$\mathcal{F}_k(\theta)$	Objective function on device k (e.g., loss function like cross-entropy)
ζ^t	Mini-batch of data sampled from local dataset \mathcal{D}_k at iteration t
r	FL training round
θ_k^r	Model parameters on device k at round r
γ	Learning rate
$\nabla \mathcal{F}_k(\theta_k^r)$	Gradient of the objective function on device k at round r
\mathcal{M}	Freezing mask for model parameters (binary mask)
\odot	Hadamard (element-wise) product
T	Number of local iterations for each FL round
\mathcal{M}_k^r	Local freezing mask for device k at round r
λ	Regularization coefficient in the loss function
$\ \theta - \theta^r\ $	Norm of the difference between current and initial parameters
$\Delta \theta^t$	Parameter update at iteration t
G_i	Upper bound of the gradient for layer i
T_i	Number of iterations allocated for layer i
G_{min}	Minimum gradient upper bound across all layers
ϵ	Fraction of iterations used to estimate layer-wise gradients

Table 5.1 Notation used in FedFreeze

Algorithm 2: FL training with FedFreeze

```

1 Input: Initial global weight  $\boldsymbol{\theta}^0$  and data  $\mathcal{D} := \{\mathcal{D}_k\}_{k=1}^K$ 
2 Output:  $\boldsymbol{\theta}^*$ 
3  $\boldsymbol{\theta}^r = \boldsymbol{\theta}^0$ ,  $\mathcal{M}^r = 1^{|\boldsymbol{\theta}|}$ ;
4 for each round  $r \in R$  do
5   for each device  $k \in K$  in parallel do
6      $\boldsymbol{\theta}_k^i = \boldsymbol{\theta}^r$ ;
7     for each local iteration  $t \in T$  do
8       if  $t \leq \varepsilon \cdot T$  then
9         /*Apply global freezing matrix*/
10         $\boldsymbol{\theta}_k^{t+1} = \boldsymbol{\theta}_k^t - \gamma(\mathcal{M}^r \odot \nabla \mathcal{F}_k(\boldsymbol{\theta}_k^t))$ ;
11      end
12      else
13        /*Monitor local gradient*/
14         $\nabla \mathcal{F}_k(\boldsymbol{\theta}_k^{\varepsilon \cdot T}) = \boldsymbol{\theta}_k^{\varepsilon \cdot T} - \boldsymbol{\theta}^r$ ;
15        /*Local Freezer outputs the freezing matrix based on the gradient
16         $\nabla \mathcal{F}_k(\boldsymbol{\theta}_k^{\varepsilon \cdot T})$ , as detailed in Section 5.3.1*/
17         $\mathcal{M}_k^t \leftarrow \text{LocalFreezer}(\nabla \mathcal{F}_k(\boldsymbol{\theta}_k^{\varepsilon \cdot T}))$ ;
18         $\mathcal{M}_k^t = \mathcal{M}_k^t \vee \mathcal{M}^r$ ; ▷ Element-wise OR operation
19        /*Apply local freezing matrix*/
20         $\boldsymbol{\theta}_k^{t+1} = \boldsymbol{\theta}_k^t - \gamma(\mathcal{M}_k^t \odot \nabla \mathcal{F}_k(\boldsymbol{\theta}_k^t))$ ;
21      end
22    end
23     $\nabla \mathcal{F}_k(\boldsymbol{\theta}_k^r) = \boldsymbol{\theta}_k^T - \boldsymbol{\theta}^r$ ;
24  /*Monitor global gradient*/
25  Collect gradients  $\nabla \mathcal{F}_k(\boldsymbol{\theta}_k^r)$  from each device  $k$ ;
26   $\nabla \mathcal{F}(\boldsymbol{\theta}^r) = \sum_{k=1}^K \frac{|\mathcal{D}_k|}{|\mathcal{D}|} \nabla \mathcal{F}_k(\boldsymbol{\theta}_k^r)$ ;
27  /*Apply global freezing matrix*/
28   $\boldsymbol{\theta}^{r+1} = \boldsymbol{\theta}^r - \gamma(\mathcal{M}^r \odot \nabla \mathcal{F}(\boldsymbol{\theta}^r))$ ;
29  /*Global Freezer outputs the freezing matrix based on the gradient  $\nabla \mathcal{F}(\boldsymbol{\theta}^r)$ , as
30  detailed in Section 5.3.2*/
31   $\mathcal{M}^{r+1} \leftarrow \text{GlobalFreezer}(\nabla \mathcal{F}(\boldsymbol{\theta}^r))$ 
32 end
33  $\boldsymbol{\theta}^* = \boldsymbol{\theta}^R$ ;
34 return  $\boldsymbol{\theta}^*$ 

```

5.2.3 FL Training Workflow with FedFreeze

Algorithm 2 outlines the FL workflow incorporating FedFreeze. FL repetitively performs two phases of round training, i.e., parallel local learning (Line 5 - Line 23) and synchronized global aggregation (Line 24 - Line 30), until the optimal model θ^* is achieved. Table 5.1 lists the notations used in this chapter.

Parallel local learning: Upon receiving the global weights θ^r from the server in round r (line 6), each device k independently conducts local updates on θ_k^t for T iterations (Line 7 - Line 23). During the initial $\varepsilon \cdot T$ iterations, FedFreeze only applies the global freezing matrix \mathcal{M}^r for updating the model (Line 10). This set of iterations is also utilized for gathering layer-wise gradients to generate the local freezing matrix \mathcal{M}_k^r . Following the initial $\varepsilon \cdot T$ iterations, a local freezing matrix \mathcal{M}_k^t is calculated based on the accumulated gradient $\nabla \mathcal{F}_k(\theta_k^{\varepsilon \cdot T})$ by the *Local Freezer*, as detailed in Section 5.3.1 (Line 16). Additionally, the local freezing matrix is merged with the global freezing matrix \mathcal{M}^r (Line 17), and this combined matrix is utilized to mask the model update (Line 19). Upon completion of local training, the accumulated gradient is transmitted to the server (Line 22).

Synchronized global aggregation: Upon receiving the updated gradients from all devices for round r (Line 25), the server proceeds to perform a gradient update on the global model. Initially, it aggregates the gradients from devices to generate an updated gradient $\nabla \mathcal{F}(\theta^r)$ (Line 26). Subsequently, this aggregated gradient $\nabla \mathcal{F}(\theta^r)$ is utilized to update the global model combined with the global freezing matrix \mathcal{M}^r (Line 28). Moreover, FedFreeze assesses convergence on the global gradient $\nabla \mathcal{F}(\theta^r)$ and updates the global freezing matrix by the *Global Freezer*, as detailed in Section 5.3.2 (Line 30).

5.3 Design of Layer Freezing in FedFreeze

This section presents the design of the device-side and server-side layer freezing strategies, namely *aggressive regularization-based layer freezing* and *conservative convergence-based layer freezing*. A new reformulation of regularization loss, facilitating the design of the regularization-based layer freezing is first proposed (Section 5.3.1). Then convergence-based layer freezing is discussed on the server-side model for obtaining high accuracy (Section 5.3.2).

5.3.1 Aggressive Regularization-based Layer Freezing

Local training with layer regularization: Section 5.1 recognized that bottom layers in DNN models require fewer training updates in FL training and multiple local learning updates on

the local dataset result in over-fitting. One solution to address these issues is by introducing an additional regularization term [187, 76] on the traditional loss (e.g., cross-entropy loss [194]). FedProx [89] is the first work proposed in FL to add a regularization loss on the local training to reduce the gap between the initial weights $\boldsymbol{\theta}^r$ of round r and its local updates as shown below:

$$\begin{aligned} & \min_{\boldsymbol{\theta}} \mathcal{F}_k(\boldsymbol{\theta}; \boldsymbol{\theta}^r) \\ \text{where } & \mathcal{F}_k(\boldsymbol{\theta}) = \underbrace{\frac{1}{|\mathcal{D}_k|} \sum_{\zeta^t \sim \mathcal{D}_k} \mathcal{F}_k(\boldsymbol{\theta}; \zeta^t)}_{\text{Normal loss}} + \underbrace{\lambda \|\boldsymbol{\theta} - \boldsymbol{\theta}^r\|}_{\text{Regularization loss}} \end{aligned} \quad (5.7)$$

where $\boldsymbol{\theta}^r$ is the initial weights and λ is a penalty coefficient for the change of the model parameters, $\|\boldsymbol{\theta} - \boldsymbol{\theta}^r\|$. A higher value of λ corresponds to a greater penalty added to the loss. This regularization loss guides the optimization of weights $\boldsymbol{\theta}$ to mitigate the effect of statistical heterogeneity of local Non-I.I.D. data [89]. The additional regularization term $\|\boldsymbol{\theta} - \boldsymbol{\theta}^r\|$ is equal to the norm of the gradient. Therefore, it facilitates faster convergence of parameters by encouraging small gradient updates. Although adding layer regularization loss can result in faster convergence, it does not provide any early-stage acceleration in FL. This is because, at the start of training, the normal loss dominates the gradient updates, while regulation loss has a minor impact. This motivates the reconsideration of the approach to applying layer regularization.

Reformulating regularization loss by layer freezing: In each local training round r , loss regularization represented as $\lambda \|\boldsymbol{\theta} - \boldsymbol{\theta}^r\|$ accelerates convergence by adding the regularization term to the local loss function $\mathcal{F}_k(\boldsymbol{\theta})$. However, the ‘penalty’ during training cannot be precisely controlled as it is jointly optimized with the normal learning loss. Furthermore, the regularization loss does not directly result in computational savings in the early stages. To address this, the traditional regularization loss is first reformulated, and a corresponding algorithm is proposed to enable early-stage layer freezing, achieving the same effect as traditional loss regularization.

In the local learning of global round r of FL, device k updates the model from the initialized $\boldsymbol{\theta}^r$ for T iterations over the dataset \mathcal{D}_k . For each iteration t , the model is updated as follows:

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} + \Delta\boldsymbol{\theta}^t \quad \text{where} \quad \Delta\boldsymbol{\theta}^t = \gamma \nabla \mathcal{F}_k(\boldsymbol{\theta}^t) \quad (5.8)$$

$\Delta\boldsymbol{\theta}^t$ represents the parameter changes, which is equal to the product of the γ (learning rate) and $\nabla\mathcal{F}_k(\boldsymbol{\theta}^t)$ (the gradient)². Therefore, the regularization loss after T iterations $\lambda\|\boldsymbol{\theta}^T - \boldsymbol{\theta}^r\|$ is reformulated as:

$$\begin{aligned}
& \lambda\|\boldsymbol{\theta}^T - \boldsymbol{\theta}^r\|^2 \\
&= \lambda\|\boldsymbol{\theta}^T - \boldsymbol{\theta}^0\|^2 \\
&= \lambda\|(\boldsymbol{\theta}^{T-1} + \Delta\boldsymbol{\theta}^T) - \boldsymbol{\theta}^0\|^2 \\
&= \lambda\|((\boldsymbol{\theta}^{T-2} + \Delta\boldsymbol{\theta}^{T-1}) + \Delta\boldsymbol{\theta}^T) - \boldsymbol{\theta}^0\|^2 \\
&= \lambda\|((\boldsymbol{\theta}^0 + \Delta\boldsymbol{\theta}^1) + \Delta\boldsymbol{\theta}^2) + \dots + \Delta\boldsymbol{\theta}^T - \boldsymbol{\theta}^0\|^2 \\
&= \lambda\|\Delta\boldsymbol{\theta}^1 + \Delta\boldsymbol{\theta}^2 + \dots + \Delta\boldsymbol{\theta}^T\|^2 \\
&= \lambda\|\sum_{t=0}^T \Delta\boldsymbol{\theta}^t\|^2
\end{aligned} \tag{5.9}$$

Consider the following assumption - *the squared norm of the stochastic gradient has an upper bound on the local objective function, i.e., $\|\nabla\mathcal{F}_k(\boldsymbol{\theta}^t)\|^2 \leq G, \forall k, \forall t$. In addition, for each layer i ($\boldsymbol{\theta}_i^t$), there is a corresponding upper bound, i.e., $\|\nabla\mathcal{F}_k(\boldsymbol{\theta}_i^t)\|^2 \leq G_i, \forall k, \forall t$ ³. Based on this assumption, the upper bound of regularization loss for T iterations is as follows:*

$$\lambda\|\sum_{t=0}^T \Delta\boldsymbol{\theta}^t\|^2 \leq \lambda\sum_{t=0}^T \|\Delta\boldsymbol{\theta}^t\|^2 \leq \lambda\gamma TG \tag{5.10}$$

where λ is the penalty coefficient that controls the degree of penalty. For local training of T iterations, the regularization loss $\lambda\|\boldsymbol{\theta}^t - \boldsymbol{\theta}^r\|^2$ has an upper bound of $\lambda\gamma TG$ on overall updates. In terms of layer i , the upper bound of regularization loss is $\lambda\gamma TG_i$.

Given the upper bound of the regularization loss ($\lambda\gamma TG_i$), the traditional regularization loss is reconsidered to incorporate it into the local training of different layers by using layer freezing. For traditional regularization loss, T is a fixed constant for all layers, and the layer-wise regularization effect is achieved by reducing the norm of gradient G_i through loss optimization. *An alternative approach to applying the same regularization effect is to limit the parameter T_i for different layers instead of reducing G_i .* In other words, one can allocate different lengths of iterations (T_i) for each layer i to be trained to achieve the same regularization target (Equation 5.9) as traditional loss regularization.

Determining T_i for each layer: In the traditional regularization loss $\lambda\gamma TG_i$ for layer i , if a layer has a large G_i (the upper bound of the gradient of layer i), the regularization loss will impose a higher penalty on this layer. Building upon this insight, an algorithm is proposed to

²To simplify the notation, $+\Delta\boldsymbol{\theta}^t$ is used to represent gradient update, while the minus sign conventionally used in SGD can be replaced by a negative of γ .

³The assumption is commonly presented in the literature that analyzes the convergence of FL [173, 178, 189].

automatically determine T_i based on the G_i of the layer. Therefore, for local training of T iterations, T_i is calculated of layer i using:

$$T_i = \lambda T \times \frac{G_{min}}{G_i} \quad (5.11)$$

where G_i is the gradient upper bound of layer i and G_{min} is the minimum G_i of all layers. The above equation ensures that the layer with a higher G_i will be penalized more by being allocated with a smaller T_i for training. In practice, the exact value of G_i is not known as it is the theoretical upper bound of the gradient. Therefore, the average value of the gradient for layer i is used as an estimate of G_i .

Adaptive changes of T_i : Traditional regularization loss dynamically adjusts the penalty as training progresses - as G_i becomes smaller, the regularization penalty automatically diminishes. In line with this, FedFreeze adaptively adjusts T_i . FedFreeze records the G_i across layers in the first round and calculates their average denoted as \bar{G}^0 . For the subsequent rounds, the penalty λ is adjusted by considering the change of \bar{G}^r , with $\lambda = \lambda \times \bar{G}^0 / \bar{G}^r$. Therefore, as \bar{G}^r decreases during training, λ becomes larger, resulting in less regularization penalty on each layer.

Overall algorithm: At the start of every local training of T iteration, all layers are trained for $\varepsilon \cdot T$ iterations to allow FedFreeze estimates G_i of layer i . For the remaining iterations $(1 - \varepsilon) \cdot T$, layer i will be trained for only T_i iterations calculated using Equation 5.11⁴. In addition, λ is dynamically optimized based on the changes in the average gradients of layers \bar{G}^r .

Impact of ε on local freezing: During local freezing, $\varepsilon \times T$ defines the number of local training iterations dedicated to approximating parameter changes. A smaller ε allows for more extensive local freezing, thereby accelerating training by reducing the need to update all model parameters in each round. However, this approach may lead to less accurate gradient approximations. Conversely, a larger ε provides more accurate gradient estimates by incorporating a larger number of model update iterations, but it decreases the efficiency of local freezing, thus slowing the process. In our experiments, we manually set $\varepsilon = 0.2$ to ensure that the model processes all local data once across five iterations of all data samples in the FL setting.

⁴Since $\varepsilon \cdot T$ iterations are required for estimating G_i , the remaining iterations $(1 - \varepsilon) \cdot T$ are used to calculate T_i instead of T .

5.3.2 Conservative Convergence-based Layer Freezing

On the server, FedFreeze employs a *conservative* convergence-based strategy to apply layer freezing to guarantee the final accuracy. The parameters of the layer are monitored to determine whether a layer has converged.

Metrics for measuring layer convergence: There are two metrics for measuring layer convergence: (1) The gradient-based metric determines the stability of the layers by checking whether there is a change of gradient for a layer [16]. (2) The activation-based metric that determines the stability by assessing the activation generated by a layer [174].

FedFreeze utilizes the gradient-based metric, as the activation-based metric often requires a reference model for comparison, which is impractical in FL⁵. In FedFreeze, the *Convergence Monitor* records the average norm of the server-side gradients for each layer. In addition, the exponential moving average (EMA) method is used to calculate the EMA of the average norm of server gradients to minimize the impact of gradient variation. In addition, the EMA method is more computationally efficient as only the latest gradients need to be saved in the memory [16].

Determining if a layer has converged: On the server, if a layer in the global model is frozen, then its parameters remain unchanged during both global aggregation and local learning for the subsequent rounds. Therefore, a *conservative* criteria should be adopted to ensure that the final accuracy of the global model is not reduced. Two stringent conditions are set by the *Global Freezer* to determine whether the layer has converged and avoid premature layer freezing:

Condition 1: The EMA of the average norm of server-side gradients for a layer is below a pre-defined threshold compared to the initial gradients;

Condition 2: The change of the EMA gradient is considered to be negligible if it is less than a predefined threshold.

The rationale behind these two conditions is to verify that the gradient of the layer is smaller than the initial gradient (Condition 1) and that there will be no substantial future change in the gradient (Condition 2). Upon satisfaction of both conditions, the *Global Freezer* deems the layer to have converged and proceeds to freeze it. The choice of the two pre-defined thresholds is discussed in Section 5.4.

Bottom-up layer freezing: The order in which the layers are frozen determines whether there will be computational benefits to training. Freezing a layer accelerates training only when all preceding layers are frozen. This is also referred to as gradient locking introduced by

⁵Although recent research proposes the use of an in-training model instead of a fully-trained model as a reference model, it is unrealistic in a real-world FL setting since it requires parallel training of a reference model on devices.

backward propagation [64]. Recent research confirms that the layers of the model converge in a bottom-up manner during the training. Observation 2 in Section 5.1 suggests that bottom-up learning also holds in FL. Therefore, FedFreeze adopts bottom-up layer freezing and only freezes a layer in the global model if all preceding layers are already frozen.

5.4 Evaluation

This section will present the results obtained from evaluating FedFreeze. Specifically, the end-to-end performance and the performance breakdown by comparing FedFreeze to other state-of-the-art baselines are presented. In addition, the impact of hyperparameters and system overhead of FedFreeze are discussed.

5.4.1 Evaluation Setup

The setup, namely the datasets and models, training hyperparameters, experimental testbed, baselines, and metrics, used to evaluate FedFreeze is considered as follows:

Datasets and models: FedFreeze is evaluated on three datasets, namely FMNIST [181], CIFAR-10 [79], and CIFAR-100 [79]. For data partitioning on devices, the common method reported in the literature [111] is adopted to simulate a non-I.I.D. setting of FL [75]. In particular, the dataset is sorted based on labels to create 500 shards. Each device is then randomly assigned 5 shards such that each device has training samples from around half of all available classes. The test dataset is on the server for evaluating model performance after each training round.

Three popular CNNs are trained: LeNet [81] (lightweight CNN), VGG11 [158] (plain CNN), and ResNet12 [54] (residual CNN) on the FMNIST, CIFAR-10, and CIFAR-100 datasets, respectively. A LeNet model is trained for the FMNIST task because it has a similar architecture as the VGG5 model used in Chapter 3 and is more manageable for the Raspberry Pi testbed, detailed later in this section, in terms of training time required. The architectures of the CNNs are shown in Table 5.2.

FL training hyperparameters: For each FL round, 10 devices are uniformly sampled from a pool of 100 devices participating in a round of training. The most popular aggregation algorithm is adopted, i.e., standard FedAvg [111] for the *Aggregator* module in FedFreeze. The same data augmentation of horizon flip and random crop is used for all experiments. The SGD optimizer with a constant learning rate of 0.01 is employed. The total number of training rounds is set to 200 for training on all datasets. For local training, the local epoch is set to 10 for all datasets. The pre-trained weights of VGG11 are obtained from the PyTorch

Table 5.2 Evaluated models. Convolution layers denoted as C followed by the no. of filters; filter size of convolution layer is 5×5 for LeNet and 3×3 for VGG11 and ResNet12 except for downsampling convolution which is 1×1 ; Max Pooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers and a downsampling convolution layer; number following is no. of output channels. The batch normalization layer is applied after every convolutional layer in VGG11 and ResNet12.

Dataset	Model	Architecture
FMNIST	LeNet	C6-MP-C16-MP-FC120-FC84-FC10
CIFAR-10	VGG11	C64-MP-C128-MP-C256-C256-MP-C512-C512-MP-C512-C512-FC512-FC512-FC10
CIFAR-100	ResNet12	C64-MP-C64-MP-RB64-RB128-RB256-FC100



(a) Raspberry Pi cluster.

(b) Jetson Nano cluster.

Fig. 5.5 Two prototypes are used to evaluate FedFreeze. A Raspberry Pi cluster with a laptop (edge server) and a Jetson Nano cluster with an Nvidia A6000 GPU server.

Model Zoo⁶ that was trained on the ImageNet [27] dataset, while the pre-trained weights of LeNet and ResNet12 are trained on the Tiny-ImageNet [80] dataset.

Testbed: To evaluate the system performance (i.e., training latency), two prototypes are considered as shown in Figure 5.5. The first is a Raspberry Pi (low-end IoT device) cluster and the second is a Jetson Nano (high-end IoT device) cluster. The Raspberry Pi cluster consists of 10 Raspberry Pi 4 Model B single-board computers, each with a 1.5GHz quad-core ARM Cortex-A53 CPU. A laptop serves as the edge server that has a 2.5GHz Intel i7 8-core CPU and 16GB RAM. In the Jetson Nano cluster, 10 Jetson Nano development boards, each with a 1.43GHz quad-core ARM Cortex-A57 CPU and a 128-core Maxwell GPU. The devices are connected to a cloud server with 2GHz AMD EPYC 7713P 64-Core CPU, 252GB RAM, and an Nvidia A6000 GPU. Communication between devices and the

⁶https://pytorch.org/serve/model_zoo.html

server is using socket TCP with a bandwidth of 100 Mbps. All devices and server use PyTorch as the training framework. The Jetson Nano cluster enables the training of both the VGG11 and ResNet12 models on a real-world prototype compared to the simulated testbed utilized in Chapter 4. This facilitates the reporting of the training time for entire rounds.

Choice of baselines: Vanilla FL is considered, which refers to the training of classic FL without using layer freezing. State-of-the-art layer freezing methods are further considered in both centralized and FL training contexts. In the context of FL, an approach named **ALF** [105] is considered, which is a convergence-based layer freezing approach for the server side. It calculates a metric referred to as ‘perturbation effectiveness’ to analyze the convergence of layers. The same metric is reported in adaptive parameter freezing (APF) [16]. However, APF conducts fine-grained parameter freezing making it impractical for accelerating training and is not considered to be compared to FedFreeze.

