Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

HIGH-PERFORMANCE SCHEDULING OF DEEP LEARNING TASKS IN COLLABORATIVE EDGE COMPUTING

MINGJIN ZHANG

PhD

The Hong Kong Polytechnic University

2024

The Hong Kong Polytechnic University

Department of Computing

High-performance Scheduling of Deep Learning Tasks in Collaborative Edge Computing

Mingjin ZHANG

A thesis submitted in partial fulfillment of the requirements for

the degree of Doctor of Philosophy

May 2024

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

Signature: ____________________

Name of Student: Mingjin ZHANG

Abstract

In recent years, deep learning (DL) models and algorithms have been extensively used in various applications. Traditionally, DL tasks, including model training and inference, are usually performed on centralized cloud servers in data centers due to their powerful and abundant computing resources. However, the computation on the cloud usually suffers from high communication costs, long response latency, and privacy concerns. In this case, edge computing was proposed recently to migrate the computation and services from the remote cloud to the network edge on edge nodes, closer to the data sources.

However, performing deep learning model training and inference tasks at the edge is challenging. While the deep learning models are usually computation-intensive and resource-greedy, the computation resources on edge nodes are constrained, which may not be able to burden the training and inference tasks. Besides, the data are usually on geo-distributed edge nodes, which are from different stakeholders and have heterogeneous networking and computation capabilities. Furthermore, deep learning tasks have inner characteristics. There are various model training paradigms. Many hyper-parameters, such as batch size, learning rate, and aggregation frequency, can affect the model performance. Also, many AI applications involve a set of dependent DL models, making it more complex.

To address the above problems, this study aims to schedule the AI model training and inference tasks among heterogeneous edge devices and cloud servers to reduce

latency while preserving accuracy by jointly considering the edge resources and the characteristics of the deep learning tasks. This thesis makes the following three contributions.

First, design and develop ENTS, an edge-native task scheduling system runtime, to schedule the deep learning tasks among large-scale, geo-distributed, and heterogeneous edge nodes. While existing task scheduling systems for edge computing consider only computation resources, ENTS collaboratively schedules computation and networking resources while considering both the DL task profile and resource status.

Second, schedule the model training tasks in edge computing to reduce overall training time. Existing distributed machine learning framework at edge suffers from the heterogeneous and constrained edge resources. We propose a novel federated learning framework that adaptively splits and schedules the training tasks among the heterogeneous edge nodes and the FL server for acceleration without compromising accuracy.

Third, schedule the inference tasks among edge nodes to achieve low latency and high system throughput. While existing methods focus on the cloud-edge collaboration, and seldom consider the collaboration among edge nodes, we develop a collaborative edge intelligence platform to enable edge nodes to share the data and computation resources for performing latency-sensitive video analytics tasks.

In summary, this thesis systematically investigates the requirements and solves the deep learning tasks scheduling problem for achieving high-performance model training and deployment in edge computing. The proposed framework and solutions address the challenging issues resulting from constrained and heterogeneous edge resources, and complexity of DNN model training and inference tasks. We also outline future directions, including decentralized scheduling framework for edge resources from multiple stakeholders and general programming models for efficient workload partition of deep learning tasks.

Publications Arising from the Thesis

1. Mingjin Zhang, Jiannong Cao, Jon Crowcroft, Lei Yang, Yuvraj Sahni, Shan Jiang, Yinfeng Cao, "EdgeSplit: Resource-aware Task Scheduling for Accelerating Distributed Model Training in Heterogeneous Edge Computing", manuscript submitted to *IEEE Transactions on Parallel and Distributed Systems.*

2. Mingjin Zhang, Jiannong Cao, Lei Yang, Shan Jiang, "Resource-efficient Parallel Split Learning in Heterogeneous Edge Computing", in *International Conference on Computing, Networking and Communication (ICNC 2024)*.

3. Mingin Zhang, Jiannong Cao, Lei Yang, Liang Zhang, Yuvraj Sahni, Shan Jiang, "ENTS: An Edge Native Task Scheduling System for Collaborative Edge Computing", in *The Seventh ACM/IEEE Symposium on Edge Computing (SEC)* (2022).