In the context of centralized training, **AutoFreeze** [98] is extensively utilized for layer freezing in traditional centralized training. This approach is adapted for FL as a baseline by implementing it on the server side. It is worth noting that Egeria [174] has demonstrated superior performance compared to **AutoFreeze**. However, Egeria relies on a ‘reference model’ to guide the analysis of layer convergence, which involves parallel training of a reference model on the server. This approach is impractical for FL since training data is distributed across devices making the simultaneous training of the reference model not possible. In summary, three baselines, namely **Vanilla FL**, **ALF**, and **AutoFreeze** are compared to FedFreeze.

5.4.2 End-to-end Performance

Table 5.3 summarizes the evaluation by presenting the highest test accuracy achieved and total training latency along with speedups compared to vanilla FL. The test accuracy curves for each baseline are shown in Figure 5.6.

Training LeNet on FMNIST: A LeNet model is trained using the FMNIST dataset on the Raspberry Pi testbed. The LeNet model only contains two convolutional layers and three fully-connected layers, making it computationally lightweight. However, it still requires up to 19480s (5.4 hrs) for training vanilla FL. Figure 5.6a and Figure 5.6b show the test accuracy curves of FedFreeze and other baselines on FMNIST for LeNet. All baselines, including FedFreeze, converge rapidly, reaching a relatively high accuracy after around 50 rounds. Specifically, for random initialization, 87.39%, 88.19%, 88.02%, and 87.7% accuracy, and for pre-trained initialization, 88.35%, 87.37%, 87.45%, and 87.87% accuracy is achieved for vanilla FL, ALF, AutoFreeze, and FedFreeze, respectively, at round 50. The rapid improvement in the test accuracy of the model provides the opportunity for

Table 5.3 Summary of evaluation. The highest accuracy achieved and total training time of FedFreeze and other baselines are reported. FedFreeze accelerates FL training by $1.07\times$ to $1.30\times$ while maintaining comparable highest accuracy.

Dataset	Testbed	Model	Initialization	Methods			
				Vanilla FL	ALF	AutoFreeze	FedFreeze
FMNIST	Raspberry Pi	LeNet	Random	89.57%	89.78%	88.75%	89.16%
				19480s (1 \times)	17426s (1.12 \times)	13821s (1.41 \times)	15031s (1.3 \times)
			Pre-trained	89.67%	89.61%	87.91%	89.21%
				18882s (1 \times)	18668s (1.01 \times)	12907s (1.46 \times)	15402s (1.23 \times)
CIFAR-10	Jetson Nano	VGG11	Random	82.6%	81.96%	76.26%	80.93%
				13365s (1 \times)	12972s (1.03 \times)	8795s (1.52 \times)	12517s (1.07 \times)
			Pre-trained	88.52%	87.92%	87.03%	87.96%
				13259s (1 \times)	13259s (1 \times)	9839s (1.35 \times)	11579s (1.15 \times)
CIFAR-100	Jetson Nano	ResNet12	Random	28.54%	29.28%	28.6%	28.96%
				4181s (1 \times)	4187s (1 \times)	3469s (1.21 \times)	3839s (1.09 \times)
			Pre-trained	36.19%	36.81%	35.3%	35.62%
				4159s (1 \times)	4083s (1.02 \times)	3577s (1.16 \times)	3633s (1.14 \times)

layers to be frozen, especially when it approaches the final accuracy by the 50th round. However, ALF has minimal acceleration ($1.12\times$ and $1.01\times$ on random and pre-trained initialization, respectively) by only adopting layer freezing in the late stages. AutoFreeze achieves a higher acceleration ($1.41\times$ and $1.46\times$ on random and pre-trained initialization, respectively) but has a relatively high accuracy loss of 0.82% and 1.76% compared to vanilla FL. FedFreeze balances between accuracy and speedup, with speedups of $1.3\times$ and $1.23\times$ while experiencing less than a 0.5% loss (0.41% and 0.46%) compared to vanilla FL.

Training VGG11 on CIFAR-10: A larger model, namely the VGG11 model that has a higher computational overhead than LeNet, is trained using the CIFAR-10 dataset on a testbed with GPU-enabled devices, namely Jetson Nanos. The VGG11 model has more layers (eight convolutional layers and three fully-connected layers), which makes layer freezing more complex. Figure 5.6c and Figure 5.6d show the test accuracy curves of FedFreeze and other baselines on CIFAR-10 for VGG11 with random and pre-trained initialization. It is noted that there is more variability in training due to the increased complexity of the model and dataset and more training rounds are required to achieve the highest accuracy. Only a marginal training time improvement is noted for ALF ($1.03\times$ speedup on random initialization and no speedup on pre-trained initialization). Moreover, ALF has an accuracy loss of around 0.6% for both random and pre-trained initialization even when applying layer freezing only in later training rounds. AutoFreeze suffers a significant loss when aggressively applying layer freezing in the early stages, with losses of 6.34% and 1.49% on random and pre-trained initialization, respectively, despite achieving speedups of $1.52\times$ and $1.35\times$. In contrast, FedFreeze can still achieve a $1.07\times$ speedup with a 1.67% accuracy loss when trained using random initialization, while with pre-trained initialization, FedFreeze has a $1.23\times$ speedup and a relatively small 0.56% accuracy loss compared to vanilla FL.

Training ResNet12 on CIFAR-100: Finally, a ResNet12 model is evaluated on the CIFAR-100 dataset using the Jetson Nano testbed. The residual architecture in the ResNet12 model makes the application of layer freezing more complex compared to a plain convolutional network (e.g., VGG11). Figure 5.6e and Figure 5.6f show the test accuracy curves of FedFreeze and other baselines on CIFAR-100 for ResNet12. It is observed that the training requires more rounds to converge for both random and pre-trained initialization similar to VGG11 on CIFAR-10. For ALF, a final accuracy of 29.28% and 36.81% is achieved using random and pre-trained initialization, respectively, but does not achieve any notable training acceleration ($1\times$ and $1.02\times$ on random and pre-trained initialization, respectively). Surprisingly, better final accuracy is achieved compared to vanilla FL in the pre-trained setting. There is conjecture that layer freezing in the late stages may stabilize aggregation in FL. AutoFreeze has an acceleration of $1.21\times$ on random initialization with an accuracy of

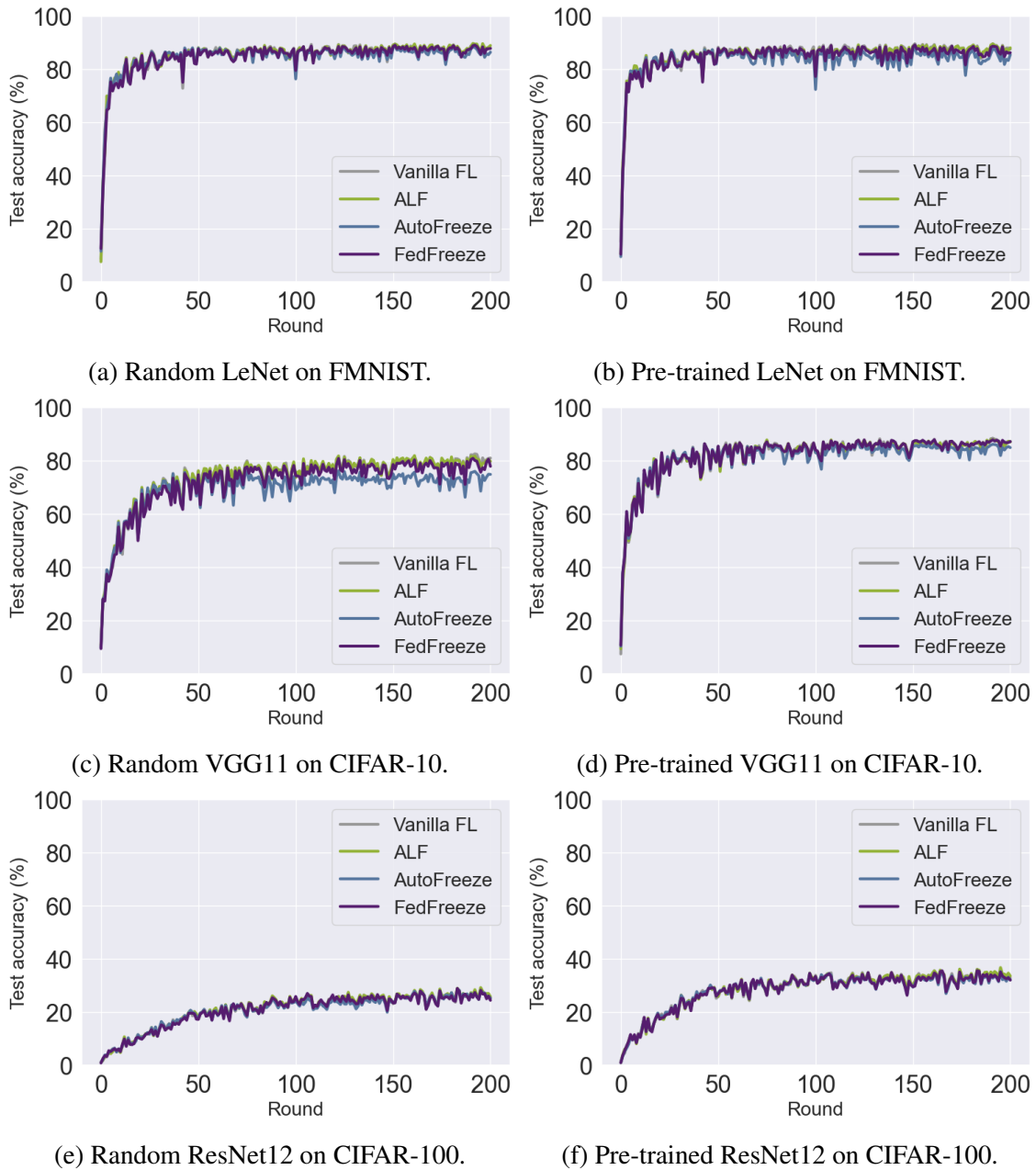


Fig. 5.6 Test accuracy curves of Vanilla FL, ALF, AutoFreeze and FedFreeze for the LeNet, VGG11 and ResNet12 models on the FMNIST, CIFAR-10 and CIFAR-100 datasets with random and pre-trained initialization.

28.6% and $1.16\times$ speedup with a 0.89% accuracy loss compared to vanilla FL. However, FedFreeze has superior performance achieving a comparable speedup of $1.09\times$ and $1.14\times$ with a higher final accuracy of 28.96% and 35.62%.

Summary: FedFreeze demonstrates competitive highest accuracy compared to vanilla FL while accelerating training up to $1.3\times$. Compared to other state-of-the-art baselines, FedFreeze is more robust when trained with both random and pre-trained initialization methods achieving a better trade-off between accuracy and speedup. While ALF offers only marginal training speedup and AutoFreeze results in significant accuracy loss.

5.4.3 Performance Breakdown

To take a closer look at training to understand the decisions made by FedFreeze and other baselines, the layer freezing choices are reported to provide valuable insights into the accuracy and speedup performance achieved by each method.

Global freezing decisions

Figure 5.7 shows the global freezing decisions of FedFreeze and other methods during the training of the three datasets and the three models with random and pre-trained initialization. The y-axis represents the number of frozen layers determined by the global freezer. For the LeNet, VGG11, and ResNet12 models, there are a total of 5, 11, and 12 layers, respectively. The freezing of the last layer in each model is excluded - if it is frozen, then it indicates that all layers are frozen, and training is stopped. Therefore, the maximum number of frozen layers is 4, 10, and 11 for the models.

ALF only makes freezing decisions in the later training stages and does not freeze any layers in the early stages, leading to a high final accuracy but inefficient training latency. ALF calculates the ‘perturbation effectiveness’ to analyze the convergence of a layer. ‘Perturbation effectiveness’ is a value between 0 and 1 that is uniform across layers. It starts at 1 and gradually decreases during training. ALF sets a pre-defined threshold to determine when a layer is frozen [105]. However, this results in layer freezing only in the later stages of training, limiting any training acceleration in the early stages. In some cases, as illustrated in Figure 5.7e and Figure 5.7d, ALF does not freeze any layers. The threshold used in ALF needs to be defined before training and can vary for different datasets, models, and initialization types, posing a challenge to generalization across various settings. Overall, ALF maintains the final accuracy but achieves marginal speedup for the training.

AutoFreeze aggressively freezes bottom layers in the early stages, resulting in training speedups, but the premature freezing of layers leads to a significant accuracy loss.

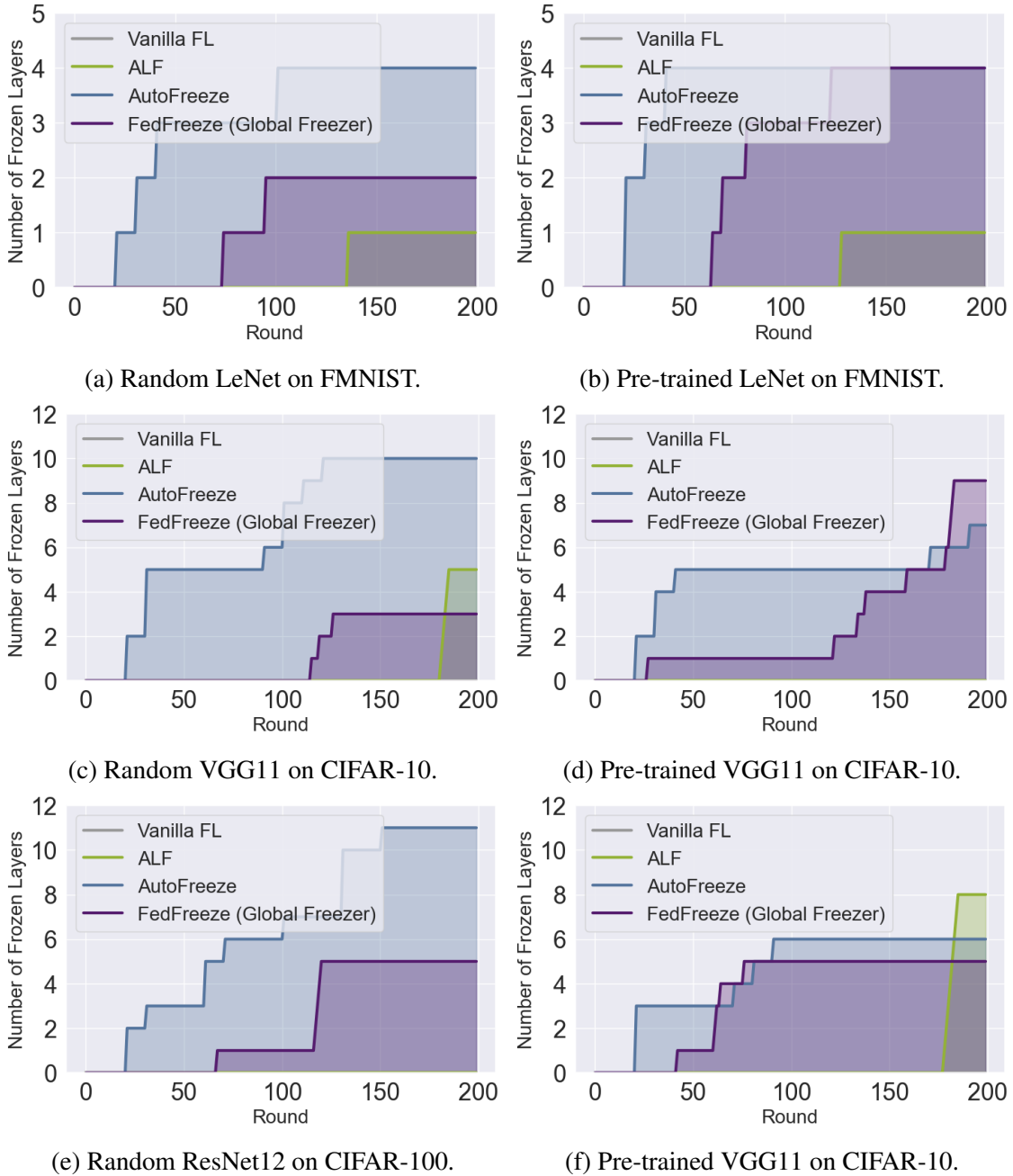


Fig. 5.7 The breakdown of global freezing decisions when training LeNet, VGG11 and ResNet12 on the FMNIST, CIFAR-10 and CIFAR-100 datasets for both random and pre-trained initialization.

AutoFreeze adopts a more aggressive approach to layer freezing by not requiring a layer to be converged. Specifically, it calculates the rate of change in gradient norm at fixed intervals and sorts all layers based on the rate [98]. As training progresses, the rate of change in the gradient norm decreases, allowing layers to be frozen accordingly. However, instead of enforcing a target rate threshold for each layer, which is unknown before training and may result in later-stage freezing like ALF, AutoFreeze adopts a more aggressive strategy. It freezes a layer if its rate of change in gradient norm falls within the N-percentile of all layers [98]. This relaxation enables AutoFreeze to freeze layers early, resulting in speedups. However, permanently freezing immature layers can lead to a significant accuracy loss. For instance, when training VGG11 using random initialization, which requires more training for each layer, AutoFreeze freezes the bottom layers (layers 1 to 5) before 50 rounds, resulting in a substantial accuracy loss (6.34%).

FedFreeze **achieves improved accuracy-speedup trade-off through adaptive global freezing decisions across random and pre-trained initializations.** The freezing decisions made by FedFreeze with the *Global Freezer* module are also reported. FedFreeze makes global freezing decisions based on two conditions discussed in Section 5.3.2. Compared to ALF, FedFreeze makes more aggressive freezing decisions by evaluating gradient changes. Therefore, FedFreeze requires less prior knowledge compared to ALF, as finding a uniform freezing criterion for all layers can be challenging, rendering it impractical. In comparison to AutoFreeze, FedFreeze only decides on permanent freezing of a layer in *Global Freezer*, minimizing the impact on final accuracy, while leaving early-stage freezing to the regularization-based *Local Freezer*. It achieves early freezing of the bottom layers while avoiding premature freezing of layers on the server. In addition, Figure 5.7 also demonstrates the ability of FedFreeze to adapt to pre-trained initialization, allowing for more aggressive layer freezing compared to random initialization. This is evident by the aggressive freezing in the pre-trained setting of FedFreeze, which is not the case in both ALF and AutoFreeze as shown in Figure 5.7. In summary, FedFreeze adjusts the freezing strategy based on initialization type by being more aggressive in the pre-trained context to accelerate convergence while preserving accuracy. With random initialization, a more conservative approach is adopted to keep more layers trainable early on to achieve high accuracy.

Local freezing decisions

Unlike ALF and AutoFreeze, FedFreeze also employs local freezing decisions made by the *Local Freezer* module. The *Local Freezer* adopts regularization-based layer freezing - layers are temporarily frozen for several iterations instead of permanently freezing. Figure 5.8 shows the local freezing decisions made by FedFreeze during the training. The absolute

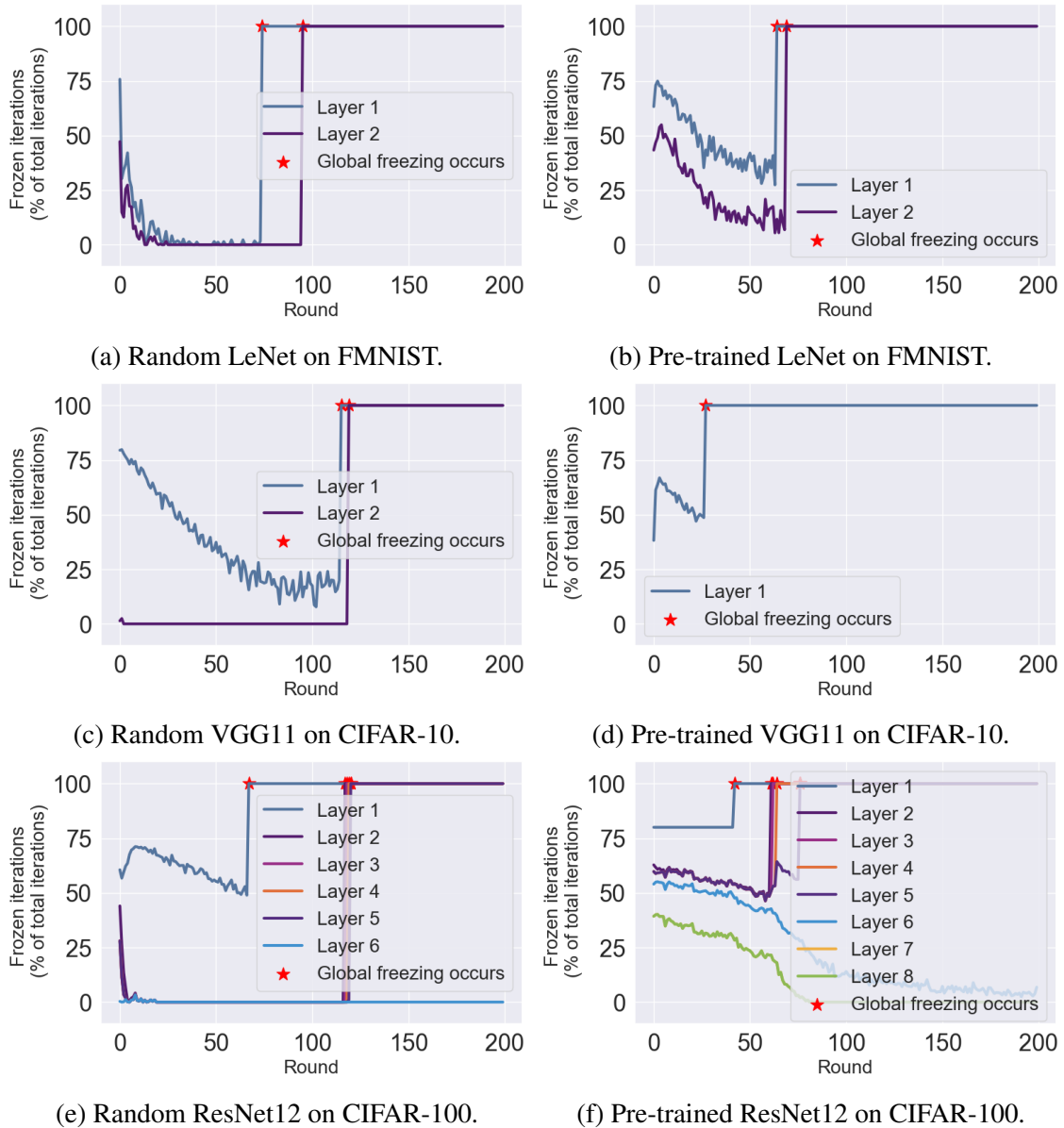


Fig. 5.8 Local freezing decisions made by FedFreeze when training LeNet, VGG11 and ResNet12 on the FMNIST, CIFAR-10 and CIFAR-100 datasets for both random and pre-trained initialization. The results show the percentage of local frozen iterations for each layer out of the total iterations provided by the *Local Freezer* of FedFreeze. Layers that have no local freezing are omitted from the figures. The results are the average of using 10 devices for each round.

value of iterations is normalized into percentages of the total iterations for each round. The results highlight the ability of FedFreeze to apply regularization-based layer freezing to specific layers based on their gradients. For instance, it is observed that bottom layers, such as layers 1 to 2 when training LeNet on FMNIST, layer 1 when training VGG on CIFAR-10, and layers 1 to 8 when training ResNet12 on CIFAR-100, undergo regularization-based layer freezing, while other layers are not frozen locally.

It also can be observed that the *Local Freezer* module in FedFreeze can adapt to different architectures and initialization methods. For example, when training VGG11, the *Local Freezer* module in FedFreeze only freezes Layer 1, as shown in Figure 5.8c and Figure 5.8d. In contrast, more layers are frozen locally when training ResNet12 on CIFAR-100, as depicted in Figure 5.8e and Figure 5.8f. In addition, it adapts to different initialization methods, as demonstrated by the more aggressive regularization-based layer freezing decisions when training with pre-trained initialization.

5.4.4 Discussion

This subsection examines the impact of hyperparameters and system overhead in FedFreeze.

Impact of Hyperparameters

FedFreeze evaluates the convergence of the global model after every aggregation on the server and employs two conditions to determine whether to freeze a layer. These conditions are based on the value of the gradient norm and the change in the gradient. As a result, two hyperparameters in the *Global Freezer* control the global freezing decision. In addition, in the *Local Freezer*, the hyper-parameter μ , which is the coefficient that controls the initial penalty of local regularization is considered.

Setting lower thresholds for the gradient norm and the change of the gradient in the *Global Freezer* will result in a more conservative global freezing policy. Similarly, a lower value of the hyper-parameter μ in the *Local Freezer* leads to a more aggressive strategy of local layer freezing. In the experiments, the hyperparameters are set separately for random and pre-trained initialization. For random initialization, a threshold value of 0.7 is set for the gradient norm compared to 0.9 for pre-trained initialization. Similarly, a higher coefficient of $\mu = 8$ was set for random initialization, while $\mu = 4$ was used for pre-trained initialization. This adjustment is necessary because pre-trained initialization will start with more mature parameters than random initialization. The threshold for the change of the gradient was set to 0.001 in both cases. In general, it was found in the experiments that the hyperparameters generalized across the datasets and models.

System Overhead

The system overhead in FedFreeze originates from two modules: the *Layer Regularizer* and the *Convergence Monitor*. The *Layer Regularizer* calculates the local layer freezing scheme for the *Local Freezer*, and the *Convergence Monitor* analyzes the gradient of layers for the *Global Freezer*.

The **Layer Regularizer** on the device needs to maintain a copy of the gradient changes after several iterations to calculate the layer-wise freezing scheme. This incurs a memory cost equivalent to the size of the model parameters. This is a practical cost as it only requires additional memory for the model parameters without including the activations. For the computational overhead, it was found that generating the local freezing scheme introduces up to 1.5% time overhead to the overall training on the device, which is negligible compared to the overall training time.

The **Convergence Monitor** on the server also needs to maintain an additional copy of gradient changes after the aggregation of each round to generate global freezing decisions. However, the memory cost and computational overhead are negligible, as it is performed on the server, which typically is not resource-constrained as the device.

5.5 Existing Solutions

This section first presents the existing work on layer freezing in both centralized and federated learning contexts. It then discusses two alternatives to layer freezing in FL, namely transfer learning and partial training. Finally, the differences between the above approaches and FedFreeze are compared and highlighted in Table 5.4.

Layer Freezing: Recent research that analyses post-hoc layer convergence has demonstrated that layers converge from the initial to final layers or in a bottom-up manner [137, 116]. This offers the opportunity for implementing layer freezing during training to accelerate training. Recent layer freezing frameworks have effectively accelerated training by gradually freezing layers. Specifically, the freezing decisions are determined by the analysis of either the convergence behavior of layer activations [174] or gradients [98, 42, 13]. However, since freezing is applied at the later training stages there are limited benefits during the early stages.

There are minimal efforts in applying layer freezing for accelerating FL training although FL has a relatively larger computational bottleneck than centralized training [119, 177]. Recent research investigated the application of fine-grained parameter freezing [16, 131] and disordered layer freezing [105] to address communication costs in FL. However, the use of unstructured or disordered freezing strategies offers no computational benefits for training.

Transfer Learning in FL: This is another approach to implement layer freezing in FL [188, 63]. In this approach, the initial DNN layers are initialized with pre-trained weights and subsequently frozen during the training [63]. The rationale is based on the observation that initial layers of a DNN are task-agnostic and pre-trained weights obtained from training other tasks can be effectively transferred for a given task. Recent studies on pre-training initialization have demonstrated that the accuracy gap between FL and centralized learning can be reduced [17, 120, 178]. However, there is a larger accuracy loss when more layers are frozen or if there is a significant domain shift between the target and pre-trained dataset, as highlighted in Section 5.1.

Partial Training in FL: This approach surmounts the challenge of limited computational resources on devices for FL and aims to reduce on-device computation using pruning [71, 83] or subnetwork training [58, 72, 3]. These methods can be seen as general parameter freezing techniques at a finer granularity of individual neurons or filters within a neural network instead of the entire layers. However, their reliance on heuristic freezing decisions, lacking rigorous convergence analysis, often results in a substantial accuracy loss [19].

How does FedFreeze differ from prior work? Table 5.4 provides a comparison of unstructured parameter freezing, partial training, transfer learning, vanilla layer freezing, and FedFreeze, all in the context of FL. While unstructured parameter freezing achieves high final accuracy, it does not reduce on-device computation. To obtain any computational benefits sparse libraries or custom hardware are required. Both structured partial training and transfer learning have a larger accuracy loss. Vanilla layer freezing offers no early-stage acceleration but only in the later stages of training. In contrast, FedFreeze enables early-stage layer freezing and has better learning efficiency while ensuring that the final accuracy is not compromised and the overall computation is reduced.

5.6 Summary

This chapter presents FedFreeze, an effective layer freezing framework that achieves early-stage acceleration and guarantees a high final accuracy. FedFreeze proposes an aggressive regularization-based layer freezing to enable early-stage layer freezing on the local devices for the first time and a conservative convergence-based layer freezing on the global server to maintain high final accuracy. The combination of the local and global layer freezing techniques enables FedFreeze to strike a good balance between accuracy and training speedup. FedFreeze is evaluated on a range of testbeds, datasets, models, and initialization methods. FedFreeze achieves similar early-stage speedup compared to state-of-the-art

Table 5.4 Comparing different freezing techniques and FedFreeze.

	Unstructured Parameter Freezing [16, 131]	Partial Training [58, 72, 3]	Transfer Learning [178, 17, 120]	Vanilla Layer Freezing[174, 98, 42, 105]	FedFreeze
Reducing device com- putation	✗	✓	✓	✓	✓
High final accuracy	✓	✗	✗	✓	✓
Early-stage accelera- tion	✗	✓	✓	✗	✓

early-stage layer freezing approaches while achieving a comparable final accuracy compared to vanilla FL and state-of-the-art accuracy-guaranteed layer freezing methods.

The use of FedFreeze within classic FL has been presented. Its utility extends beyond to other variants of FL, such as DPFL. In the next chapter, the adaptation of FedFreeze modules to a DPFL system and their integration with FedAdapt and EcoFed to collectively enhance the performance of DPFL training is explored.

Chapter 6

Integrating FedAdapt, EcoFed, and FedFreeze into EPFL

Chapter 3, Chapter 4, and Chapter 5 explored techniques for adaptively offloading DNN training to the server, efficiently reducing communication costs in DPFL training, and effectively accelerating on-device training using layer freezing, respectively. Three corresponding frameworks—FedAdapt, EcoFed, and FedFreeze were built upon the classic FL or vanilla DPFL system. These frameworks are designed to address the specific challenges of deploying FL at the edge, as discussed in the respective chapters. However, natural questions arise here: how do these specialized modules interact with one another? Can they be seamlessly integrated into a holistic framework for collectively enhancing the efficiency of classic FL? To address these questions, this chapter presents the integration of FedAdapt, EcoFed, and FedFreeze within a framework referred to as EPFL, efficient partitioning-based federated learning (EPFL).

EPFL incorporates the core functionalities of FedAdapt, EcoFed, and FedFreeze, thereby enabling it with the capabilities to adaptively offload DNN training, reduce partitioning-based communication overhead, and accelerate on-device training. These capabilities are achieved by interacting with the function interfaces of FedAdapt, EcoFed, and FedFreeze modules during the entire FL training process. In particular, EPFL optimizes classic FL training in the deployment and runtime phases. Moreover, the overall pipeline and redesign for integrating each module facilitate the collaborative work of the individual components.

Finally, the performance of EPFL on different datasets, models, and testbeds is evaluated and compared to classic FL and vanilla DPFL. The empirical results demonstrate a significant improvement in training latency and communication cost while achieving comparable accuracy to classic FL and vanilla DPFL. In addition, the aggregated performance of EPFL surpasses that of its individual parts, demonstrating the importance of the integration.

6.1 Motivation

FedAdapt, EcoFed, and FedFreeze proposed techniques aimed at facilitating adaptive offloading strategies, reducing communication costs, and accelerating training speeds of a DPFL system. These hold the potential to enhance the efficiency of FL in EC environments. However, whether these modules can be seamlessly integrated to provide collective benefits is unknown. Therefore, this chapter presents the EPFL framework, which integrates the modules from FedAdapt, EcoFed, and FedFreeze.

EPFL is not an easy and direct combination of the individual modules of FedAdapt, EcoFed, and FedFreeze but addresses technical hurdles in catering to two phases (i.e., deployment and runtime) in DPFL training. Specifically, the integrated system is capable of:

1. Effectively utilizing the DNN partitioning-based offloading technique to accelerate FL training and adapting it to mitigate device heterogeneity by generating different offloading strategies tailored to each device.
2. Efficiently reducing communication costs incurred in a DPFL system between devices and the server.
3. Optimizing the training overhead during FL rounds by implementing layer freezing techniques.

By incorporating the above, EPFL offers a holistic solution that maximizes FL training efficiency (e.g., latency and communication cost) in edge environments with resource-constrained devices while maintaining comparable learning performance (e.g., accuracy).

EPFL answers the following research problems:

RP1: How to integrate the modules into a coherent system and position them within the FL training pipeline? The implementation of the EPFL system must address how the modules within FedAdapt, EcoFed, and FedFreeze are integrated. Specifically, the EPFL system needs to determine the appropriate placement of the FedAdapt, EcoFed, and FedFreeze modules within the lifecycle of FL training. Furthermore, there is a need for an overarching pipeline (e.g. training rounds in EPFL) to define the workflow and steps of the EPFL system.

RP2: What redesign is required to facilitate the integration of the modules? Although FedAdapt, EcoFed, and FedFreeze are shown to enhance different aspects of a DPFL system, they require modifications to ensure that they can collaborate. For example, when FedAdapt implements different strategies for heterogeneous devices, the cached buffers required by EcoFed also need to be heterogeneous to adapt to varying PPs. In addition, when EcoFed applies device-side layer freezing, FedFreeze must further refine the layer

freezing decision and also take into account the adaptive partitioning scenarios provided by FedAdapt.

RP3: What will be the final performance of the integrated system? This final question relates to the performance of the EPFL framework, including learning efficiency (i.e., accuracy and convergence rate) and system efficiency (i.e., latency and communication cost). Specifically, what will be the end-to-end performance improvements of EPFL compared to classic FL and vanilla DPFL? Additionally, how does the performance of EPFL compare to its individual components, i.e., FedAdapt, EcoFed, and FedFreeze? To answer these questions a real-world prototype is designed and developed that is used for evaluating EPFL.

RP1 is addressed in Section 6.2, where the system architecture and training pipeline of EPFL are presented. In terms of **RP2**, the modifications to FedAdapt, EcoFed, and FedFreeze modules for integrating into an EPFL system are presented, including the *Clustering* module in FedAdapt, *Replay Buffer* module in EcoFed, and *Global Freezer* as well as *Local Freezer* in FedFreeze. Finally, in Section 6.4, an evaluation of EPFL across different datasets, models and prototypes is provided to compare against classic FL and vanilla DPFL.

6.2 Overview

This section outlines the overarching architecture and training pipeline of EPFL. Specifically, three distinct optimization strategies built upon vanilla DPFL are presented. Additionally, the control flow and training pipeline of EPFL for a single training round is considered.

6.2.1 System Architecture

Figure 6.1 illustrates the system architecture of EPFL built on a vanilla DPFL system. The EPFL framework incorporates three optimization strategies, i.e., offloading, communication, and freezing strategies. These strategies are adopted for solving the RQs in Chapter 1 supported by corresponding modules of FedAdapt, EcoFed, and FedFreeze.

In particular, by utilizing FedAdapt modules to generate adaptive PPs for devices, EPFL can tailor its PPs of models to adapt to the varying computation capabilities of different devices. For example, EPFL can partially offload model training from devices with limited resources or retain device-native training for other devices that can accommodate the entire model for training. In addressing the communication overhead incurred by model partitioning in a DPFL system, EPFL employs EcoFed modules to minimize communication costs. This is achieved by reducing the communication of activation and eliminating the transfer of the gradient. Finally, during training, EPFL monitors layer convergence on both the device-side

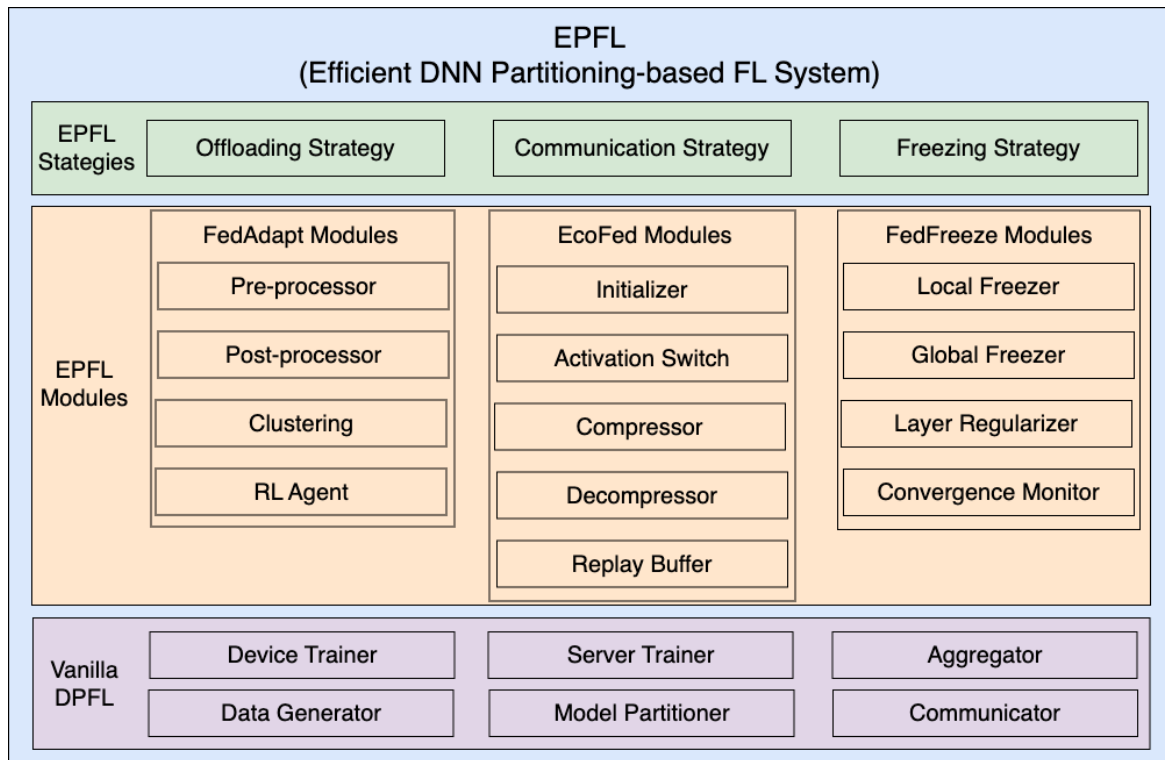


Fig. 6.1 The EPFL system architecture.

and server-side models. It effectively eliminates redundant computation by freezing layers that have already converged. In addition, EPFL adopts regularization-based layer freezing techniques to further accelerate training.

6.2.2 EPFL Training Mode

Positioning and control flow of modules in EPFL. Figure 6.2 illustrates the position of EPFL modules in an FL training round and the corresponding control flow on top of DPFL components. During each FL round, there are two phases: the deployment phase and the runtime phase. In the deployment phase, the FedAdapt modules in EPFL generate control information required for offloading, such as PPs for each device in a given round. The FedAdapt modules then call the device-side and server-side interfaces of the *DPFL Builder*. The builder deploys the DPFL system based on the offloading strategies generated by the FedAdapt modules in EPFL. Subsequently, the DPFL system is invoked to run after the deployment phase is completed. During the DPFL runtime phase, the EcoFed and FedFreeze modules within EPFL oversee the behavior of communication and layer convergence on both the devices and the server. Specifically, EcoFed provides control information regarding the transmission of activation and gradients, while the FedFreeze modules generate and update

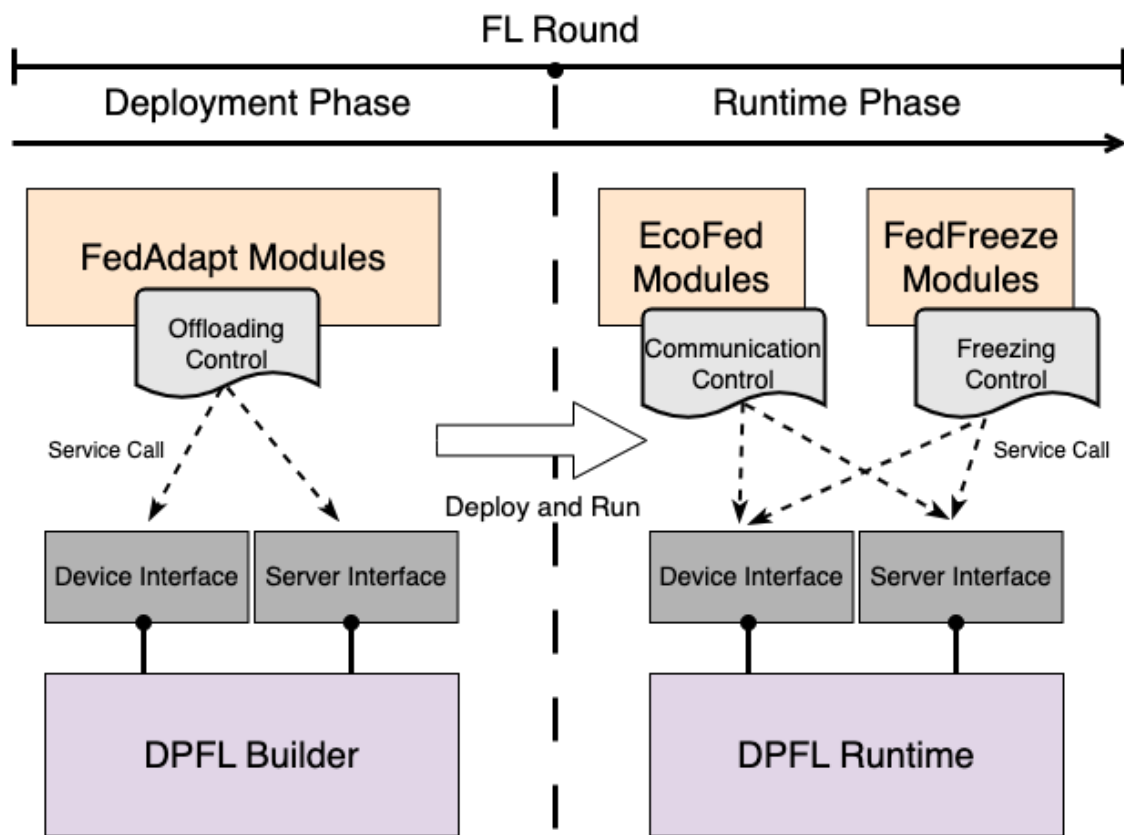


Fig. 6.2 The positioning and control flow of EPFL modules in a training round.

control information of layer freezing. Both the control information of communication and layer freezing are transmitted to the device-side and server-side interfaces exposed by the DPFL runtime engine that serves as the primary method for interaction between the EPFL framework and the DPFL system.

Training pipeline in EPFL. Figure 6.3 demonstrates the training pipeline of EPFL in a given round. Firstly, the FedAdapt modules collect observations from devices and assign PPs to each device. Different PPs result in EPFL selecting one of the two modes of training: the first is device-native training, which is similar to classic FL training, and the second is DPFL training.

Subsequently, the model will be initialized with pre-trained weights provided by the EcoFed modules. If offloading is not required, then device-native local training will be initiated, incorporating two types of layer freezing decisions (convergence-based and regularization-based layer freezing) from the FedFreeze modules. In the second EPFL training mode that uses DPFL training, EcoFed modules will first freeze the device-side model, and an *Activation Switch* in EcoFed will decide whether the server-side buffer needs to be updated by receiving compressed activations from devices. If not required, the server-side model will directly be trained using the buffer.

After multiple training epochs, whether employing device-native or DPFL training, the server aggregates models to produce the updated global model, concluding the training round.

6.3 Module Integration

To integrate the modules in FedAdapt, EcoFed, and FedFreeze into the EPFL framework, it is necessary to redesign the modules of FedAdapt, EcoFed, and FedFreeze.

6.3.1 Redesigning FedAdapt Modules for EPFL

In Chapter 3, FedAdapt clustered devices into three groups, including an additional group reserved for devices with limited network bandwidth, such as those operating on 3G (0.4/3 Mbps) networks. This design in FedAdapt was necessary since the additional communication costs of transferring activation and gradient could potentially offset the training acceleration gained from offloading computation from devices to the server. In the worst-case scenario, it might increase the overall latency. To avoid this, devices with low network bandwidth (e.g., 3G) are categorized into additional groups and were assigned distinct offloading strategies by the *RL Agent* in FedAdapt.