4. Mingin Zhang, Jiannong Cao, Yuvraj Sahni, Qianyi Chen, Shan Jiang, Lei Yang, "Blockchain-based Collaborative Edge Intelligence for Trustworthy and Real-time Video Surveillance", in *IEEE Transactions on Industrial Informatics* (2022).

5. Mingin Zhang, Jiannong Cao, Yuvraj Sahni, Qianyi Chen, Shan Jiang, Tao Wu, "EaaS: A Service-Oriented Edge Computing Framework Towards Distributed Intelligence", in *IEEE Congress on Intelligent and Service-Oriented Systems Engineering* (2022).

6. Yinfeng Cao, Jiannong Cao, Dongbin Bai, Zhiyuan Hu, Kaile Wang, Mingjin Zhang, "PolyVerse: An Edge Computing-Empowered Metaverse with Physical-to-Virtual Projection", in *IEEE International Conference on Intelligent Metaverse Technologies & Applications (iMETA2023).*

7. Xiangchun Chen, Jiannong Cao, Zhixuan Liang, Yuvraj Sahni, Mingjin Zhang, "Digital Twin-assisted Reinforcement Learning for Dynamic Microservice Offloading in Edge Computing", in *The 20th IEEE International Conference on Mobile Ad-Hoc and Smart Systems (MASS)* (2023).

8. Lei Yang, Yanyan Lu, Jiannong Cao, Jiaming Huang, Mingjin Zhang, "E-Tree Learning: A Novel Decentralized Model Learning Framework for Edge AI", in *IEEE Internet of Things Journal* (2021).

Acknowledgments

Time really flies. It feels like just yesterday when I arrived in Hong Kong in 2019, yet here I am, five years later, at the end of an enriching and enlightening journey. At this moment, I am overwhelmed with profound joy and gratitude for the knowledge and skills I have gained, particularly in the field of edge intelligence. This area of study aims to enable faster and more secure artificial intelligence closer to the data source, ultimately achieving ubiquitous intelligence. My journey through this cutting-edge field has not only been academically fulfilling but has also ignited a deeper passion for exploring the frontiers of science and technology.

Apart from the joy of getting knowledge, my heart is brimming with gratitude.

First and foremost, I want to thank my supervisor, Professor Jiannong Cao. His guidance has been instrumental in shaping my understanding of what constitutes high-quality and impactful research. He has consistently encouraged me to think big and tackle challenging problems.

Second, I want to thank my mentor Professor Jon Crowcroft at the University of Cambridge. I am deeply appreciative of his kind guidance and encouragement. My gratitude also extends to friends there for their daily discussion and companion. They are Yilei Liang, Guoliang He, Andrew Jeffery, Chris Jensen, Justas Brazauskas, Roman Kolcun, and Vadim Safronov. The six months I spent there were filled with invaluable experiences. It is the happiest and most comfortable period of my PhD journey.

Third, I extend my heartfelt thanks to my co-authors at IMCL including Dr. Lei Yang, Dr. Sahni Yuvraj, Dr. Qianyi Chen, Dr. Shan Jiang, Mr. Yinfeng Cao, and Mr. Xiangchun Chen. Your collaboration and insights have been vital in enriching my research work. The synergy of working together on complex problems has not only led to successful outcomes but has also been a source of great learning and enjoyment.

Fourth, thanks to my peers and colleagues in our research group IMCL. They are Dr. Yuqi Wang, Dr. Wengen Li, Dr. Zhuo Li, Dr. Jia Wang, Dr. Yanni Yang, Dr. Jiaxing Shen, Dr. Yu Yang, Dr. Ruosong Yang, Dr. Liang Zhang, Dr. Zhixuan Liang, Dr. Shuaiqi Liu, Dr. Zhiyuan Wen, Mr. Zhiyuan Hu, Ms Esther KU. Your support and camaraderie have been indispensable. The mutual respect, encouragement, and intellectual stimulation has been a cornerstone of this wonderful journey.