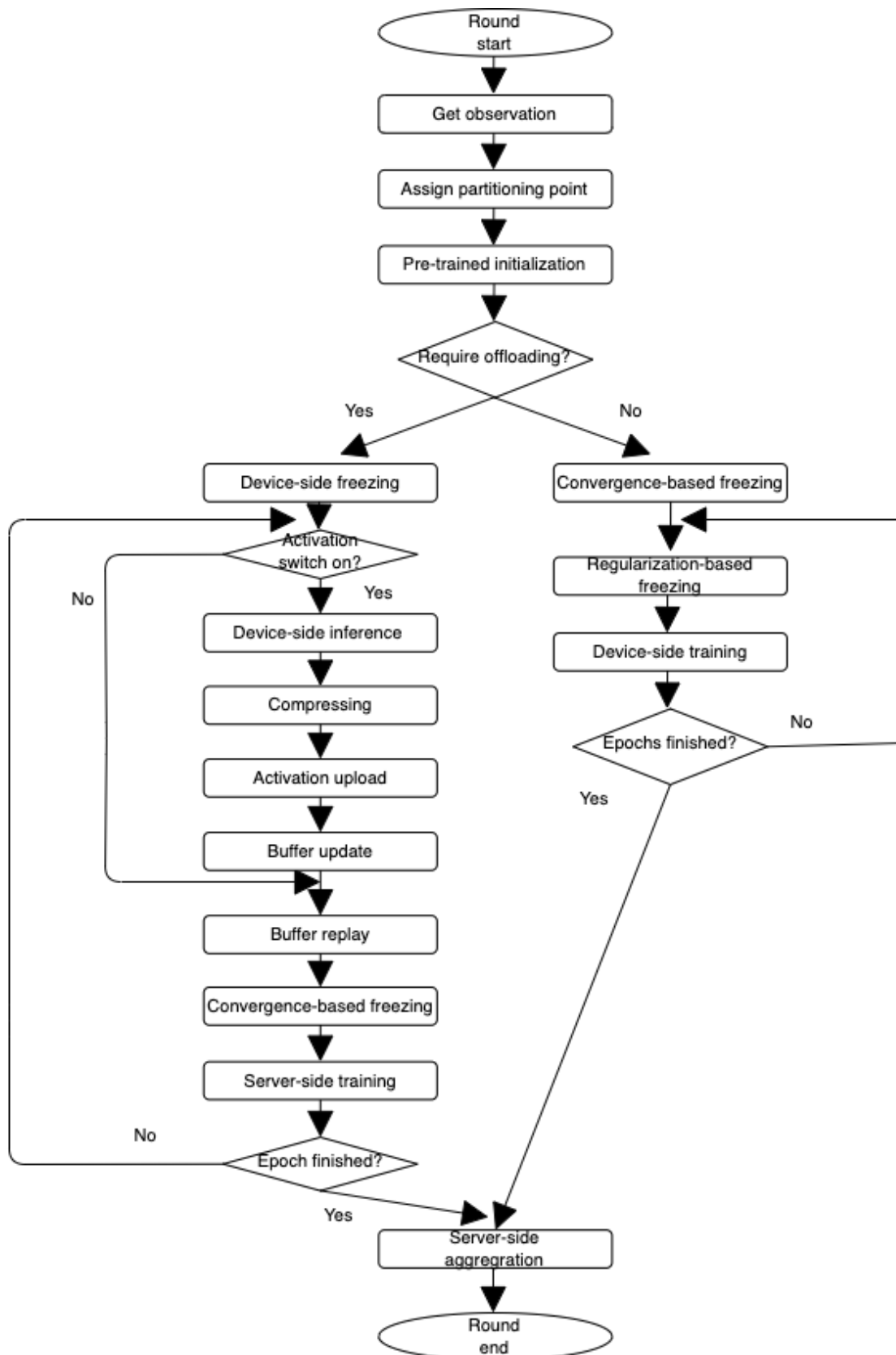


Fig. 6.3 The training pipeline of EPFL in a given training round.

However, this additional clustering group in FedAdapt can be eliminated when EcoFed is integrated into EPFL. This is because EcoFed can reduce the communication overhead by eliminating the need for transferring gradients and reducing the communication costs of activation. In addition, when using the activations cached in the *Replay Buffer* module on the server, EcoFed can eliminate the need for additional communication inherent in DPFL approaches. It has been demonstrated that even under 3G conditions, using EcoFed, DPFL approaches can achieve training acceleration by offloading training workloads to the server compared to device-native training of classic FL.

Given the lower communication overhead achieved by EcoFed, EPFL reduces the number of clustering groups in the *Clustering* module, eliminating the need for a dedicated group for the limited network bandwidth scenario. This design also simplifies the training process for the *RL agent*, as it reduces the dimension of both the input state (no need for observing network speed) and the action space (due to the reduced number of groups). Therefore, at the deployment phase of each round, FedAdapt collects the observations from different devices including the offloading ratio and training speed per batch from the last round and sends it to the *Pre-Processor* module. The *Pre-Processor* of FedAdapt normalizes observations into states and sends them to the *Clustering* module. The *Clustering* module assigns a group ID to each state and calls the *RL Agent* to determine the offloading ratio for the current round. Subsequently, the offloading ratio is translated into partition points and assigned to each device based on their group ID by the *Post-Processor*.

6.3.2 Redesigning EcoFed Modules for EPFL

In Chapter 4, the EcoFed modules did not consider the case of changing PPs. However, within the EPFL framework, where partition points may vary during DPFL training optimized by FedAdapt, a change in design is required. This is necessary for the *Replay Buffer* in EcoFed, as it needs to store activations generated at multiple PPs.

To address the above, the data structure of the cached buffer in *Replay Buffer* module on the server is redesigned and coupled with a corresponding sampling mechanism for generating training samples using the buffer. The *Replay Buffer* module uses the data structure of a dictionary to store the activations, namely *Buffer Dictionary*. Figure 6.4 shows the data structure of the *Buffer Dictionary* on the server. In addition to the standard buffer chunk, which is a batch of activations, EcoFed also stores the PP and device ID as the key of the *Buffer Dictionary*. This allows EcoFed to index the appropriate buffer from different PP and client IDs. The sampling mechanism in the *Replay Buffer* module is further modified. Firstly, the server will check if the device ID is available in the buffer and then verify the PP. If there is no match for the device ID or PP, a randomly sampled device ID is replaced

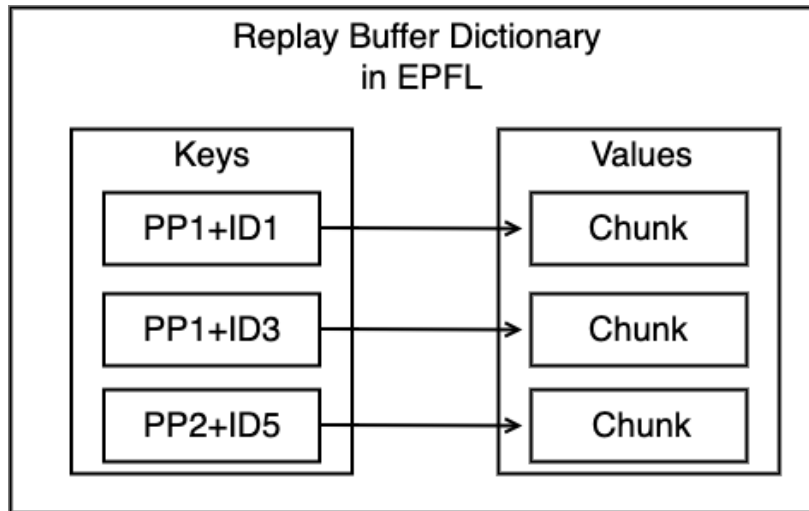


Fig. 6.4 The data structure of buffer dictionary of *Replay Buffer* in EPFL.

with the same PP as requested. This adjustment improves the hit ratio of the available cached buffers in the *Replay Buffer* module.

6.3.3 Redesigning FedFreeze Modules for EPFL

Chapter 5 delves into the development of FedFreeze within the context of device-native FL training. To integrate FedFreeze into a DPFL system and align it with the FedAdapt and EcoFed modules, the positioning of layer freezing of FedFreeze requires reconsideration.

In case a device trains entire models by itself (device-native training), FedFreeze can be directly applied to this device. This includes both convergence-based layer freezing generated by the *Global Freezer* and regularization-based layer freezing generated by the *Local Freezer*. In other cases, if offloading is adopted for a device, FedFreeze will only apply convergence-based layer freezing on the server-side model. This is because EcoFed has already frozen the device-side model with pre-trained initialization, rendering further regularization-based and convergence-based layer freezing unnecessary on the devices. In addition, freezing the device-side model imposes a regularization effect on the entire model, eliminating the need for conducting regularization-based layer freezing on the server-side model.

6.4 Evaluation

This section evaluates the final performance of EPFL framework across two testbeds, utilizing three datasets and their corresponding models. A comparative analysis between EPFL and

classic FL, as well as vanilla DPFL, is presented, focusing on final accuracy, training latency and communication cost as key evaluation metrics. In addition, the performance breakdown of EPFL, compared to the individual modules of FedAdapt, EcoFed, and FedFreeze, is considered.

6.4.1 Evaluation Setup

The details of the experimental setup, including the datasets and models used, as well as the training hyperparameters and testbed configurations are shown in this subsection.

Datasets and models. EPFL is evaluated with the same datasets used in FedFreeze, which are three datasets with distinct levels of difficulty, namely FMNIST [181], CIFAR-10 [79], and CIFAR-100 [79]. The same data partitioning method as in FedFreeze is adopted for each device to simulate a non-I.I.D. setting. In terms of model architecture, three popular CNNs: LeNet [81] (lightweight CNN), VGG11 [158] (plain CNN), and ResNet12 [54] (residual CNN) are trained on the FMNIST, CIFAR-10, and CIFAR-100 datasets, respectively. Please refer to Section 5.4 in Chapter 5 for the details.

FL training hyperparameters: For each FL round, 10 devices are uniformly sampled from a pool of 100 devices participating in a round of training. The most popular aggregation algorithm, i.e., standard FedAvg [111] is employed for global aggregation. The same data augmentation of horizontal flip and random crop is used for all experiments. The SGD optimizer with a constant learning rate of 0.01 is employed. A total number of 200 rounds is set for training on all datasets. For local training, the local epoch is set to 5 for all datasets.

DPFL training hyperparameters. For vanilla DPFL, Table 6.1 illustrates all PPs for three model architectures. For all experiments conducted on vanilla DPFL, PP1 is utilized for the static offloading position.

EPFL training hyperparameters. For the hyperparameters of the three modules in EPFL, FedAdapt selects PPs ranging from PP1 to PP4, along with device-native training. In the EcoFed module, pre-trained weights for VGG11 are obtained from the PyTorch Model Zoo (https://pytorch.org/serve/model_zoo.html), trained on the ImageNet dataset [27], while the pre-trained weights for LeNet and ResNet12 are trained on the Tiny-ImageNet dataset [80]. The period (ρ) for the *Replay Buffer* is set to 2 for all experiments. For FedFreeze, the two thresholds of the *Global Freezer* and the μ in the *Local Freezer* are adopted the same as in Section 5.4 in Chapter 5.

Testbed: To evaluate the system performances (i.e., training latency and communication cost), two prototypes are considered for evaluating EPFL. The first prototype is a Raspberry Pi cluster (representing low-end IoT devices). The second prototype is a Jetson Nano (high-end

Table 6.1 Different PPs used for evaluating LeNet, VGG11, and ResNet12. Convolution layers denoted as C followed by the no. of filters; filter size is 3×3 for all convolution layers except for the downsampling convolution where filter size is 1×1 ; Max Pooling layer is MP; Fully Connected layer is FC; and Residual Block is RB including two convolution layers, a max pooling layer, and downsampling convolution layers; number followed is the number of output channels.

PP	Device			Server		
	LeNet	VGG11	ResNet12	LeNet	VGG11	ResNet12
PP1	C6-MP	C64-MP	C64-MP	C16-MP- FC120- FC84- FC10	C128-MP- C256- C256-MP- C512- C512-MP- C512- C512-MP- FC512- FC512- FC10	C64-MP- RB64- RB128- RB256- FC10
PP2	C6-MP- C16-MP	C64-MP- C128-MP	C64-MP- C64-MP	FC120- FC84- FC10	C256- C256-MP- C512- C512-MP- C512- C512-MP- FC512- FC512- FC10	RB64- RB128- RB256- FC10
PP3	C6-MP- C16-MP- FC120	C64-MP- C128-MP- C256- C256-MP	C64-MP- C64-MP- RB64	FC84- FC10	C512- C512-MP- C512- C512-MP- FC512- FC512- FC10	RB128- RB256- FC10
PP4	C6-MP- C16-MP- FC120- FC84	C64-MP- C128-MP- C256- C256-MP- C512- C512-MP	C64-MP- C64-MP- RB64- RB128	FC10	C512- C512-MP- FC512- FC512- FC10	RB256- FC10

Table 6.2 Summary of evaluation. The highest accuracy achieved, total training time and total communication cost of FL, DPFL and EPFL are reported. EPFL accelerates training by $1.66\times$ to $5.57\times$ compared to FL and DPFL while maintaining a comparable accuracy. In addition, the communication cost of EPFL is up to $18.66\times$ lower than vanilla DPFL.

Dataset	Testbed	Model	Methods		
			FL	DPFL	EPFL
FMNIST	Raspberry Pi	LeNet	89.66%	89.59%	89.6%
			9336.04s	8763.83s	4493.74s
			0.93GB	52.95GB	2.84GB
CIFAR-10	Jetson Nano	VGG11	86.82%	86.57%	86.19%
			7527.29s	7765.03s	1798.5s
			145.36GB	610.71GB	38.26GB
CIFAR-100	Jetson Nano	ResNet12	32.96%	36.98%	37.32%
			2242.06s	7524.83s	1350.26s
			19.3GB	610.71GB	38.26GB

IoT device) cluster. Please refer to Section 5.4 in Chapter 5 for the details of the Raspberry Pi and Jetson Nano cluster used.

6.4.2 Comparison of Classic FL, Vanilla DPFL, and EPFL

Summary. Table 6.2 summarizes the evaluation by presenting the highest test accuracy achieved, total training latency and cumulative communication cost for FL, DPFL, and EPFL. In addition, Figure 6.5 shows the test accuracy curves achieved for each round. Figure 6.6 and Figure 6.7 show the minimum training latency and communication cost required to achieve a target accuracy, i.e., latency/communication cost vs. test accuracy curves for FL, DPFL, and EPFL.

Training LeNet on FMNIST. A LeNet model is trained using the FMNIST dataset on the Raspberry Pis testbed. The highest accuracy achieved by all three methods is similar, with 89.66%, 89.59%, and 89.6% for FL, DPFL, and EPFL, respectively. Figure 6.5a further demonstrates that the test accuracy curves of the three methods are almost overlapped, indicating similar convergence speeds during the training.

In terms of training latency, classic FL requires 9336.04s (2.59 hrs) to complete. DPFL, despite adopting computation offloading, requires 8763.84s (2.43 hrs), resulting in only a $1.06\times$ speedup in overall training latency. However, EPFL completes training in 4493.74s (1.24 hrs), representing a significant speedup of $2.08\times$ and $1.95\times$ compared to FL and DPFL, respectively. Figure 6.6a further illustrates the minimum latency to achieve the target accuracy for FL, DPFL, and EPFL when training LeNet on FMNIST. It is observed that

DPFL shows similar latency efficiency as FL in achieving target accuracy, particularly at higher target accuracies (above 80%). This is evident as the two curves nearly overlap at higher accuracy values. However, EPFL consistently demonstrates superior latency efficiency compared to FL and DPFL across a wide range of accuracy values.

In terms of communication cost, for lightweight models, such as LeNet, the communication cost of transferring model parameters is much lower than transferring intermediate results (activations and gradients) in a DPFL system. The overall communication costs are 0.93 GB, 52.95 GB, and 2.84 GB for FL, DPFL, and EPFL, respectively. EPFL significantly reduces the communication cost of transferring activations and gradients by $18.66\times$ compared to DPFL, due to the optimizations introduced in FedAdapt, EcoFed, and FedFreeze modules in EPFL. Figure 6.7a illustrates the communication cost vs target accuracy. It is observed that EPFL achieves communication efficiency an order of magnitude higher than DPFL across all target accuracy values.

Training VGG11 on CIFAR-10. When training a more complex model of VGG11 on CIFAR-10, EPFL still achieves comparable highest accuracy to FL and DPFL, with only a 0.63% and 0.38% difference in accuracy, respectively. In addition, Figure 6.5b further demonstrates the similar convergence curves for the three methods.

When training the VGG11 model on the CIFAR-10 dataset using the the Jetson Nano testbed, FL completes training in 7527.29s (2.09 hrs), despite VGG11 being more computation-intensive than LeNet. Surprisingly, DPFL has longer training time than FL, taking 7765.03s (2.15 hrs) due to the communication time incurred by transferring activations and gradients. However, EPFL completes training in 1798.5s (0.49 hrs), resulting in a speedup of $4.19\times$ and $4.32\times$ compared to FL and DPFL, respectively. Figure 6.6b further illustrates the latency efficiency of FL, DPFL, and EPFL. While FL and DPFL show similar performance, EPFL demonstrates higher latency efficiency across all target accuracy values.

In terms of communication cost, FL requires 145.36 GB for the entire training. DPFL has a communication overhead of a total of 610.71 GB. However, EPFL has a lower communication cost than both FL and DPFL with only 38.26 GB of communication for the entire training, resulting in a $3.8\times$ and $15.96\times$ reduction compared to FL and DPFL, respectively. Figure 6.7b further demonstrates the detailed communication costs required for various target accuracy. There is a noticeable efficiency gap between EPFL and FL/DPFL.

Training ResNet12 on CIFAR-100. Finally, the training of the ResNet12 model on the CIFAR-100 dataset using the Jetson Nano testbed is reported. EPFL even achieves better accuracy than FL and DPFL. This is because the layer freezing adopted in the FedFreeze module in EPFL improves the training in the late stages, resulting in higher accuracy than FL and DPFL. Figure 6.5b further reports the convergence curves of the three methods, where

all three methods still show similar convergence speeds. However, EPFL achieves higher final accuracy than FL and DPFL.

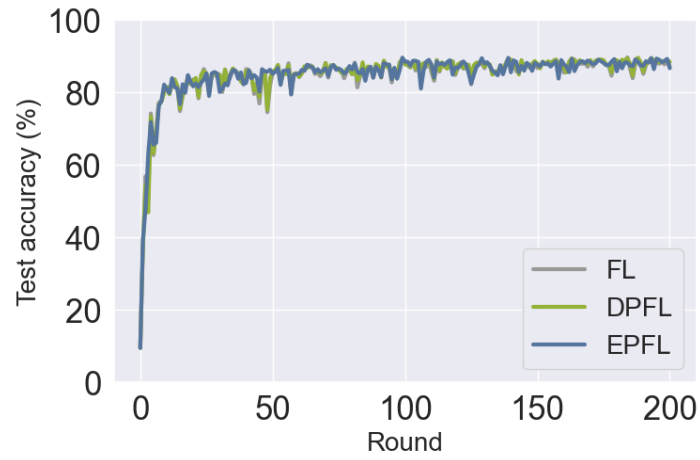
In terms of latency performance, FL completes the training within 2242.06s (0.62 hrs), whereas DPFL has a longer training time with 7524.83s (2.09 hrs). The communication cost of activations and gradients when training ResNet12 is similar to the communication cost of training VGG11 on CIFAR-10, but the computation time for ResNet12 is much smaller than for VGG11. This results in the reduction of FL training time. However, DPFL is limited by communication costs, resulting in longer training latency compared to FL. However, EPFL only requires 1350.26s (0.37 hrs) to complete the training, resulting in a speedup of $1.66\times$ and $5.57\times$ compared to FL and DPFL, respectively. Figure 6.6c further illustrates the latency efficiency of FL, DPFL, and EPFL, where the latency efficiency is in the descending order of EPFL, FL, and DPFL.

In terms of communication cost, DPFL has a higher communication cost than FL, with 610.71 GB compared to 19.3 GB for FL. In contrast, EPFL only requires 38.26 GB in total, despite also being a partition-based training method. This reduces communication by $15.96\times$ compared to vanilla DPFL. Figure 6.7c further illustrates the communication efficiency of FL, DPFL, and EPFL. It is obvious that EPFL and FL are more communication-efficient than DPFL. Although FL has a lower communication cost than EPFL, as the training progresses to the late stage (after reaching a target accuracy of 30%), EPFL shows similar communication efficiency to FL, as the curves of FL and EPFL converge. This is because the FedFreeze modules in EPFL reduce communication in the late stages of training when layers are frozen.

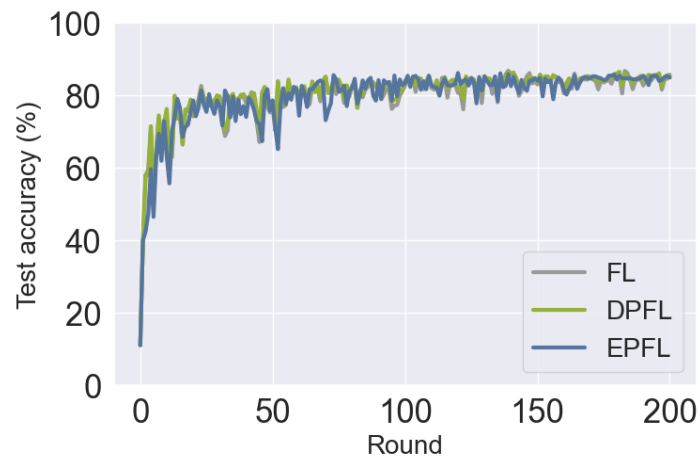
Summary. EPFL has similar final accuracy compared to classic FL and vanilla DPFL while significantly accelerating training. It is observed that EPFL can achieve up to a $4.19\times$ and $5.57\times$ speedup compared to FL and DPFL, respectively. Compared to DPFL which has a substantial communication cost, EPFL reduces the communication overhead by up to $18.66\times$. In general, the latency efficiency and communication efficiency are significantly improved in EPFL compared to FL and DPFL.

6.4.3 Performance Breakdown

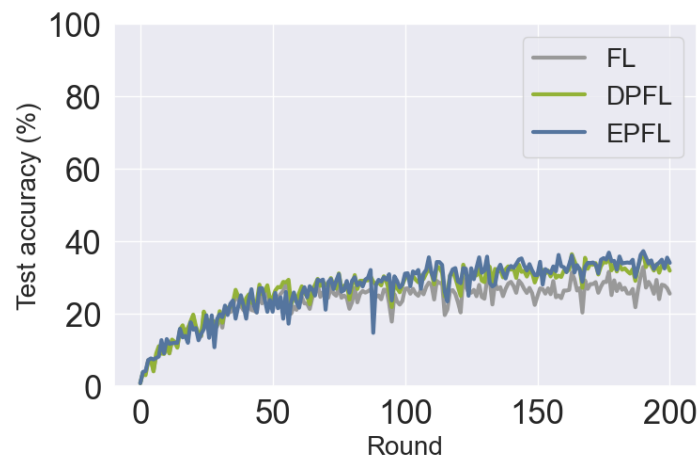
Comparison to FedAdapt only. When individually employing FedAdapt modules in EPFL, the output PPs for two testbeds are PP1 for the Raspberry Pi testbed and no offloading (device-native training) for the Jetson Nano testbed. As a result, individual FedAdapt modules have the same training configurations as vanilla DPFL for training the LeNet model on the FMNIST dataset and the same configurations as classic FL for training the VGG11 and ResNet12 models on CIFAR-10 and CIFAR-100 datasets, respectively.



(a) LeNet on FMNIST.

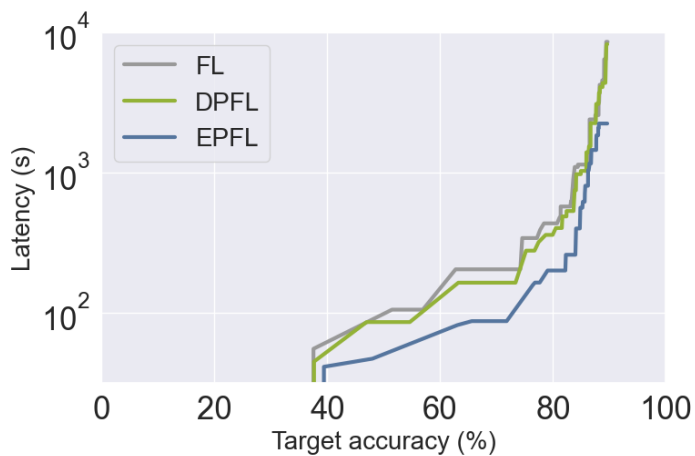


(b) VGG11 on CIFAR-10.

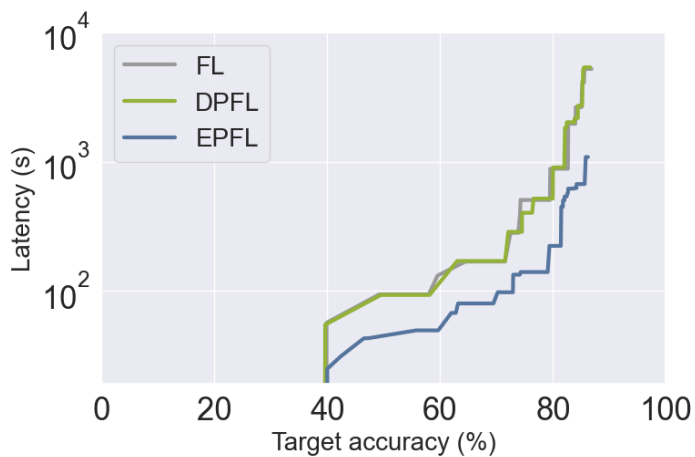


(c) ResNet12 on CIFAR-100.

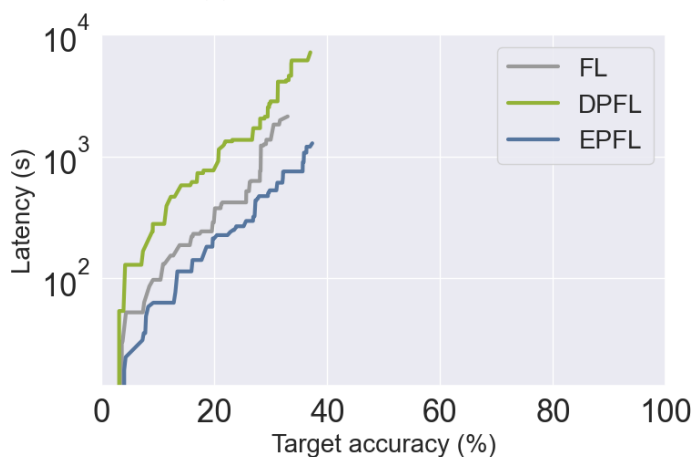
Fig. 6.5 Test accuracy curves of classic FL, vanilla DPFL and EPFL for the LeNet, VGG11 and ResNet12 models on the FMNIST, CIFAR-10 and CIFAR-100 datasets.



(a) LeNet on FMNIST.

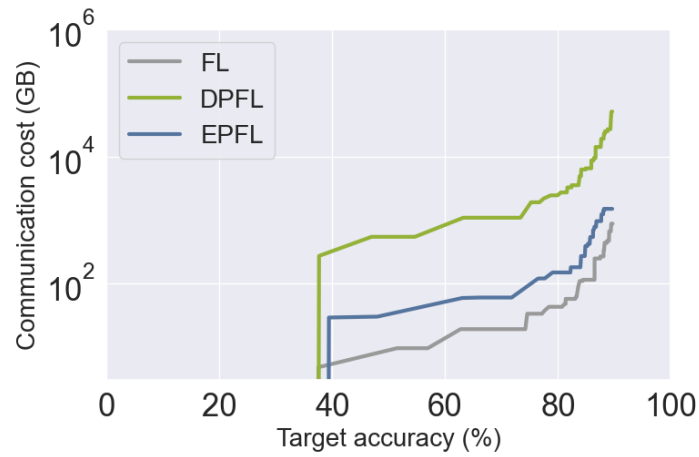


(b) VGG11 on CIFAR-10.

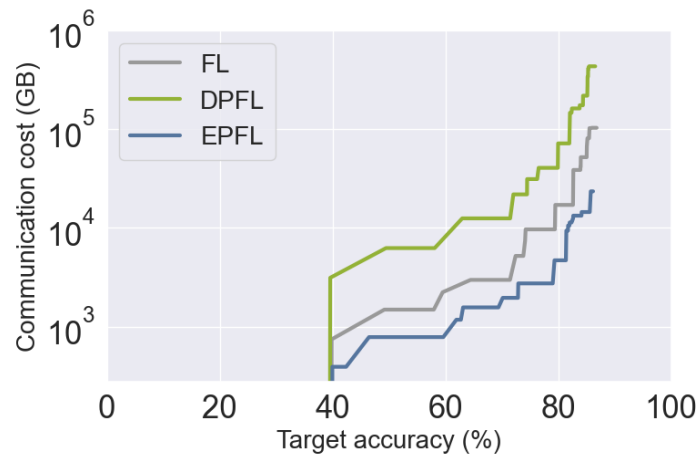


(c) ResNet12 on CIFAR-100.

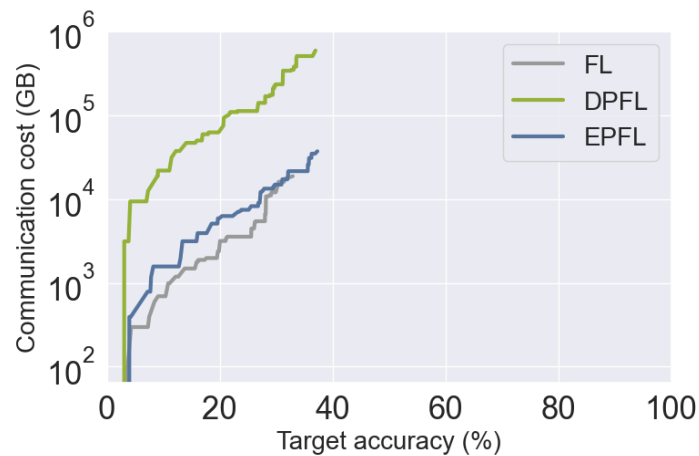
Fig. 6.6 Accumulated latency versus test accuracy of LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.



(a) LeNet on FMNIST.



(b) VGG11 on CIFAR-10.



(c) ResNet12 on CIFAR-100.

Fig. 6.7 Accumulated communication cost versus test accuracy of LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.

Table 6.3 Evaluation performances when only employing FedAdapt. The highest accuracy achieved, total training time and total communication cost of FedAdapt and EPFL are reported. It is worth noting that FedAdapt is equivalent to the vanilla DPFL training on the Raspberry Pi testbed and equivalent to the classic FL training on the Jetson Nano testbed.

Dataset	Testbed	Model	Methods	
			FedAdapt	EPFL
FMNIST	Raspberry Pi	LeNet	89.59%	89.6%
			8763.83s	4493.74s
			52.95GB	2.84GB
CIFAR-10	Jetson Nano	VGG11	86.82%	86.19%
			7527.29s	1798.5s
			145.36GB	38.26GB
CIFAR-100	Jetson Nano	ResNet12	32.96%	37.32%
			2242.06s	1350.26s
			19.3GB	38.26GB

Table 6.3 compares the performances of FedAdapt to EPFL. It shows that EPFL achieves comparable accuracy to FedAdapt on the FMNIST and CIFAR-10 dataset and better accuracy on CIFAR-100 dataset. However, EPFL further accelerates the training speed by $1.95\times$, $4.18\times$, and $1.66\times$ on the FMNIST, CIFAR-10, and CIFAR-100 datasets, respectively. This is due to the other two modules (i.e., EcoFed and FedFreeze), which reduce the communication time and accelerate on-device training, respectively. In terms of communication costs, EPFL reduces the overall communication costs by $18.64\times$ compared to FedAdapt on the Raspberry Pi testbed. On the Jetson Nano testbed, where FedAdapt executes classic FL training with lower communication overhead compared to vanilla DPFL training, EPFL still further reduces communication costs by $3.79\times$ when training a large DNN model like VGG11. When training a lightweight model like ResNet12, EPFL shows higher communication costs than FedAdapt but achieves higher accuracy and lower training time.

Benefits from employing EcoFed with FedAdapt. When EcoFed modules are adopted in EPFL, communication costs and training latency are further reduced. With the introduction of EcoFed, the output PP generated by FedAdapt changes from no offloading to PP1 on all three tasks, as partitioning and offloading more layers to the server provide the best training acceleration.

Figure 6.8 shows the overall training latency of FL, DPFL, FedAdapt, FedAdapt with EcoFed (denoted as FedAdapt + EcoFed), and EPFL (denoted as FedAdapt + EcoFed + FedFreeze).¹ When EcoFed is combined with FedAdapt, the training latency is reduced by $2\times$, $4.03\times$, and $1.58\times$ compared to FedAdapt for training LeNet on FMNIST, VGG11

¹For the latency and communication analysis of FL and DPFL, please refer to the discussion of Table 6.2.

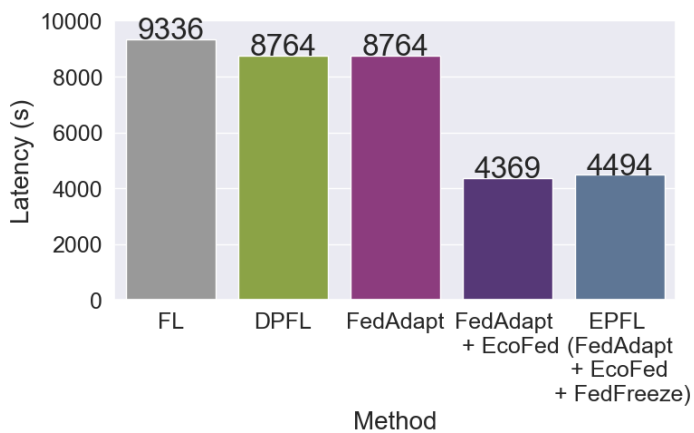
on CIFAR-10, and ResNet12 on CIFAR-100, respectively. In terms of communication costs, Figure 6.9 shows the overall communication costs of FL, DPFL, FedAdapt, FedAdapt + EcoFed, and EPFL. The overall communication costs are reduced by $17.7\times$ and $3.71\times$ compared to FedAdapt for training LeNet on FMNIST and ResNet12 on CIFAR-100, respectively. For training ResNet12 on CIFAR-100, FedAdapt + EcoFed reduces communication costs by $15.58\times$ compared to vanilla DPFL but still incurs $2.03\times$ more communication costs than FedAdapt.

Benefits of using FedFreeze with FedAdapt and EcoFed. When FedFreeze modules are adopted in EPFL, the training speed is further increased by freezing the gradient computation of convergent layers during training. In addition, the communication cost is further reduced as the frozen layers eliminate the need to communicate their parameters between devices and the server.

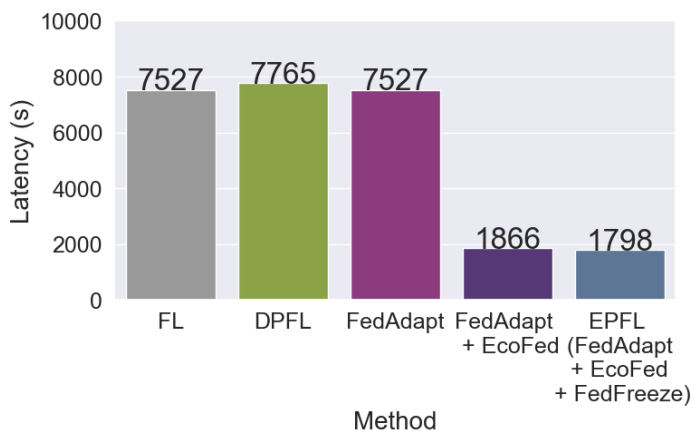
It is observed from Figure 6.8 that FedFreeze further reduces the overall training latency by $1.04\times$ and $1.05\times$ compared to EPFL without FedFreeze (FedAdapt + EcoFed) for training VGG11 on CIFAR-10 and ResNet12 on CIFAR-100, respectively. Training the small LeNet model on FMNIST using FedFreeze slightly increases the overall training latency by $1.02\times$ because the added overhead offsets the benefits of adopting FedFreeze. This is because the computation-intensive part of the LeNet model is on the device, which is already frozen by EcoFed. Consequently, the savings in computation latency by FedFreeze are marginal and offset by the added overhead of FedFreeze. This indicates that FedFreeze is more suitable for accelerating large DNN models when FedAdapt and EcoFed are used. In terms of communication costs, Figure 6.9 shows the overall communication costs are further reduced by $1.05\times$, $1.02\times$, and $1.02\times$ compared to EPFL without FedFreeze (FedAdapt + EcoFed) for training LeNet on FMNIST, VGG11 on CIFAR-10, and ResNet12 on CIFAR-100, respectively. It is worth noting that the performance gains are relatively smaller than the results reported in Chapter 5 since local freezing, particularly on the device side, is already considered in EcoFed.

6.5 Summary

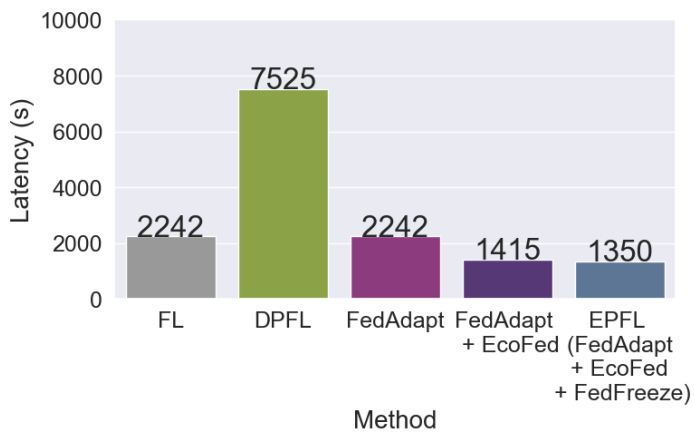
This chapter presents EPFL, a framework designed to enhance DPFL training. Specifically, EPFL incorporates three optimization strategies into vanilla DPFL: offloading strategy during deployment, communication strategy during runtime, and layer freezing strategy during runtime. In addition, EPFL refines the design of modules within FedAdapt, EcoFed, and FedFreeze to facilitate seamless integration and introduces a new training pipeline for integrating these modules. Evaluation of EPFL across two testbeds, three datasets, and two



(a) LeNet on FMNIST.

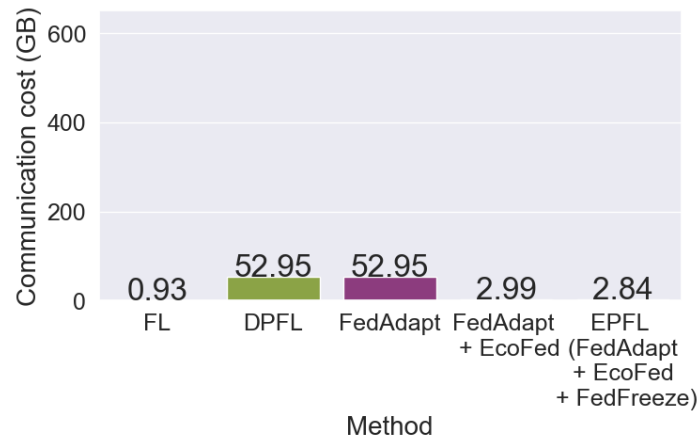


(b) VGG11 on CIFAR-10.

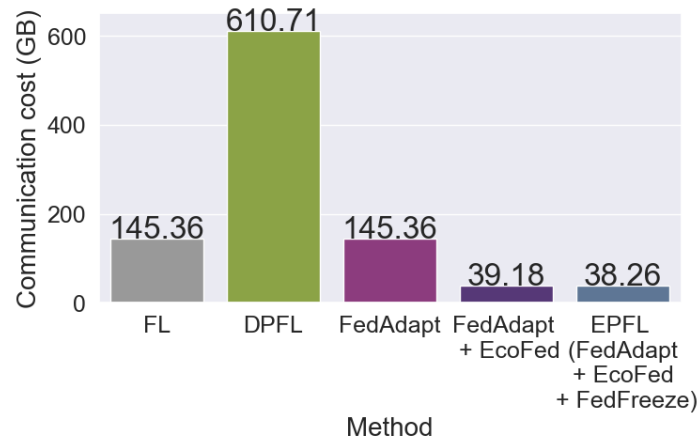


(c) ResNet12 on CIFAR-100.

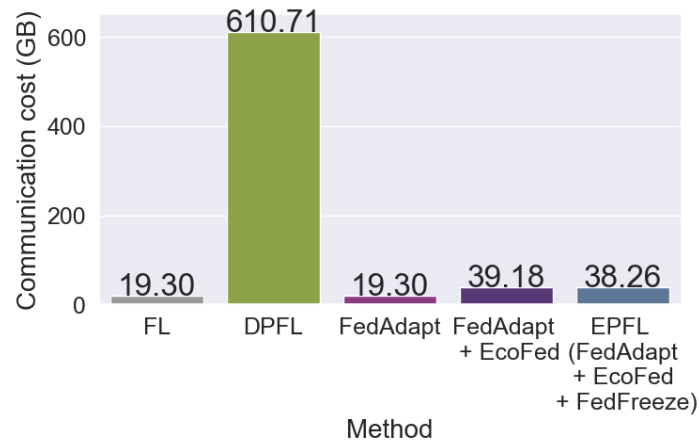
Fig. 6.8 Overall latency of different methods for training LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.



(a) LeNet on FMNIST.



(b) VGG11 on CIFAR-10.



(c) ResNet12 on CIFAR-100.

Fig. 6.9 Accumulated communication costs of different methods for training LeNet, VGG11 and ResNet12 on FMNIST, CIFAR-10 and CIFAR-100 datasets.

models demonstrate the efficiency enhancements compared to FL and DPFL. Compared to FL, EPFL can achieve up to a $4.19\times$ speedup in training time and a $3.8\times$ reduction in communication. Compared to vanilla DPFL, EPFL achieves up to $5.57\times$ acceleration in training time and $18.66\times$ reduction in communication overhead. In addition, the above improvements are achieved with an accuracy loss of no more than 0.63%.

Chapter 7

Conclusion

This section concludes the thesis by presenting the insights gleaned from the technical aspects of the work reported in the thesis and its broader impact. The limitations of the reported work and future directions for addressing them are further considered. Finally, the challenges considered by the thesis and the corresponding solutions developed are summarized.

7.1 Insights and Broader Impact

Three key observations from the research are summarized below. Then, the impact of the work reported in this thesis is highlighted, focusing on contributions to knowledge, research communities, and practical applications.

7.1.1 Insights

This thesis investigates DML systems, an underexplored avenue specifically for EC environments. Prior to the start of the work for this thesis (i.e., before October 2020), research primarily focused on cloud-based DML systems rather than those at the edge. The thesis has advanced the state-of-the-art and offers three valuable insights for building an efficient DML system at the edge:

1. **Device-side computation is usually the bottleneck in classic FL in an EC system.** In 2017, Google proposed the seminal work in FL [111], in which decentralized training on end-user devices was proposed. Following this, a series of works focused on further optimizing communication in FL, as the communication cost remains high when using large DNN models such as VGG11 [158] compared to the shallow neural networks that were used in the Google application. However, the research in this thesis

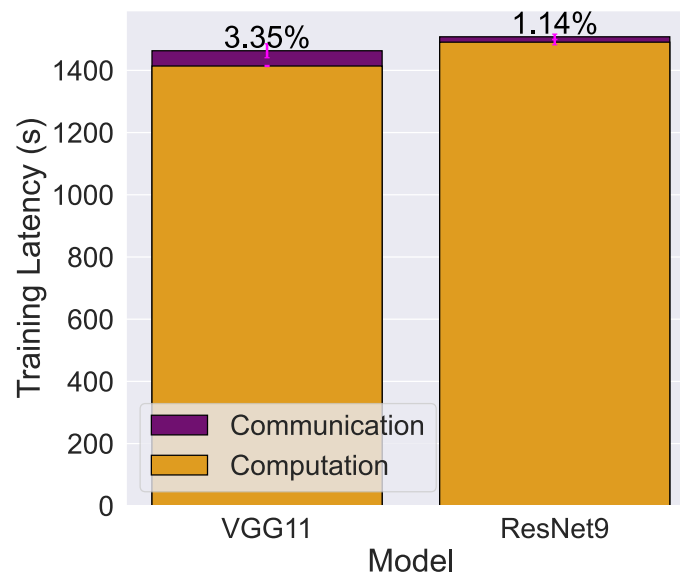


Fig. 7.1 The computation versus communication latency when training VGG11 and ResNet9 models on 5,000 samples of CIFAR-10 using a Raspberry Pi 4 single-board computer using a Wi-Fi network. The numbers above each bar represent the percentage of communication latency.

highlights that the device-side computation overhead is often the real bottleneck when deploying classic FL in an EC system, rather than communication. This is verified by investigating the computation and communication latencies of classic FL on IoT devices. For example, when training VGG11 and ResNet9 [54] models using the CIFAR-10 dataset on a Raspberry Pi 4 single-board computer with a Wi-Fi network bandwidth of 75 Mbps, the computation time accounts for approximately 96% and 98% of the total time, respectively, as shown in Figure 7.1. Similar observations are also found in recent research on FL for resource-constrained devices [66, 39, 171, 53]. This motivates the research undertaken and reported in Chapter 3 and Chapter 5 of this thesis for reducing the computational burden on devices when they train a DNN.

2. **Communication cost becomes a new bottleneck when DNN partitioning and offloading are adopted in FL.** DNN partitioning and offloading are useful techniques for reducing computation costs on devices in classic FL [165, 47, 53, 177]. This type of system is referred to as a DPFL system in this thesis. However, the communication costs of DPFL systems were not previously understood, specifically, the overhead incurred by transferring activations and gradients between devices and the server. It is important to reduce this overhead because it was observed that the communication cost in DPFL may exceed that of classic FL. Consequently, communication latency

becomes the new bottleneck in DPFL systems, accounting for up to 90% of the overall latency ¹. Therefore, novel techniques are proposed in Chapter 4 that are essential for building efficient DPFL systems at the edge by reducing the cost of communication.

3. **Early-stage layer freezing is essential for effectively accelerating on-device training.** Theoretically, layer freezing can eliminate the computation of gradients for the frozen layers in a DNN, which accelerates on-device training in FL. However, traditional convergence-based layer freezing often occurs only in the late training rounds of FL, leading to inadequate acceleration. For instance, when training the VGG11 [158] model on the CIFAR-10 [79] dataset using FL, the traditional convergence-based layer freezing only occurs after 181 rounds out of a total 200 ². Therefore, this thesis underscores the importance of implementing an early-stage layer freezing technique in FL training to accelerate on-device training.

7.1.2 Broader Impact

The impact of the work reported in this thesis is in three main areas: contributions to knowledge, research communities and practical applications.

Contribution to knowledge. The contribution to knowledge is in the following four avenues:

1. This thesis, for the first time, demonstrates that RL can be adopted to optimize DNN partitioning and implementing offloading can address the challenges originating from limited resources and device heterogeneity in FL. Specifically, the adaptive partitioning technique employed during DNN training that is proposed in Chapter 3 is the first of its kind in FL literature. Previous research has focused on using adaptive partitioning during inference, which is less challenging, than adopting it during training as presented in this thesis. Additionally, the concept of using clustering to simplify the training of the RL agents is proposed for the first time in this thesis.
2. This thesis introduces novel communication-reduction techniques for transferring activations and gradients in DPFL systems. Notably, the development of a replay buffer on the server represents a new approach to reducing the communication cost of activations in DPFL systems, a concept previously unexplored in FL literature. Additionally, an in-depth analysis of accuracy degradation in local loss-based methods is provided, an avenue in FL research that has not been covered in the prior work.