Last but certainly not least, I express my deepest appreciation to my parents and family. Your unwavering support, faith, and love have been my strongest pillars. You have been my constant source of motivation and strength, and this achievement is as much yours as it is mine.

Table of Contents

Abstract **i**

Publications Arising from the Thesis **iii**

Acknowledgments **v**

List of Figures **xi**

List of Tables **xiv**

1 Introduction **1**

1.1 Background and Motivations 1

1.2 Research Objectives and Framework 3

1.3 Thesis Organization 5

2 Literature Review **7**

2.1 Task scheduling in Edge Computing 7

2.2 Model Training at Edge 9

2.3 Model Inference at Edge 11

3 **Edge-native Task Scheduling System** **15**
3.1 Overview 15
3.2 Background and Motivations 18
3.2.1 Kubernetes Scheduler 19
3.2.2 A Motivating Example 20
3.3 System Overview 22
3.3.1 Design Goals 22
3.3.2 System Architecture 23
3.4 System Design 25
3.4.1 Application Development and Profiling 25
3.4.2 Collaborative Task Scheduling 27
3.4.3 Distributed Task Execution 29
3.5 Collaborative Task Scheduling with Data Streaming Applications . . 31
3.5.1 System Model. 31
3.5.2 Problem Formulation 32
3.5.3 Proposed Solution 33
3.5.4 Online Scheduling 36
3.6 Experimental Results 40
3.6.1 Experimental Setup 40
3.6.2 Results and Analysis 42
3.7 Conclusion 47

4 Scheduling Model Training Tasks **48**

4.1 Overview 48

4.2 Motivations 52

4.3 Framework of EdgeSplit 53

4.3.1 EdgeSplit Framework 54

4.3.2 Challenges 57

4.4 Methodology 58

4.4.1 Determine the Best Partition Points 58

4.4.2 Alleviate Memory Overhead 64

4.5 Experimental Evaluation 67

4.5.1 Experimental Setup 67

4.5.2 Results and Analysis 70

4.6 Conclusion 77

5 Scheduling Model Inference Tasks **78**

5.1 Overview 78

5.2 Related Work 82

5.3 System Design 84

5.3.1 System Model 84

5.3.2 System Components 85

5.4 Joint Stream Mapping and Task Scheduling for Pedestrian Re-identification 86

5.4.1 Pedestrian Re-identification Pipeline 87

5.4.2 Motivations of Joint Stream Mapping and Task Scheduling . . 89
5.4.3 Problem Formulation 91
5.4.4 Optimization Solution 93
5.5 Implementation and Performance Evaluation 96
5.5.1 System Implementation 96
5.5.2 Evaluation Metrics and Experimental Settings 98
5.5.3 Influence of Number of Edge Devices 100
5.5.4 Influence of Dynamic Workload 101
5.5.5 Effects of Bandwidth 103
5.6 Conclusion 104

6 Conclusion and Future Directions 105

6.1 Conclusion 105
6.2 Future Research 106

References 108

List of Figures

1.1 System Structure of High-performance Scheduling of Deep Learning Tasks in Collaborative Edge Computing 4

3.1 Components of Kubernetes System 19

3.2 A Motivating Example of Collaborative Task Scheduling 21

3.3 Architecture of the ENTS System 23

3.4 Code Snippet of User Application 26

3.5 Code Snippet of Application Configuration 27

3.6 ENTS Task Scheduling Workflow 28

3.7 Collaborative Task Scheduling Strategy 29

3.8 Bandwidth Allocation and Customized Routing of Network Manager 30

3.9 Application Graph of Object Attributes Recognition 40

3.10 Test Environment of ENTS 42

3.11 a) Impact of the number of edge nodes on average throughput with average bandwidth 1Mbps. b) Impact of the number of edge nodes on average throughput with average bandwidth 10Mbps. c) Impact of the number of edge nodes on average waiting time. d) Impact of the number of submitted jobs on average throughput. e) Impact of the number of submitted jobs on average waiting time. f) Impact of average bandwidth on average throughput. 44