¹This is demonstrated in Figure 4.1 of Chapter 4

²This is highlighted in Figure 5.2a of Chapter 5

3. This thesis presents an innovative early-stage layer freezing technique designed to expedite on-device FL training. The proposed regularization-based layer freezing method is unique in providing the same regularization benefits as traditional loss-based methods while also speeding up the training process.
4. This thesis proposes a practical DPFL training pipeline that uniquely combines DNN offloading, pre-trained initialization, replay buffer, and layer freezing techniques. The pipeline can be deployed in a cloud-edge environment to accelerate training by considering both communication and computation trade-offs. Many existing works offer piecemeal techniques that cannot be combined in a meaningful way for practical use. The work in this thesis highlights the potential and limits of what is possible when combining essential techniques to accelerate FL. A comprehensive assessment of the pipeline is provided across various settings, including realistic testbeds, datasets, and models and compared against relevant baselines obtained from recent literature.

Contribution to research communities. This thesis contributes to the FL research community by undertaking a comprehensive investigation of the limits of existing FL techniques and offers effective solutions for building efficient FL systems at the edge. The holistic framework underpinned by a range of new techniques is a promising step towards extending FL training in EC environments. Experimental results demonstrate that the proposed variant of the DPFL training paradigm significantly improves the efficiency of classic FL.

It is to be noted that some of the proposed techniques, such as adaptive offloading, replay buffer, and pre-trained initialization, will not only benefit the FL community but also the broader DML community, as these techniques can be also applied to improve efficiency within a cloud data center. Additionally, the proposed regularization-based layer freezing technique contributes to the general research on on-device training in resource-constrained environments, such as TinyML [34].

Contribution to practical applications. Applying a classic FL approach to EC applications is challenging [192]. These challenges include impractical training times, significant communication overhead, training workloads that cannot be executed on devices, and the lack of practical frameworks as detailed in Chapter 1. The new variant of DPFL proposed in this thesis offers an effective solution to the challenges. This enables potential applications for building DML systems in areas such as smart homes [2], smart cities [126], and autonomous driving [91].

The individual techniques proposed in this thesis can be adopted for other DML applications as well. Specifically, adaptive offloading can enhance the performance of an edge-cloud DNN co-inference system, such as a streaming video analysis system [136].

The replay buffer can reduce communication costs for split learning systems commonly used in healthcare applications [133]. Additionally, the regularization-based layer freezing technique can accelerate the training of large DNN tasks, including large language models in the cloud [174].

The work reported in this thesis has resulted in four patents filed, as listed in Chapter 1 Section 1.5, by the funder Rakuten Mobile Inc. in Japan, which operates in the telecommunications sector. It is anticipated that the work presented in this thesis will find applications in wireless communication, such as vehicular networks, cross-border payment, and intelligent networks, as 5G and 6G technologies mature [121, 106].

7.2 Limitations and Future Work

While every effort has been made to create a technically sound and complete piece of work in the time frame that was available to complete the technical work, there are limitations to the work presented in the thesis. They are acknowledged and explored in this section while suggesting potential future research directions to address these constraints.

7.2.1 Lack of Practical Understanding of the Limits to Scalability

One limitation of the work is a lack of understanding of the limits to scalability of the proposed DPFL framework. The framework has been validated and evaluated in lab-based testbeds comprising tens of IoT devices. However, practical and operational constraints within which the research has been undertaken, such as institutional access policies or cost of equipment, have prevented testing in a real-world setting in which FL must execute, such as with hundreds or even thousands of devices.

The thesis also accounts for simulating a relatively large pool of devices in an FL setting. For instance, FL training is simulated with a total of 100 participating devices and 20 devices connected to the server in each training round, despite the fewer number of devices in the physical testbed. Training is performed on ten devices in parallel at a time. If there are more than ten devices, then training is run sequentially in two rounds. During each round, detailed logs of both training time and communication time are recorded. These logs are then analyzed for extracting information on the effectiveness of the proposed techniques.

Additionally, in the experiments conducted for this thesis, an edge server comprising a 2.5GHz dual-core Intel i7 CPU is used. This server is capable of handling partial offloading from 5 Raspberry Pis. Extrapolating from this setup, processing 100 devices would require approximately 20 similar CPUs to achieve the same level of performance. Such configurations

are common in modern cloud infrastructures. For instance, an AWS m5.12xlarge instance, which offers 64 vCPUs with 3.1 GHz Intel Xeon Platinum 8175M processors, can provide the necessary computational resources. Therefore, it provides a theoretical guarantee for the performance achieved by the proposed techniques.

Nonetheless, the simulation-based approach is limited in that it does not achieve the real-world scale of FL training. When directly applying the proposed DPFL framework to large deployments, several challenges need to be considered, including:

1. *The challenge of memory consumption on servers.* The proposed EPFL framework is built on a DPFL system, where each device needs to offload the server-side model parameters to the server. The server then maintains a copy of these parameters in its memory for independent training. As the number of connected devices increases, the server's memory consumption for maintaining these models also increases linearly. For example, a large DNN model such as VGG11 [158] requires nearly 0.5 GB for storing its parameters. Therefore, 100 instances of the model (i.e., 100 connected devices) would require 50 GB on the server, and 1000 instances would require 500 GB of memory on the server. State-of-the-art GPUs, such as A100 have a maximum memory of 80 GB [123], which could support the models of 100 devices, but to support 1000 devices would require at least 7 A100 GPUs.
2. *The challenge of running out of storage for the EcoFed buffer.* The replay buffer used in the EcoFed modules caches activations for each device to facilitate the training of server-side models. While this reduces the communication cost of activations, it increases the storage overhead on the server. The storage size for these activations is proportional to the total amount of data available from all devices. Additionally, the buffer size is influenced by the output size of the device-side model. For instance, the output size of the first convolutional layer of the VGG11 model on the CIFAR-10 dataset is nearly 64 KB per data sample. Consequently, storing activations for 50K CIFAR-10 samples would require approximately 3.2 GB. As the size of aggregated datasets from all devices increases, it is possible that the server may run out of storage. For example, the ImageNet [27] dataset, consisting of 1.2 million samples with an image resolution of 224×224 pixels, would require approximately 3.74 TB for storing the activations. Ideally, this data should be stored in memory for fast access during training. However, this becomes impractical with large datasets, necessitating storage on disk, which incurs I/O latency for loading data into memory from the disk.

Potential optimizations for improving scalability. As previously mentioned, with the increase in the number of devices and data samples, the system requirements, such as memory,

for building the proposed DPFL system need further optimization. Potential optimizations include:

1. *Optimizing the storage of server-side models.* One optimization approach for reducing the memory consumption of server-side models is to maintain a shared model for all devices instead of individual models for each device. This approach makes the storage of the server-side model independent of the number of devices, similar to the shared model in SL training. However, it is worth noting that this does not reduce the overall training workload.
2. *Optimization for the EcoFed buffer.* To address the challenge of potentially running out of storage for the EcoFed buffer, a maximum buffer size threshold needs to be set in the *Replay Buffer* module of EcoFed. This threshold will limit the activation cache available on the server. Alongside this limit, a corresponding cache update mechanism would be required that accounts for the buffer size. For fast access to the cached buffer during training, a two-tier buffer stored in both memory and disk can be implemented. Consequently, separate thresholds for memory and disk storage would need to be determined.

7.2.2 Need For Improving Resilience

Another limitation of this thesis is the need for improving resilience in the proposed DPFL framework. In real-world DML systems at the edge, robustness to device failures during the entire training process is crucial, as the stability of end devices is often unpredictable. Device failures can occur either between FL training rounds or during a training round. This thesis only considers the former and assumes that a device may or may not participate in a round of training due to its availability. However, if a device goes offline during an FL training round, it will result in the loss of training for that device and may even cause a cascade failure that affects the entire FL round. This is an important consideration as Google, for example, has reported a failure rate of 6% to 10% of devices in a round of FL training [11]. However, since resilience within FL was not the primary consideration of this thesis, this aspect has not been fully developed within the current work, but has dedicated efforts towards reducing computation and communication costs that are essential towards meeting initial deployment goals of FL at the edge.

Potential optimizations for enhancing resilience. To address the issue of device failures during a training round, several fault-tolerant techniques can be incorporated into the proposed system. Specifically, when a device goes offline during training, it is essential to adopt techniques such as creating and saving checkpoints [140] to store the updated gradients

on the server for final aggregation. This approach preserves the computational workload completed by the devices and can potentially improve the convergence rate of FL training.

7.2.3 Limited Consideration of Asynchronous Learning

In this thesis, the proposed DML training paradigm is built and evaluated using the synchronous FL training paradigm. Synchronous FL training involves a synchronization process of all devices at the end of each round before proceeding training. In other words, the server waits for all participating devices to finish their round of training before advancing to the next step of global aggregation. This synchronization ensures that each device starts the training from the same global model, which has been shown to accelerate the convergence rate [184]. However, in real-world FL at the edge, synchronization causes idle time for devices because devices that complete training must wait for the slower devices (or stragglers) [75, 171, 184]. Stragglers adversely impact the training time of each round and increase the idle time and resource utilization on other devices.

Potential optimizations to achieve asynchronous learning. To further improve the efficiency of the proposed DPFL system, an asynchronous mechanism can be incorporated that facilitates updates from the clients to be sent to the server and aggregated asynchronously. Specifically, devices can send their gradients to the server independently and continue to the next round of training immediately, rather than waiting for a global synchronization point. Additionally, the server can continuously update the global model as it receives updates from different devices. While fully asynchronous DPFL might be harder to implement as the model may require a substantial number of training rounds to converge, semi-synchronized approaches can improve resource efficiency on both devices and the server.

7.3 Summary

Applying DML on EC systems is challenging since EC has more stringent operational requirements for DML than on the cloud. In particular, this thesis has highlighted four challenges encountered when implementing a DML system at the edge, which are:

1. **Limited and heterogeneous resources.** In an EC system, devices typically have limited computing and memory resources, unlike their cloud-based counterparts. Consequently, applying DML to EC systems is difficult, as modern DML training on DNNs usually requires substantial resources. Additionally, the heterogeneity of devices, ranging from low-end IoT sensors to high-end GPU devices, further complicates achieving efficient DML on EC systems. Furthermore, the dynamic nature of EC environments,

such as fluctuating network bandwidth, poses additional challenges to the performance of DML systems.

2. **Substantial communication overhead.** Communication overhead poses another challenge in deploying DML at the edge. This issue arises from the substantial volume of intermediate data transferred during DML training, including model parameters, activations, and gradients. The volume of communication between devices and the server increases the overall latency of DML training and often requires ideal network conditions.
3. **Significant on-device training cost.** Modern DML training of DNNs requires intensive on-device computation. Unlike cloud-based DML systems that benefit from thousands of high-end GPU nodes, end-user devices and edge nodes in EC systems do not have the range of resources to manage large computational tasks. Therefore, the computational workload on end devices and edge nodes needs to be alleviated for efficient training.
4. **Lack of a holistic framework.** To enhance the overall efficiency of DML systems at the edge, it is crucial to collectively consider the aforementioned challenges within a single solution. This entails developing a robust framework that simultaneously addresses these challenges and integrates solutions within a unified approach. Therefore, there is a need to design a new training pipeline, interfaces, and modules to support the DML training paradigm at the edge.

To address the above challenges, this thesis proposes the following solutions:

1. **Adaptive partitioning and offloading techniques.** Chapter 3 of this thesis proposes adaptive partitioning and offloading techniques. The challenge posed by limited computational resources on devices is addressed by using a DNN offloading technique. Additionally, an RL agent optimizes partitioning points in DNN layers for heterogeneous devices and adapts to dynamic network conditions.
2. **Pre-trained initialization and replay buffer techniques.** Chapter 4 of this thesis proposes two communication reduction techniques specifically for DPFL: pre-trained initialization and a replay buffer. In particular, pre-trained initialization eliminates the need for gradient communication. Additionally, a replay buffer reduces the communication overhead of activation transmission. Moreover, a quantization technique is also adopted to further compress activation data.
3. **Early-stage and accuracy-guaranteed Layer freezing techniques.** Chapter 5 of this thesis proposes a dual-phase layer freezing approach to reduce on-device training

workloads. The core components of this approach are early-stage and accuracy-guaranteed layer freezing techniques, which balance the acceleration of training time and reduce accuracy loss. Regularization-based device-side layer freezing focuses on speeding up training in the early stages, while convergence-based server-side layer freezing ensures final accuracy.

4. **A holistic and integrated framework for DPFL training.** Chapter 6 of this thesis presents a holistic framework that integrates all the aforementioned techniques to achieve a highly efficient DPFL system at the edge. A high-level pipeline is proposed to enable the deployment of DPFL at the edge.

As a final note, this thesis proposes four sets of solutions to address the challenges of developing DML on EC systems. Notably, a holistic framework is developed that offers an integrated solution to significantly reduce computation and communication costs. Based on this framework, a new type of DML system has been implemented and demonstrated on multiple lab-based testbeds. Evaluations conducted across various settings show a thorough analysis of the proposed system and demonstrate significant improvements compared to classic FL and existing DML systems at the edge. In short, the thesis offers a promising solution that opens a further avenue for developing and deploying efficient DML systems at the edge.

References

- [1] Sharif Abuadbbba, Kyuyeon Kim, Minki Kim, Chandra Thapa, Seyit A Camtepe, Yansong Gao, Hyounghick Kim, and Surya Nepal. Can we use split learning on 1D CNN models for privacy preserving training? In *Proceedings of the ACM Asia conference on computer and communications security*, pages 305–318, 2020. doi: 10.1145/3320269.3384740.
- [2] Ulrich Matchi Aïvodji, Sébastien Gambs, and Alexandre Martin. IOTFLA: A Secured and Privacy-Preserving Smart Home Architecture Implementing Federated Learning. In *IEEE Security and Privacy Workshop*, pages 175–180, 2019. doi: 10.1109/SPW.2019.00041.
- [3] Samiul Alam, Luyang Liu, Ming Yan, and Mi Zhang. FedRolex: Model-Heterogeneous Federated Learning with Rolling Sub-Model Extraction. *Advances in Neural Information Processing Systems*, 35:29677–29690, 2022.
- [4] Zaiwar Ali, Lei Jiao, Thar Baker, Ghulam Abbas, Ziaul Haq Abbas, and Sadia Khaf. A Deep Learning Approach for Energy Efficient Computational Offloading in Mobile Edge Computing. *IEEE Access*, 7:149623–149633, 2019. doi: 10.1109/ACCESS.2019.2947053.
- [5] HamidReza Asaadi, Dounia Khaldi, and Barbara Chapman. A Comparative Survey of the HPC and Big Data Paradigms: Analysis and Experiments. In *IEEE International Conference on Cluster Computing*, pages 423–432, 2016. doi: 10.1109/cluster.2016.21.
- [6] Syreen Banabilah, Moayad Aloqaily, Eitaa Alsayed, Nida Malik, and Yaser Jararweh. Federated Learning Review: Fundamentals, Enabling Technologies, and Future Applications. *Information Processing and Management*, 59(6):103061, 2022. doi: 10.1016/j.ipm.2022.103061.
- [7] Chaim Baskin, Natan Liss, Eli Schwartz, Evgenii Zheltonozhskii, Raja Giryes, Alex M Bronstein, and Avi Mendelson. UNIQ: Uniform Noise Injection for Non-Uniform Quantization of Neural Networks. *ACM Transactions on Computer Systems*, 37(1-4): 1–15, 2021. doi: 10.1145/3444943.
- [8] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Greedy Layerwise Learning Can Scale To Imagenet. In *International Conference on Machine Learning*, pages 583–593, 2019.

- [9] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Decoupled Greedy Learning of CNNs. In *International Conference on Machine Learning*, pages 736–745, 2020.
- [10] Enrique Tomás Martínez Beltrán, Mario Quiles Pérez, Pedro Miguel Sánchez Sánchez, Sergio López Bernal, G r me Bovet, Manuel Gil P rez, Gregorio Mart nez P rez, and Alberto Huertas Celdr n. Decentralized Federated Learning: Fundamentals, State of the Art, Frameworks, Trends, and Challenges. *IEEE Communications Surveys & Tutorials*, 2023. doi: 10.1109/COMST.2023.3315746.
- [11] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Kone n , Stefano Mazzocchi, H Brendan McMahan, et al. Towards Federated Learning at Scale: System Design. In *Proceedings of Machine Learning and Systems*, pages 374–388, 2019.
- [12] Christopher Briggs, Zhong Fan, and Peter Andras. *A Review of Privacy Preserving Federated Learning for Private IoT Analytics*, pages 21–50. Springer International Publishing, 2021. doi: 10.1007/978-3-030-70604-3_2.
- [13] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. FreezeOut: Accelerate Training by Progressively Freezing Layers. In *Advances in Neural Information Processing Systems Workshop on Optimization for Machine Learning*, 2017.
- [14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.
- [15] Jie Cao, Lanyu Xu, Raef Abdallah, and Weisong Shi. EdgeOS_H: A Home Operating System for Internet of Everything. In *IEEE International Conference on Distributed Computing Systems*, pages 1756–1764, 2017. doi: 10.1109/ICDCS.2017.325.
- [16] Chen Chen, Hong Xu, Wei Wang, Baochun Li, Bo Li, Li Chen, and Gong Zhang. Communication-Efficient Federated Learning with Adaptive Parameter Freezing. In *IEEE International Conference on Distributed Computing Systems*, pages 1–11, 2021. doi: 10.1109/ICDCS51616.2021.00010.
- [17] Hong-You Chen, Cheng-Hao Tu, Ziwei Li, Han-Wei Shen, and Wei-Lun Chao. On the Importance and Applicability of Pre-Training for Federated Learning. In *International Conference on Learning Representations*, 2023.
- [18] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. Asynchronous Online Federated Learning for Edge Devices with Non-IID Data. In *IEEE International Conference on Big Data*, pages 15–24, 2020. doi: 10.1109/bigdata50022.2020.9378161.
- [19] Gary Cheng, Zachary Charles, Zachary Garrett, and Keith Rush. Does Federated Dropout Actually Work? In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3387–3395, 2022. doi: 10.1109/cvprw56347.2022.00382.

- [20] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Dimitrios Papadopoulos, and Qiang Yang. Secureboost: A Lossless Federated Learning Framework. *IEEE Intelligent Systems*, 36(6):87–98, 2021. doi: 10.1109/mis.2021.3082561.
- [21] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A Survey of Model Compression and Acceleration for Deep Neural Networks. *IEEE Signal Processing Magazine*, 35(1):126–136, 2018. doi: 10.1109/MSP.2017.2765695.
- [22] Li Chuan. OpenAI’s GPT-3 Language Model: A Technical Overview. Available online: <https://lambdalabs.com/blog/demystifying-gpt-3>, Accessed 06/19/2024.
- [23] Intersoft Consulting. General Data Protection Regulation (GDPR). Available online: <https://gdpr-info.eu/>, Accessed 06/19/2024.
- [24] Anirban Das and Thomas Brunschwiler. Privacy is What We Care About: Experimental Investigation of Federated Learning on Edge Devices. In *International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, pages 39–42, 2019. doi: 10.1145/3363347.3363365.
- [25] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed Deep Learning Using Synchronous Stochastic Gradient Descent. *arXiv preprint arXiv:1602.06709*, 2016. doi: 10.48550/arXiv.1602.06709.
- [26] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A Tutorial on the Cross-Entropy Method. *Annals of Operations Research*, 134:19–67, 2005. doi: 10.1007/s10479-005-5724-z.
- [27] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-Scale Hierarchical Image Database. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/cvprw.2009.5206848.
- [28] Li Deng, Dong Yu, et al. Deep Learning: Methods and Applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014. doi: 10.1561/9781601988157.
- [29] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y Zomaya. Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020. doi: 10.1109/JIOT.2020.2984887.
- [30] Xiumei Deng, Jun Li, Chuan Ma, Kang Wei, Long Shi, Ming Ding, and Wen Chen. Low-Latency Federated Learning With DNN Partition in Distributed Industrial IoT Networks. *IEEE Journal on Selected Areas in Communications*, 41(3):755–775, 2022. doi: 10.1109/JSAC.2022.3229436.
- [31] Sagar Dhakal, Saurav Prakash, Yair Yona, Shilpa Talwar, and Nageen Himayat. Coded Federated Learning. In *IEEE Globecom Workshops*, pages 1–6, 2019. doi: 10.1109/gcwkshps45667.2019.9024521.

- [32] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. A Survey of On-Device Machine Learning: An Algorithms and Learning Theory Perspective. *ACM Transactions on Internet of Things*, 2(3):1–49, 2021. doi: 10.1145/3450494.
- [33] Shi Dong, Ping Wang, and Khushnood Abbas. A Survey on Deep Learning and Its Applications. *Computer Science Review*, 40:100379, 2021. doi: 10.1016/j.cosrev.2021.100379.
- [34] Lachit Dutta and Swapna Bharali. TinyML Meets IoT: A Comprehensive Survey. *Internet of Things*, 16:100461, 2021. doi: 10.1016/j.iot.2021.100461.
- [35] Anis Elgabli, Jihong Park, Amrit S Bedi, Mehdi Bennis, and Vaneet Aggarwal. GADMM: Fast and Communication Efficient Framework for Distributed Machine Learning. *Journal of Machine Learning Research*, 21(76):1–39, 2020.
- [36] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 1–6, 2019. doi: 10.1109/islped.2019.8824955.
- [37] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Rémi Gribonval, Herve Jegou, and Armand Joulin. Training with Quantization Noise for Extreme Model Compression. In *International Conference on Learning Representations*, 2021.
- [38] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*, 2019.
- [39] Yansong Gao, Minki Kim, Sharif Abuadba, Yeonjae Kim, Chandra Thapa, Kyuyeon Kim, Seyit A Camtepe, Hyoungshick Kim, and Surya Nepal. End-to-End Evaluation of Federated Learning and Split Learning for Internet of Things. In *International Symposium on Reliable Distributed Systems*, pages 91–100, 2020. doi: 10.1109/SRDS51746.2020.00017.
- [40] Álvaro López García, Jesus Marco De Lucas, Marica Antonacci, Wolfgang Zu Castell, Mario David, Marcus Hardt, Lara Lloret Iglesias, Germán Moltó, Marcin Plociennik, Viet Tran, et al. A Cloud-Based Framework for Machine Learning Workloads and Applications. *IEEE Access*, 8:18681–18692, 2020. doi: 10.1109/ACCESS.2020.2964386.
- [41] Anthony Gillioz, Jacky Casas, Elena Mugellini, and Omar Abou Khaled. Overview of the Transformer-Based Models for NLP Tasks. In *IEEE Conference on Computer Science and Information Systems*, pages 179–183, 2020. doi: 10.15439/2020F20.
- [42] Kelam Goutam, S Balasubramanian, Darshan Gera, and R Raghunatha Sarma. Layer-Out: Freezing Layers in Deep Neural Networks. *SN Computer Science*, 1(5):295, 2020.