4.1 Overview of EdgeSplit. Edge devices train part of the full model with different depths adapting to local resources and offload the rest of model training task to the FL server for acceleration. 50

4.2 Federated learning on heterogeneous edge devices suffers from the straggler issue. 52

4.3 Framework of EdgeSplit. It consists of three stages: model and device profiling, task scheduling optimization, and online split edge training. 54

4.4 Output data size varies across DNN layers. Random model partition can hardly generate optimal execution latency. 57

4.5 EdgeSplit with SplitPipe mechanism to reduce memory overhead by pipeling and reusing duplicated layers of partial models. 65

4.6 Different task execution strategy of SplitPipe. Backward-First strategy reduces memory overhead by releasing memory space of cached intermediate variables. 66

4.7 Hybrid testbed with physical and emulated devices 68

4.8 Convergence Time v.s. Accuracy. EdgeSplit achieves fast convergence without accuracy loss 72

4.9 Impact of Number of Edge Devices 73

4.10 Impact of Total Bandwidth 74

4.11 Impact of number of edge devices on system overhead. 77

5.1 System model of distributed and collaborative edge intelligence system for video surveillance 83

5.2 Model pipeline and task offloading of pedestrian re-identification . . . 87

5.3 A motivation example of joint stream mapping and task scheduling: (a) random camera streams and local execution; (b) scheduling camera streams and local execution; (c) random camera streams and offloading ML models; (d) joint scheduling of camera streams and offloading ML models. 90

5.4 Demo of pedestrian re-identification (with mosaics for anonymization) 97

5.5 Throughput vs. number of devices 100

5.6 Workload dynamics in real-world deployment 101

5.7 Throughput vs. dynamic workload 102

5.8 Latency vs. dynamic workload 102

5.9 Throughput vs. bandwidth 103

5.10 Latency vs. bandwidth 103

List of Tables

3.1 Specifications of Physical Devices 43

4.1 List of notations 59

4.2 Specifications of heterogeneous physical devices 69

4.3 One-round training time on physical devices (s) 71

4.4 Comparison of per-round training time. Best partition points and acceleration ratio to vanilla FL are given. 71

4.5 Performance of EdgeSplit with different serverpipe mechanisms. Memory: the active peak memory footprint within the training; RoundTime: the average round time in the training. 75

5.1 Comparison of the related work of video analytics 80

Chapter 1

Introduction

This research studies how to enable the high-performance execution of DNN models and applications in edge computing environment by scheduling and distribute the computation tasks among geo-distributed and heterogeneous edge nodes and cloud servers. In this chapter, we first describe the background of deploying DNN models and applications at the network edge and motivate our research. In Section 1.2, we introduce the research objective and system structure. Finally, we outline the organization of this thesis in Section 1.3.

1.1 Background and Motivations

Recent advances in deep learning have driven the development of applications with advanced analytics services, including computation vision [85] and natural language processing [93]. DL tasks, including model training and inference, are usually computation-extensive and resource-greedy, which are traditionally trained and deployed in the cloud with the data collected from end devices. Though the cloud-based deep learning has achieved great success, there are still several drawbacks: 1) long transmission latency. There is usually a long distance from the end devices to the cloud center,

which makes it less feasible to support the time-critical applications, such as VR Gaming [21], autonomous driving [66]. 2) privacy concerns. Since all the data at end devices will be sent to the cloud, there may be some security issues during the transmission and cloud storage. 3) high communication cost. Large amount of data transmission may lead to high bandwidth cost and network traffic. Thus, edge computing was proposed recently to migrate the computation and services from cloud to the network edge, where DNN models are trained and deployed locally on the edge nodes (such as bases stations, edge gateways, roadside units, .etc), closer to data sources.

Due to its great benefits of privacy protection and agile response speed, deep learning at edge has become the key enabling technology for various time-critical and mission-critical applications, e.g., real-time edge video analytics, autonomous driving, and intelligent manufacturing. However, deep learning at edge is vastly different from that of cloud. Servers at cloud are usually with powerful computation abilities. However, edge nodes are with constraint compute capacity, which may not be able perform the computing intensive and resource-greedy deep learning tasks solely. Hence, collaborative edge computing becomes a promising solution, where geo-distributed edge nodes and cloud servers collaborate to share computation resources and data to perform application tasks.

A fundamental problem is to efficiently distribute the deep learning tasks among collaborating geo-distributed edge nodes for meeting applications' performance requirements, such as latency, privacy, and throughput. However, it is nontrivial. First, computation resources of edge nodes and cloud servers are heterogeneous with different capabilities. Besides, different from the cloud server in data centers, which are connected by stable and high-bandwidth networks, edge nodes are usually connected with low-bandwidth and intermittent network. The scheduling of deep learning tasks at edge requires joint consideration and optimization of the computation and networking resources. Second, data collected from end devices are centralized stored in

the cloud. However, for distributed deep learning at edge, data distributes in various devices and may belong to different stakeholders. Such differences make it difficult to efficiently train and deploy machine learning models at edge. Third, deep learning tasks have inner characteristics. There are various model training paradigms, such as federated learning, gossip learning, hierarchical federated learning. Many hyper-parameters, such as batch size, learning rate, and aggregation frequency, can affect the model performance. Moreover, many AI applications involve a set of dependent DL models rather than a single model, making it more complex.

In the following, I sometimes use edge devices to refer to edge nodes. Moreover, the role of edge devices may change in different environments. For example, in smart wearable scenario, mobile phones act as edge device compared to smart watches, as watches may offload computation to phones. However, in mobile edge computing scenario, base station becomes edge devices, as mobile phones can offload computation to base stations. We will make clear the definition of edge devices in dedicated research work.

1.2 Research Objectives and Framework

In this thesis, we aims to study high-performance task scheduling framework, algorithms, and methods to efficiently distribute training and inference tasks among edge nodes and cloud servers, by jointly considering and optimizing the underlying edge resources and the characteristics of the deep learning tasks. To achieve the research objective, I designed a task scheduling system structure for collaborative edge computing. Within the system structure, I developed an edge-native task scheduling system runtime. Above the runtime, I designed resource-aware scheduling algorithms for AI training and inference tasks, respectively.

The system structure of collaborative scheduling for deep learning tasks is summa-

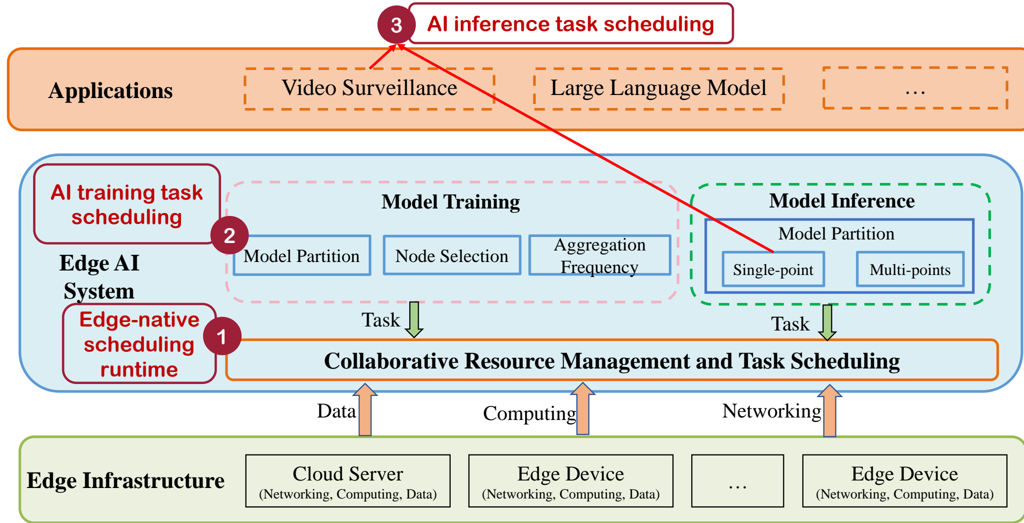


Figure 1.1: System Structure of High-performance Scheduling of Deep Learning Tasks in Collaborative Edge Computing