- [43] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training Imagenet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017. doi: 10.48550/arXiv.1706.02677.
- [44] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates. In *IEEE International Conference on Robotics and Automation*, pages 3389–3396, 2017. doi: 10.1109/icra.2017.7989385.
- [45] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. SpotTune: Transfer Learning through Adaptive Fine-Tuning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4805–4814, 2019. doi: 10.1109/cvpr.2019.00494.
- [46] Ido Hakimi, Saar Barkai, Moshe Gabel, and Assaf Schuster. Taming Momentum in a Distributed Asynchronous Environment. *arXiv preprint arXiv:1907.11612*, 2019. doi: 10.48550/arXiv.1907.11612.
- [47] Dong-Jun Han, Hasnain Irshad Bhatti, Jungmoon Lee, and Jaekyun Moon. Accelerating Federated Learning with Split Learning on Locally Generated Losses. In *International Conference on Machine Learning Workshop on Federated Learning for User Privacy and Data Confidentiality*, 2021.
- [48] Dong-Jun Han, Do-Yeon Kim, Minseok Choi, Christopher G Brinton, and Jaekyun Moon. SplitGP: Achieving Both Generalization and Personalization in Federated Learning. In *IEEE Conference on Computer Communications*, pages 1–10, 2023. doi: 10.1109/infocom53939.2023.10229027.
- [49] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. A Survey on Vision Transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):87–110, 2022. doi: 10.1109/TPAMI.2022.3152247.
- [50] Pengchao Han, Shiqiang Wang, and Kin K Leung. Adaptive Gradient Sparsification for Efficient Federated Learning: An Online Learning Approach. In *IEEE International Conference on Distributed Computing Systems*, pages 300–310, 2020. doi: 10.1109/icdcs47774.2020.00026.
- [51] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *International Conference on Learning Representations*, 2016.
- [52] Natascha Harth, Christos Anagnostopoulos, Hans-Joerg Voegel, and Kostas Kolomvatsos. Local & Federated Learning at the Network Edge for Efficient Predictive Analytics. *Future Generation Computer Systems*, 134:107–122, 2022. doi: 10.1016/j.future.2022.03.030.
- [53] Chaoyang He, Murali Annavaram, and Salman Avestimehr. Group Knowledge Transfer: Federated Learning of Large CNNs at the Edge. *Advances in Neural Information Processing Systems*, 33:14068–14080, 2020.

- [54] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. doi: 10.1109/cvpr.2016.90.
- [55] Peng He, Chunhui Lan, Yaping Cui, Ruyan Wang, and Dapeng Wu. Accelerated Federated Learning with Dynamic Model Partitioning for H-IoT. In *IEEE Wireless Communications and Networking Conference*, pages 1–6, 2023. doi: 10.1109/WCNC55385.2023.10118865.
- [56] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural Networks for Machine Learning Lecture 6a Overview of Mini-Batch Gradient Descent. *University of Toronto*, 14(8):2, 2012.
- [57] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. Distilling the Knowledge in A Neural Network. In *Advances in Neural Information Processing Systems Workshop on Deep Learning*, 2014.
- [58] Samuel Horvath, Stefanos Laskaridis, Mario Almeida, Ilias Leontiadis, Stylianos Venieris, and Nicholas Lane. FjORD: Fair and Accurate Federated Learning under Heterogeneous Targets with Ordered Dropout. *Advances in Neural Information Processing Systems*, 34:12876–12889, 2021.
- [59] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. Vehicular Fog Computing: A Viewpoint of Vehicles as the Infrastructures. *IEEE Transactions on Vehicular Technology*, 65(6):3860–3873, 2016. doi: 10.1109/TVT.2016.2532863.
- [60] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017. doi: 10.48550/arXiv.1704.04861.
- [61] Hanpeng Hu, Dan Wang, and Chuan Wu. Distributed Machine Learning through Heterogeneous Edge Systems. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 7179–7186, 2020. doi: 10.1609/aaai.v34i05.6207.
- [62] Haochen Hua, Yutong Li, Tonghe Wang, Nanqing Dong, Wei Li, and Junwei Cao. Edge Computing with Artificial Intelligence: A Machine Learning Perspective. *ACM Computing Surveys*, 55(9):1–35, 2023. doi: 10.1145/3555802.
- [63] Minyoung Huh, Pulkit Agrawal, and Alexei A Efros. What Makes ImageNet Good for Transfer Learning? In *Advances in Neural Information Processing Systems Workshop on Large Scale Computer Vision Systems*, 2016.
- [64] Zhouyuan Huo, Bin Gu, Heng Huang, et al. Decoupled Parallel Backpropagation with Convergence Guarantee. In *International Conference on Machine Learning*, pages 2098–2106, 2018.
- [65] IDC. Future of Industry Ecosystems: Shared Data and Insights. Available online: <https://blogs.idc.com/2021/01/06/>, Accessed 10/28/2024.

- [66] Ahmed Imteaj, Khandaker Mamun Ahmed, Urmish Thakker, Shiqiang Wang, Jian Li, and M. Hadi Amini. *Federated Learning for Resource-Constrained IoT Devices: Panoramas and State of the Art*, pages 7–27. Springer International Publishing, 2023. doi: 10.1007/978-3-031-11748-0_2.
- [67] Sohei Itahara, Takayuki Nishio, Yusuke Koda, Masahiro Morikura, and Koji Yamamoto. Distillation-Based Semi-Supervised Federated Learning for Communication-Efficient Collaborative Training with Non-IID Private Data. *IEEE Transactions on Mobile Computing*, 22(1):191–205, 2021. doi: 10.1109/TMC.2021.3070013.
- [68] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine Learning and Deep Learning. *Electronic Markets*, 31(3):685–695, 2021. doi: 10.1007/s12525-021-00475-2.
- [69] Congfeng Jiang, Xiaolan Cheng, Honghao Gao, Xin Zhou, and Jian Wan. Toward Computation Offloading in Edge Computing: A Survey. *IEEE Access*, 7:131543–131558, 2019. doi: 10.1109/access.2019.2938660.
- [70] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. MNN: A Universal and Efficient Inference Engine. In *Proceedings of the Conference on Machine Learning and Systems*, 2020.
- [71] Yuang Jiang, Shiqiang Wang, Victor Valls, Bong Jun Ko, Wei-Han Lee, Kin K Leung, and Leandros Tassiulas. Model Pruning Enables Efficient Federated Learning on Edge Devices. *IEEE Transactions on Neural Networks and Learning Systems*, 34(12):10374–10386, 2023. doi: 10.1109/TNNLS.2022.3166101.
- [72] Zhida Jiang, Yang Xu, Hongli Xu, Zhiyuan Wang, Chunming Qiao, and Yangming Zhao. FedMP: Federated Learning through Adaptive Model Pruning in Heterogeneous Edge Computing. In *International Conference on Data Engineering*, pages 767–779, 2022. doi: 10.1109/ICDE53745.2022.00062.
- [73] Michael I Jordan and Tom M Mitchell. Machine Learning: Trends, Perspectives, and Prospects. *Science*, 349(6245):255–260, 2015. doi: 10.1126/science.aaa8415.
- [74] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly Accurate Protein Structure Prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021. doi: 10.3410/f.740477161.793588050.
- [75] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and Open Problems in Federated Learning. *Foundations and Trends in Machine Learning*, 14(1–2):1–210, 2021.
- [76] Sham M Kakade, Shai Shalev-Shwartz, and Ambuj Tewari. Regularization Techniques for Learning with Matrices. *The Journal of Machine Learning Research*, 13(1):1865–1890, 2012.

- [77] Trupti M Kodinariya and Prashant R Makwana. Review on Determining Number of Cluster in K-Means Clustering. *International Journal of Advance Research in Computer Science and Management Studies*, 1(6):90–95, 2013.
- [78] Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated Learning: Strategies for Improving Communication Efficiency. In *Advances in Neural Information Processing Systems Workshop on Private Multi-Party Machine Learning*, 2017.
- [79] Alex Krizhevsky, Geoffrey Hinton, et al. Learning Multiple Layers of Features from Tiny Images. *University of Toronto*, 2009.
- [80] Ya Le and Xuan Yang. Tiny Imagenet Visual Recognition Challenge. Available online: http://cs231n.stanford.edu/reports/2015/pdfs/yle_project.pdf, Accessed 06/19/2024.
- [81] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.
- [82] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding up Distributed Machine Learning Using Codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, 2017. doi: 10.1109/TIT.2017.2736066.
- [83] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. Hermes: An Efficient Federated Learning Framework for Heterogeneous Mobile Clients. In *International Conference on Mobile Computing and Networking*, pages 420–437, 2021. doi: 10.1145/3447993.3483278.
- [84] En Li, Zhi Zhou, and Xu Chen. Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy. In *Proceedings of the Workshop on Mobile Edge Communications*, pages 31–36, 2018. doi: 10.1145/3229556.3229562.
- [85] Li Li, Haoyi Xiong, Zhishan Guo, Jun Wang, and Cheng-Zhong Xu. SmartPC: Hierarchical Pace Control in Real-Time Federated Learning System. In *IEEE Real-Time Systems Symposium*, pages 406–418, 2019. doi: 10.1109/rtss46320.2019.00043.
- [86] Liangzhi Li, Kaoru Ota, and Mianxiong Dong. Deep Learning for Smart Industry: Efficient Manufacture Inspection System with Fog Computing. *IEEE Transactions on Industrial Informatics*, 14(10):4665–4673, 2018. doi: 10.1109/tii.2018.2842821.
- [87] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. *Advances in Neural Information Processing Systems*, 27, 2014.
- [88] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch Distributed: Experiences on Accelerating Data Parallel Training. In *Proceedings of the VLDB Endowment*, volume 13, pages 2150–8097, 2020. doi: 10.14778/3415478.3415530.

- [89] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated Optimization in Heterogeneous Networks. *Proceedings of Machine Learning and Systems*, 2:429–450, 2020.
- [90] Yang Li, Zhenhua Han, Quanlu Zhang, Zhenhua Li, and Haisheng Tan. Automating Cloud Deployment for Deep Learning Inference of Real-Time Online Services. In *IEEE Conference on Computer Communications*, pages 1668–1677, 2020. doi: 10.1109/INFOCOM41043.2020.9155267.
- [91] Yijing Li, Xiaofeng Tao, Xuefei Zhang, Junjie Liu, and Jin Xu. Privacy-Preserved Federated Learning for Autonomous Driving. *IEEE Transactions on Intelligent Transportation Systems*, 23(7):8423–8434, 2021. doi: 10.1109/TITS.2021.3081560.
- [92] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019, 2021. doi: 10.1109/TNNLS.2021.3084827.
- [93] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. Computation Offloading Toward Edge Computing. *Proceedings of the IEEE*, 107(8):1584–1607, 2019. doi: 10.1109/JPROC.2019.2922285.
- [94] Mingbao Lin, Rongrong Ji, Yuxin Zhang, Baochang Zhang, Yongjian Wu, and Yonghong Tian. Channel Pruning via Automatic Structure Search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2020. doi: 10.24963/ijcai.2020/94.
- [95] Yimin Lin, Kai Wang, Wanxin Yi, and Shiguo Lian. Deep Learning Based Wearable Assistive System for Visually Impaired People. In *IEEE/CVF International Conference on Computer Vision Workshop*, pages 2549–2557, 2019. doi: 10.1109/ICCVW.2019.00312.
- [96] Lumin Liu, Jun Zhang, SH Song, and Khaled B Letaief. Client-Edge-Cloud Hierarchical Federated Learning. In *IEEE International Conference on Communications*, pages 1–6, 2020. doi: 10.1109/icc40277.2020.9148862.
- [97] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019. doi: 10.1109/JPROC.2019.2915983.
- [98] Yuhan Liu, Saurabh Agarwal, and Shivaram Venkataraman. AutoFreeze: Automatically Freezing Model Blocks to Accelerate Fine-Tuning. *arXiv preprint arXiv:2102.01386*, 2021. doi: 10.48550/arXiv.2102.01386.
- [99] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the Value of Network Pruning. In *International Conference on Learning Representations*, 2019.
- [100] Luke Lockhart, Paul Harvey, Pierre Imai, Peter Willis, and Blesson Varghese. Scission: Performance-Driven and Context-Aware Cloud-Edge Distribution of Deep Neural Networks. In *IEEE/ACM International Conference on Utility and Cloud Computing*, pages 257–268, 2020. doi: 10.1109/ucc48980.2020.00044.

- [101] Qianyu Long, Christos Anagnostopoulos, Shameem Puthiya Parambath, and Daning Bi. FedDIP: Federated Learning with Extreme Dynamic Pruning and Incremental Regularization. In *IEEE International Conference on Data Mining*, pages 1187–1192, 2023. doi: 10.1109/ICDM58522.2023.00146.
- [102] Pavel Mach and Zdenek Becvar. Mobile Edge Computing: A Survey on Architecture and Computation Offloading. *IEEE Communications Surveys and Tutorials*, 19(3): 1628–1656, 2017. doi: 10.1109/COMST.2017.2682318.
- [103] Aravindh Mahendran and Andrea Vedaldi. Understanding Deep Image Representations by Inverting Them. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5188–5196, 2015. doi: 10.1109/cvpr.2015.7299155.
- [104] AKM Jahangir Majumder and Joshua Aaron Izaguirre. A Smart IoT Security System for Smart-Home Using Motion Detection and Facial Recognition. In *IEEE Annual Computers, Software, and Applications Conference*, pages 1065–1071, 2020. doi: 10.1109/COMPSAC48688.2020.0-132.
- [105] Erich Malan, Valentino Peluso, Andrea Calimera, Enrico Macii, and Paolo Montuschi. Automatic Layer Freezing for Communication Efficiency in Cross-Device Federated Learning. *IEEE Internet of Things Journal*, 11(4):6072–6083, 2023. doi: 10.1109/IJOT.2023.3309691.
- [106] Utpal Mangla. Application of Federated Learning in Telecommunications and Edge Computing. In *Federated Learning: A Comprehensive Overview of Methods and Applications*, pages 523–534. 2022. doi: 10.1007/978-3-030-96896-0_25.
- [107] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani, Songtao He, et al. Park: An Open Platform for Learning Augmented Computer Systems. In *Advances in Neural Information Processing Systems*, pages 2490–2502, 2019.
- [108] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys and Tutorials*, 19(4):2322–2358, 2017. doi: 10.1109/COMST.2017.2745201.
- [109] Dan C Marinescu. *Cloud Computing: Theory and Practice*. Morgan Kaufmann, 2022. doi: 10.1016/C2020-0-02233-4.
- [110] Mohsen Marjani, Fariza Nasaruddin, Abdullah Gani, Ahmad Karim, Ibrahim Abaker Targio Hashem, Aisha Siddiqa, and Ibrar Yaqoob. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, 5: 5247–5261, 2017. doi: 10.1109/ACCESS.2017.2689040.
- [111] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *International Conference on Artificial Intelligence and Statistics*, pages 1273–1282, 2017.

- [112] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. MLLib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34): 1–7, 2016.
- [113] MIT. Split Learning: Distributed Deep Learning and Inference without Sharing Raw Data. Available online: <https://www.media.mit.edu/projects/distributed-learning-and-collaborative-learning-1/overview/>, Accessed 06/19/2024.
- [114] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013. doi: 10.48550/arXiv.1312.5602.
- [115] Chnar Mustafa Mohammed and Subhi RM Zeebaree. Sufficient Comparison among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review. *International Journal of Science and Business*, 5(2):17–30, 2021. doi: 0.5281/zenodo.4481415.
- [116] Ari Morcos, Maithra Raghu, and Samy Bengio. Insights on Representational Similarity in Neural Networks with Canonical Correlation. *Advances in Neural Information Processing Systems*, 31, 2018.
- [117] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 561–577, 2018.
- [118] Preetum Nakkiran, Behnam Neyshabur, and Hanie Sedghi. The Deep Bootstrap Framework: Good Online Learners are Good Offline Generalizers. In *International Conference on Learning Representations*, 2021.
- [119] Dinh C Nguyen, Ming Ding, Pubudu N Pathirana, Aruna Seneviratne, Jun Li, and H Vincent Poor. Federated Learning for Internet of Things: A Comprehensive Survey. *IEEE Communications Surveys and Tutorials*, 23(3):1622–1658, 2021. doi: 10.1109/COMST.2021.3075439.
- [120] John Nguyen, Kshitiz Malik, Maziar Sanjabi, and Michael Rabbat. Where to Begin? Exploring the Impact of Pre-Training and Initialization in Federated Learning. In *International Conference on Learning Representations*, 2023.
- [121] Solmaz Niknam, Harpreet S Dhillon, and Jeffrey H Reed. Federated Learning for Wireless Communications: Motivation, Opportunities, and Challenges. *IEEE Communications Magazine*, 58(6):46–51, 2020. doi: 10.1109/MCOM.001.1900461.
- [122] Arild Nøkland and Lars Hiller Eidnes. Training Neural Networks with Local Error Signals. In *International Conference on Machine Learning*, pages 4839–4850, 2019.
- [123] Nvidia. NVIDIA A100 Tensor Core GPU, Unprecedented Acceleration at Every Scale. Available online: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf>, Accessed 06/19/2024.

- [124] OpenAI. Introducing ChatGPT. Available online: <https://openai.com/blog/chatgpt>, Accessed 06/19/2024.
- [125] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep Exploration via Bootstrapped DQN. In *Advances in Neural Information Processing Systems*, volume 29, pages 4033–4041, 2016.
- [126] Sharnil Pandya, Gautam Srivastava, Rutvij Jhaveri, M Rajasekhara Babu, Sweta Bhat-tacharya, Praveen Kumar Reddy Maddikunta, Spyridon Mastorakis, Md Jalil Piran, and Thippa Reddy Gadekallu. Federated Learning for Smart Cities: A Comprehensive Survey. *Sustainable Energy Technologies and Assessments*, 55:102987, 2023. doi: 10.1016/j.seta.2022.102987.
- [127] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, 2019.
- [128] S Patro. Normalization: A Preprocessing Stage. *arXiv preprint arXiv:1503.06462*, 2015. doi: 10.48550/arXiv.1503.06462.
- [129] Chandrashekhar S Pawar, Amit Ganatra, Amit Nayak, Dipak Ramoliya, and Rajesh Patel. Use of Machine Learning Services in Cloud. In *Computer Networks, Big Data and IoT*, pages 43–52, 2021. doi: 10.1007/978-981-16-0965-7_5.
- [130] Damiano Perri, Marco Simonetti, Sergio Tasso, Federico Ragni, and Osvaldo Gervasi. Implementing a Scalable and Elastic Computing Environment Based on Cloud Containers. In *International Conference on Computational Science and Its Applications*, pages 676–689, 2021.
- [131] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. CoCoFL: Communication-and Computation-Aware Federated Learning via Partial NN Freezing and Quantization. *arXiv preprint arXiv:2203.05468*, 2022. doi: 10.48550/arXiv.2203.05468.
- [132] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. Federated Learning for Computationally Constrained Heterogeneous Devices: A Survey. *ACM Computing Surveys*, 55(14s):1–27, 2023. doi: 10.1145/3596907.
- [133] Maarten G Poirot, Praneeth Vepakomma, Ken Chang, Jayashree Kalpathy-Cramer, Rajiv Gupta, and Ramesh Raskar. Split Learning for Collaborative Deep Learning in Healthcare. *arXiv preprint arXiv:1912.12115*, 2019. doi: 10.48550/arXiv.1912.12115.
- [134] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model Compression via Distillation and Quantization. In *International Conference on Learning Representations*, 2018.
- [135] Bin Qian, Jie Su, Zhenyu Wen, Devki Nandan Jha, Yinhao Li, Yu Guan, Deepak Puthal, Philip James, Renyu Yang, Albert Y Zomaya, et al. Orchestrating the Development Lifecycle of Machine Learning-Based IoT Applications: A Taxonomy and Survey. *ACM Computing Surveys*, 53(4):1–47, 2020. doi: 10.1145/3398020.

- [136] Bin Qian, Yubo Xuan, Di Wu, Zhenyu Wen, Renyu Yang, Shibo He, Jiming Chen, and Rajiv Ranjan. Edge-Cloud Collaborative Streaming Video Analytics with Multi-agent Deep Reinforcement Learning. *IEEE Network*, 2024. doi: 10.1109/mnet.2024.3398724.
- [137] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability. *Advances in Neural Information Processing Systems*, 30, 2017.
- [138] Partha Pratim Ray. A Review on TinyML: State-of-the-Art and Prospects. *Journal of King Saud University-Computer and Information Sciences*, 34(4):1595–1623, 2022. doi: 10.1016/j.jksuci.2021.11.019.
- [139] Amirhossein Reisizadeh, Aryan Mokhtari, Hamed Hassani, Ali Jadbabaie, and Ramtin Pedarsani. FedpPAQ: A Communication-Efficient Federated Learning Method with Periodic Averaging and Quantization. In *International Conference on Artificial Intelligence and Statistics*, pages 2021–2031, 2020.
- [140] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista Gomez, and Rosa M Badia. A Study of Checkpointing in Large Scale Training of Deep Neural Networks. In *International Conference on High Performance Computing and Simulation*, 2020.
- [141] Abhijit Guha Roy, Shayan Siddiqui, Sebastian Pölsterl, Nassir Navab, and Christian Wachinger. Braintorrent: A Peer-to-Peer Environment for Decentralized Federated Learning. *arXiv preprint arXiv:1905.06731*, 2019. doi: 10.48550/arXiv.1905.06731.
- [142] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep Reinforcement Learning Framework for Autonomous Driving. *Electronic Imaging*, 29(19):70–70, 2017. doi: 10.2352/issn.2470-1173.2017.19.avm-023.
- [143] Amanpreet Kaur Sandhu. Big Data with Cloud Computing: Discussions and Challenges. *Big Data Mining and Analytics*, 5(1):32–40, 2021. doi: 10.26599/BDMA.2021.9020016.
- [144] Mehmet Tahir Sandikkaya, Yusuf Yaslan, and Cemile Diler Özdemir. DeMETER in Clouds: Detection of Malicious External Thread Execution in Runtime with Machine Learning in PaaS Clouds. *Cluster Computing*, 23(4):2565–2578, 2020. doi: 10.1007/s10586-019-03027-8.
- [145] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetv2: Inverted Residuals and Linear Bottlenecks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. doi: 10.1109/cvpr.2018.00474.
- [146] Iqbal H Sarker. Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN Computer Science*, 2(3):160, 2021. doi: 10.1007/s42979-021-00592-x.
- [147] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017. doi: 10.48550/arXiv.1707.06347.

- [148] Holger Schwenk, Guillaume Wenzek, Sergey Edunov, Edouard Grave, and Armand Joulin. CCMatrix: Mining Billions of High-Quality Parallel Sentences on the Web. In *Proceedings of Annual Meeting of the Association for Computational Linguistics*, pages 6490–6500, 2021. doi: 10.18653/v1/2021.acl-long.507.
- [149] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018. doi: 10.48550/arXiv.1802.05799.
- [150] Osama Shahid, Seyedamin Pouriyeh, Reza M Parizi, Quan Z Sheng, Gautam Srivastava, and Liang Zhao. Communication Efficiency in Federated Learning: Achievements and Challenges. *arXiv preprint arXiv:2107.10996*, 2021. doi: 10.48550/arXiv.2107.10996.
- [151] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. *The Journal of Machine Learning Research*, 20(112):1–49, 2019.
- [152] Jiawei Shao and Jun Zhang. BottleNet++: An End-to-End Approach for Feature Compression in Device-Edge Co-Inference Systems. In *IEEE International Conference on Communications Workshops*, pages 1–6, 2020. doi: 10.1109/iccworkshops49005.2020.9145068.
- [153] Akanksha Rai Sharma and Pranav Kaushik. Literature Survey of Statistical, Deep and Reinforcement Learning in Natural Language Processing. In *IEEE International Conference on Computation, Communication and Automation*, pages 350–354, 2017. doi: 10.1109/cca.2017.8229841.
- [154] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. doi: 10.1109/JIOT.2016.2579198.
- [155] Pramila P Shinde and Seema Shah. A Review of Machine Learning and Deep Learning Applications. In *International Conference on Computing Communication Control and Automation*, pages 1–6, 2018. doi: 10.1109/ICCUBEA.2018.8697857.
- [156] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019. doi: 10.48550/arXiv.1909.08053.
- [157] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the Game of Go without Human Knowledge. *Nature*, 550(7676):354–359, 2017. doi: 10.15368/theses.2018.47.
- [158] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015.

- [159] Cagatay Sonmez, Can Tunca, Atay Ozgovde, and Cem Ersoy. Machine Learning-Based Workload Orchestrator for Vehicular Edge Computing. *IEEE Transactions on Intelligent Transportation Systems*, 22(4):2239–2251, 2020. doi: 10.1109/TITS.2020.3024233.
- [160] Sebastian U Stich. Local SGD Converges Fast and Communicates Little. In *International Conference on Learning Representations*, 2019.
- [161] Yan Sun, Li Shen, and Dacheng Tao. Which Mode Is Better for Federated Learning? Centralized or Decentralized. *arXiv preprint arXiv:2310.03461*, 2023. doi: 10.48550/arXiv.2310.03461.
- [162] Ali Sunyaev and Ali Sunyaev. *Cloud Computing*, pages 195–236. Springer, 2020. doi: 10.1007/978-3-031-61014-1_6.
- [163] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, volume 99, pages 1057–1063, 1999.
- [164] Mingxing Tan and Quoc Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.
- [165] Chandra Thapa, Pathum Chamikara Mahawaga Arachchige, Seyit Camtepe, and Lichao Sun. SplitFed: When Federated Learning Meets Split Learning. In *AAAI Conference on Artificial Intelligence*, volume 36, pages 8485–8493, 2022. doi: 10.1609/aaai.v36i8.20825.
- [166] Lisa Torrey and Jude Shavlik. Transfer Learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. 2010.
- [167] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Challenges and Opportunities in Edge Computing. In *IEEE International Conference on Smart Cloud*, pages 20–26, 2016. doi: 10.1109/SmartCloud.2016.18.
- [168] Blesson Varghese, Eyal De Lara, Aaron Yi Ding, Cheol-Ho Hong, Flavio Bonomi, Schahram Dustdar, Paul Harvey, Peter Hewkin, Weisong Shi, Mark Thiele, et al. Revisiting the Arguments for Edge Computing Research. *IEEE Internet Computing*, 25(5):36–42, 2021. doi: 10.1109/MIC.2021.3093924.
- [169] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split Learning for Health: Distributed Deep Learning Without Sharing Raw Patient Data. In *International Conference on Learning Representations Workshop on AI for Social Good*, 2019. doi: 10.48550/arXiv.1812.00564.
- [170] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. A Survey on Distributed Machine Learning. *ACM Computing Surveys*, 53(2):1–33, 2020. doi: 10.1145/3377454.

- [171] Cong Wang, Yuanyuan Yang, and Pengzhan Zhou. Towards Efficient Scheduling of Federated Mobile Devices under Computational and Statistical Heterogeneity. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):394–410, 2021. doi: 10.1109/TPDS.2020.3023905.
- [172] Jianyu Wang, Hang Qi, Ankit Singh Rawat, Sashank Reddi, Sagar Waghmare, Felix X Yu, and Gauri Joshi. FedLite: A Scalable Approach for Federated Learning on Resource-constrained Clients. *arXiv preprint arXiv:2201.11865*, 2022. doi: 10.48550/arXiv.2201.11865.
- [173] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. Adaptive Federated Learning in Resource Constrained Edge Computing Systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019. doi: 10.1109/JSAC.2019.2904348.
- [174] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Egeria: Efficient DNN Training with Knowledge-Guided Layer Freezing. In *European Conference on Computer Systems*, pages 851–866, 2023. doi: 10.1145/3552326.3587451.
- [175] Herbert Woisetschläger, Alexander Erben, Bill Marino, Shiqiang Wang, Nicholas D Lane, Ruben Mayer, and Hans-Arno Jacobsen. Federated Learning Priorities Under the European Union Artificial Intelligence Act. In *International Conference on Machine Learning Workshop on Generative AI and Law*, 2024.
- [176] Di Wu, Qi Tang, Yongle Zhao, Ming Zhang, Ying Fu, and Debing Zhang. EasyQuant: Post-Training Quantization via Scale Optimization. *arXiv preprint arXiv:2006.16669*, 2020. doi: 10.48550/arXiv.2006.16669.
- [177] Di Wu, Rehmat Ullah, Paul Harvey, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. FedAdapt: Adaptive Offloading for IoT Devices in Federated Learning. *IEEE Internet of Things Journal*, 9(21):20889–20901, 2022. doi: 10.1109/JIOT.2022.3176469.
- [178] Di Wu, Rehmat Ullah, Philip Rodgers, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. EcoFed: Efficient Communication for DNN Partitioning-Based Federated Learning. *IEEE Transactions on Parallel and Distributed Systems*, 35(3):377–390, 2024. doi: 10.1109/TPDS.2024.3349617.
- [179] Qiong Wu, Xu Chen, Tao Ouyang, Zhi Zhou, Xiaoxi Zhang, Shusen Yang, and Junshan Zhang. HiFlash: Communication-Efficient Hierarchical Federated Learning with Adaptive Staleness Control and Heterogeneity-aware Client-Edge Association. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1560–1579, 2023. doi: 10.1109/tpds.2023.3238049.
- [180] Xueyu Wu, Xin Yao, and Cho-Li Wang. FedSCR: Structure-based Communication Reduction for Federated Learning. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1565–1577, 2020. doi: 10.1109/tpds.2020.3046250.
- [181] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747*, 2017. doi: 10.48550/arXiv.1708.07747.

- [182] Dianlei Xu, Tong Li, Yong Li, Xiang Su, Sasu Tarkoma, Tao Jiang, Jon Crowcroft, and Pan Hui. Edge Intelligence: Architectures, Challenges, and Applications. *arXiv preprint arXiv:2003.12172*, 2020. doi: 10.48550/arXiv.2003.12172.
- [183] Runhua Xu, Nathalie Baracaldo, and James Joshi. Privacy-Preserving Machine Learning: Methods, Challenges and Directions. *arXiv preprint arXiv:2108.04417*, 2021. doi: 10.48550/arXiv.2108.04417.
- [184] Zirui Xu, Fuxun Yu, Jinjun Xiong, and Xiang Chen. Helios: Heterogeneity-Aware Federated Learning with Dynamically Balanced Collaboration. In *ACM/IEEE Design Automation Conference*, pages 997–1002, 2021. doi: 10.1109/DAC18074.2021.9586241.
- [185] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated Machine Learning: Concept and Applications. *ACM Transactions on Intelligent Systems and Technology*, 10(2):1–19, 2019. doi: 10.1145/3298981.
- [186] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied Federated Learning: Improving Google Keyboard Query Suggestions. *arXiv preprint arXiv:1812.02903*, 2018. doi: 10.48550/arXiv.1812.02903.
- [187] Xue Ying. An Overview of Overfitting and Its Solutions. In *Journal of Physics: Conference Series*, volume 1168, page 022022, 2019. doi: 10.1088/1742-6596/1168/2/022022.
- [188] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable are Features in Deep Neural Networks? *Advances in Neural Information Processing Systems*, 27, 2014.
- [189] Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel Restarted SGD with Faster Convergence and Less Communication: Demystifying Why Model Averaging Works for Deep Learning. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 5693–5700, 2019. doi: 10.1609/aaai.v33i01.33015693.
- [190] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, and Yuan Gao. A Survey on Federated Learning. *Knowledge-Based Systems*, 216:106775, 2021. doi: 10.1016/j.knosys.2021.106775.
- [191] Tuo Zhang, Chaoyang He, Tianhao Ma, Lei Gao, Mark Ma, and Salman Avestimehr. Federated Learning for Internet of Things. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, page 413–419, 2021. doi: 10.1145/3485730.3493444.
- [192] Tuo Zhang, Lei Gao, Chaoyang He, Mi Zhang, Bhaskar Krishnamachari, and A Salman Avestimehr. Federated Learning for the Internet of Things: Applications, Challenges, and Opportunities. *IEEE Internet of Things Magazine*, 5(1):24–29, 2022. doi: 10.1109/iotm.004.2100182.

- [193] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018. doi: 10.1109/cvpr.2018.00716.
- [194] Zhilu Zhang and Mert Sabuncu. Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels. *Advances in Neural Information Processing Systems*, 31, 2018.
- [195] Zihan Zhang, Philip Rodgers, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. PiPar: Pipeline Parallelism for Collaborative Machine Learning. *Journal of Parallel and Distributed Computing*, 193:104947, 2024. doi: 10.1016/j.jpdc.2024.104947.
- [196] Tao Zheng, Jian Wan, Jilin Zhang, Congfeng Jiang, and Gangyong Jia. A Survey of Computation Offloading in Edge Computing. In *International Conference on Computer, Information and Telecommunication Systems*, pages 1–6, 2020. doi: 10.1109/CITS49457.2020.9232457.
- [197] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive Quantization for Deep Neural Network. In *AAAI Conference on Artificial Intelligence*, volume 32, 2018. doi: 10.1109/icme.2018.8486500.
- [198] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. In *Proceedings of the IEEE*, pages 1738–1762, 2019. doi: 10.1109/jproc.2019.2918951.
- [199] Michael Zhu and Suyog Gupta. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. In *International Conference on Learning Representations Workshop*, 2017.
- [200] Xiaomeng Zhu, Talha Bilal, Pär Mårtensson, Lars Hanson, Mårten Björkman, and Atsuto Maki. Towards Sim-to-Real Industrial Parts Classification With Synthetic Dataset. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4453–4462, 2023. doi: 10.1109/cvprw59228.2023.00468.

Appendix A

Proof of Convergence for EcoFed (Chapter 4.4)

Convergence of the Server-side Model in EcoFed

This Appendix presents the proof to demonstrate the convergence of the server-side model (\mathbf{w}_S) in EcoFed presented in Chapter 4.4.

$$\begin{aligned} \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \eta_t \mathbb{E}[\|\nabla F_S(\mathbf{w}_S^t)\|^2] &\leq \frac{4(F_S(\mathbf{w}_S^0) - F_S(\mathbf{w}_S^*))}{3\Gamma_T} \\ &+ \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \left(\eta_t (\sqrt{G} + 1) (H_1 + H_2) + \frac{L}{2} \eta_t^2 G \right) \end{aligned} \quad (1)$$

Proof: Based on Assumption 1

$$F_S(\mathbf{w}_S^{t+1}) \leq F_S(\mathbf{w}_S^t) + \nabla F_S(\mathbf{w}_S^t)^T (\mathbf{w}_S^{t+1} - \mathbf{w}_S^t) + \frac{L}{2} \|\mathbf{w}_S^{t+1} - \mathbf{w}_S^t\|^2 \quad (2)$$

Also, note

$$\mathbf{w}_S^{t+1} = \mathbf{w}_S^t - \eta_t \frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \quad (3)$$

When substituting Equation 3 in Equation 2

$$F_S(\mathbf{w}_S^{t+1}) \leq F_S(\mathbf{w}_S^t) - \eta_t \nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right) + \frac{L}{2} \eta_t^2 \left\| \frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right\|^2 \quad (4)$$

By taking the expectation from both sides of Equation 4

$$\begin{aligned} \mathbb{E}[F_S(\mathbf{w}_S^{t+1})] \leq & \mathbb{E}[F_S(\mathbf{w}_S^t)] - \underbrace{\eta_t \mathbb{E} \left[\nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right) \right]}_{B_1} \\ & + \underbrace{\frac{L}{2} \eta_t^2 \mathbb{E} \left[\left\| \frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right\|^2 \right]}_{B_2} \end{aligned} \quad (5)$$

Next, the lower bound of B_1 and upper bound of B_2 are identified. X is defined as

$$X = \frac{1}{K} \sum_{k=1}^K (\nabla F_{S,k}(\mathbf{w}_S^t; \mathbf{a}_k^t) - \nabla F_{S,k}(\mathbf{w}_S^t; \hat{\mathbf{a}}_k^t)) \quad (6)$$

which is the difference in average gradient using \mathbf{a}_k^t and $\hat{\mathbf{a}}_k^t$. For simplicity, the following abbreviations are used.

$$X = \frac{1}{K} \sum_{k=1}^K (\nabla F_{S,k}(\mathbf{w}_S^t) - \hat{\nabla} F_{S,k}(\mathbf{w}_S^t)) \quad (7)$$

Substituting X for B_1

$$B_1 = \mathbb{E} \left[\nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right) \right] \quad (8)$$

$$= \mathbb{E} \left[\nabla F_S(\mathbf{w}_S^t)^T \left(X + \frac{1}{K} \sum_{k=1}^K \hat{\nabla} F_{S,k}(\mathbf{w}_S^t) \right) \right] \quad (9)$$

$$\geq \mathbb{E} \left[\nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K \hat{\nabla} F_{S,k}(\mathbf{w}_S^t) \right) \right] - \|\mathbb{E}[\nabla F_S(\mathbf{w}_S^t)^T X]\| \quad (10)$$

$$\begin{aligned} &= \underbrace{\mathbb{E} \left[\nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right) \right]}_{C_1} \\ &+ \underbrace{\mathbb{E} \left[\nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K (\hat{\nabla} F_{S,k}(\mathbf{w}_S^t) - \nabla F_{S,k}(\mathbf{w}_S^t)) \right) \right]}_{C_2} - \underbrace{\|\mathbb{E}[\nabla F_S(\mathbf{w}_S^t)^T X]\|}_{C_3} \end{aligned} \quad (11)$$

C_1 is defined as

$$C_1 = \mathbb{E}[\|\nabla F_{S,k}(\mathbf{w}_S^t)\|^2] \quad (12)$$

When considering C_2

$$\nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K (\hat{\nabla} F_{S,k}(\mathbf{w}_S^t) - \nabla F_{S,k}(\mathbf{w}_S^t)) \right) \quad (13)$$

$$\geq - \left\| \nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K (\hat{\nabla} F_{S,k}(\mathbf{w}_S^t) - \nabla F_{S,k}(\mathbf{w}_S^t)) \right) \right\| \quad (14)$$

$$\geq - \left\| \nabla F_S(\mathbf{w}_S^t)^T \right\| \left\| \left(\frac{1}{K} \sum_{k=1}^K (\hat{\nabla} F_{S,k}(\mathbf{w}_S^t) - \nabla F_{S,k}(\mathbf{w}_S^t)) \right) \right\| \quad (15)$$

$$\geq -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \left\| \hat{\nabla} F_{S,k}(\mathbf{w}_S^t) - \nabla F_{S,k}(\mathbf{w}_S^t) \right\| \quad (16)$$

$$= -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \left\| \nabla F_{S,k}(\mathbf{w}_S^t; \hat{\mathbf{a}}_k^t) - \nabla F_{S,k}(\mathbf{w}_S^t; \mathbf{a}_k^t) \right\| \quad (17)$$

$$= -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \left\| \frac{1}{|D_k|} \sum_{x \in D_k} \nabla \ell(\hat{\mathbf{a}}_k^t; \mathbf{w}_S) - \frac{1}{|D_k|} \sum_{x \in D_k} \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) \right\| \quad (18)$$

$$= -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \left\| \int \nabla \ell(\hat{\mathbf{a}}_k^t; \mathbf{w}_S) q_k^t(a) da - \int \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) p_k^t(a) da \right\| \quad (19)$$

$$= -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \left\| \int \nabla \ell(\hat{\mathbf{a}}_k^t; \mathbf{w}_S) q_k^t(a) da - \int \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) q_k^t(a) da \right. \\ \left. + \int \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) q_k^t(a) da - \int \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) p_k^t(a) da \right\| \quad (20)$$

$$= -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \left\| \int (\nabla \ell(\hat{\mathbf{a}}_k^t; \mathbf{w}_S) - \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S)) q_k^t(a) da + \int \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) (q_k^t(a) - p_k^t(a)) da \right\| \quad (21)$$

$$\geq -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \left\| \int (\nabla \ell(\hat{\mathbf{a}}_k^t; \mathbf{w}_S) - \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S)) q_k^t(a) da \right\| + \left\| \int \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) (q_k^t(a) - p_k^t(a)) da \right\| \quad (22)$$

$$\geq -\sqrt{G} \frac{1}{K} \sum_{k=1}^K \int \left\| \nabla \ell(\hat{\mathbf{a}}_k^t; \mathbf{w}_S) - \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) \right\| \|q_k^t(a)\| da \\ + \int \left\| \nabla \ell(\mathbf{a}_k^t; \mathbf{w}_S) \right\| \|(q_k^t(a) - p_k^t(a))\| da \quad (23)$$

$$\geq -\sqrt{G} \frac{1}{K} \sum_{k=1}^K (\varepsilon_k^t + \delta_k^t) \quad (24)$$

Based on Assumption 4

$$C_2 \geq -\mathbb{E} \left[\sqrt{G} \frac{1}{K} \sum_{k=1}^K (\boldsymbol{\varepsilon}_k^t + \boldsymbol{\delta}_k^t) \right] \quad (25)$$

$$\geq -\sqrt{G} \frac{1}{K} \sum_{k=1}^K (H_1 + H_2) \quad (26)$$

$$\geq -\sqrt{G}(H_1 + H_2) \quad (27)$$

C_3 is considered based on $\mathbb{E}[U^T V] \leq \frac{1}{4} \mathbb{E}[\|U\|^2] + \mathbb{E}[\|V\|^2], \forall U, \forall V$. Then

$$C_3 = \|\mathbb{E}[\nabla F_S(\mathbf{w}_S^t)^T X]\| \quad (28)$$

$$\leq \frac{1}{4} \mathbb{E}[\|\nabla F_S(\mathbf{w}_S^t)\|^2] + \mathbb{E}[\|X\|^2] \quad (29)$$

$$\leq \frac{1}{4} \mathbb{E}[\|\nabla F_S(\mathbf{w}_S^t)\|^2] + H_1 + H_2 \quad (30)$$

where Equation 30 can be derived by following similar steps as shown in Equation 13 to Equation 24.

By using the results of C_1 , C_2 , and C_3

$$B_1 = \mathbb{E} \left[\nabla F_S(\mathbf{w}_S^t)^T \left(\frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right) \right] \quad (31)$$

$$\geq \frac{3}{4} \mathbb{E}[\|\nabla F_{S,k}(\mathbf{w}_S^t)\|^2] - (\sqrt{G} + 1)(H_1 + H_2) \quad (32)$$

B_2 is considered

$$B_2 = \mathbb{E} \left[\left\| \frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right\|^2 \right] \quad (33)$$

$$= \left\| \frac{1}{K} \sum_{k=1}^K \nabla F_{S,k}(\mathbf{w}_S^t) \right\|^2 \quad (34)$$

$$\leq \frac{1}{a} \sum_{k=1}^K \|\nabla F_{S,k}(\mathbf{w}_S^t)\|^2 \quad (35)$$

$$\leq \frac{G}{b} \quad (36)$$

where a is obtained from the Cauchy-Schwarz inequality and b is obtained from Assumption 2 presented in the article.

By substituting the results of Equation 32 and Equation 36

$$\mathbb{E}[F_S(\mathbf{w}_S^{t+1})] \leq \mathbb{E}[F_S(\mathbf{w}_S^t)] - \frac{3}{4}\eta_t \mathbb{E}[\|\nabla F_{S,k}(\mathbf{w}_S^t)\|^2] + \eta_t(\sqrt{G} + 1)(H_1 + H_2) + \frac{L}{2}\eta_t^2 G \quad (37)$$

By summing up Equation 37 for all global rounds $t = 0, 1, \dots, T - 1$

$$\begin{aligned} \mathbb{E}[F_S(\mathbf{w}_S^T)] &\leq \mathbb{E}[F_S(\mathbf{w}_S^0)] - \frac{3}{4} \sum_{t=0}^{T-1} \eta_t \mathbb{E}[\|\nabla F_{S,k}(\mathbf{w}_S^t)\|^2] \\ &\quad + \sum_{t=0}^{T-1} \left(\eta_t(\sqrt{G} + 1)(H_1 + H_2) + \frac{L}{2}\eta_t^2 G \right) \end{aligned} \quad (38)$$

Finally, from $F_S(\mathbf{w}_S^*) \leq \mathbb{E}[F_S(\mathbf{w}_S^T)]$

$$\begin{aligned} \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \eta_t \mathbb{E}[\|\nabla F_{S,k}(\mathbf{w}_S^t)\|^2] &\leq \frac{4(F_S(\mathbf{w}_S^0) - F_S(\mathbf{w}_S^*))}{3\Gamma_T} \\ &\quad + \frac{1}{\Gamma_T} \sum_{t=0}^{T-1} \left(\eta_t(\sqrt{G} + 1)(H_1 + H_2) + \frac{L}{2}\eta_t^2 G \right) \end{aligned} \quad (39)$$

which completes the proof.

Appendix B

Experimental Trials and Significance of The Results

In this thesis, the number of times the experiments were repeated varies. It is based on the setup of each experiment and the substantial training time of FL training on physical devices were taken into account. Unlike many FL studies reported in the literature that use simulations on GPUs [75], this work studies and mitigates the practical challenges of using IoT devices to reflect real-world conditions. Experiments involving a single FL round were repeated multiple times to capture variability. However, full FL training (which requires hundreds of training rounds) was conducted only once due to the time and resource constraints. For example, running all the experiments shown in Table 5.3 in Chapter 5 required approximately 144 hours under ideal conditions. In practice, it required over two weeks due to unexpected failures, such as device crashes and network disconnections. The full FL training experiments in this thesis were, therefore, reported based on a single run due to the significant time and resource demands. However, experiments involving a single FL round were repeated multiple times to account for variability.

The details of the experimental repetitions are shown below:

1. In Chapter 3, most experiments involving a single-round FL training and RL training were repeated three times. This includes the experiments presented in Table 3.4, Table 3.5, Figure 3.3, Table 3.7, Figure 3.4, Figure 3.5, Figure 3.6 and Figure 3.9. However, experiments requiring the entire FL training, such as those shown in Figure 3.7, Figure 3.8 and Figure 3.10 are reported based on a single run.
2. In Chapter 4, all results are based on three runs, except for the pre-training on different datasets shown in Table 4.11. However, it is worth noting that the final accuracy of the FL training was obtained using the simulation-based testbed described in Section 4.5.1.

3. In Chapter 5 and Chapter 6, all results require a full run of FL training on the physical devices and are therefore based on a single run.

The thesis demonstrates the significance of the results in alignment with commonly reported formats in FL literature, utilizing standard deviation to represent the variance in the reported outcomes. Specifically, the standard deviation is reported in tables summarizing results from multiple runs, and error bars representing the standard deviation are presented in the corresponding figures. Statistical significance methods, such as p-value calculations, are not adopted, as they typically require a large number of samples to compute p-values accurately. However, only three experimental runs were carried out in the experiments reported in this thesis.