rized and illustrated in Fig. 4.3. There are three layers from bottom to top. The first layer is the edge infrastructure layer, where there are multiple edge nodes with localized data interconnected with each other via particular networks. In mobile edge computing, edge devices are MEC servers deployed behind base stations, and they are interconnected by the cellular back-haul networks. In industrial IoTs, the edge devices could be smart routers or switches with build-in high performance CPU/GPU cores. The edge devices are interconnected by wireless mesh network. The middle layer is edge AI system layer, which we are more focused on. In this layer, we enable edge devices to share computation resources and collaboratively schedule the training and inference tasks to reduce the inference latency and accelerate the model training in heterogeneous and constraint computing and networking resources. More specifically, we will study collaborative task scheduling system to jointly manage and schedule the computation and networking resources. Based on the scheduling system, we will study solutions to schedule model training and inference tasks, respectively. Algorithms and solutions in the middle layer will be used to support various applications at the top layer, such as real-time video surveillance and industrial IoT.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- In chapter 2, we review the literature related to this thesis, including task scheduling algorithms and systems, DNN model training, and DNN model inference in edge computing scenario.

- In chapter 3, we design and develop an edge-native task scheduling system to schedule the containerized deep learning tasks among geo-distributed edge nodes. Existing task scheduling systems are not dedicated for edge computing and neglect some edge-native features. They are not designed to optimize the performance-sensitive edge-native applications and lacks joint scheduling of the computation and networking resources. We hence develop ENTS by extending Kubernetes with the unique ability to collaboratively schedule computation and networking resources, which comprehensively considers both DNN task profile and resource status. We showcase the superior efficacy of ENTS with a case study on an edge video analytics applications empowered by DNN models. We mathematically formulate a joint task allocation and flow scheduling problem that maximizes the job throughput. We design two novel online scheduling algorithms to optimally decide the task allocation, bandwidth allocation, and flow routing policies, for maximizing the average application throughput.

- In chapter 4, we study the DNN model training task scheduling based on the scheduling framework. We propose EdgeSplit, a novel FL training framework to accelerate FL on heterogeneous and resource-constraint edge devices. EdgeSplit enables model training on low-resource edge devices by partitioning the model between edge devices and the FL server and leverages heterogeneous computation capabilities to achieve acceleration. To minimize the overall training time, we formulate a task scheduling problem to jointly decide the model partition

point and allocated bandwidth for each edge device. We propose a simple and efficient alternating algorithm to iteratively optimize the two decision variables for solving the problem.

- In chapter 5, we study the DNN inference task scheduling at edge. We design a distributed and collaborative edge intelligence (DCEI) approach for trustworthy and real-time video surveillance. In DCEI, geo-distributed edge devices collaborate by sharing data and computation resources to perform computation-intensive video analytics tasks. We formulate and solve a joint stream mapping and task scheduling problem to schedule video streams and machine learning models among edge devices to reduce task execution time. A pedestrian re-identification prototype is implemented and deployed based on DCEI with extensive performance evaluation, indicating the superiority of DCEI in terms of latency reduction and throughput improvement by leveraging collaboration among edge devices.
- Chapter 6 concludes this thesis with summarization of the main contents and our vision of the future research.

Chapter 2

Literature Review

In this chapter, we present the literature review on task scheduling and DNN model training and inference in edge computing scenario.

2.1 Task scheduling in Edge Computing

This section introduces the work of task scheduling in edge computing from two aspects, i.e., system and algorithm.

Task Scheduling System. Edge nodes has heterogeneous resource and software environment. To abstract the heterogeneous edge resources and provide a unified software environment for distributed task execution, virtualization mechanisms, i.e., virtual machine and container, has been proposed and studied for years . Container technology is highly suitable for edge computing, as it provides lightweight resource virtualization and enables fast application development and flexible service deployment over heterogeneous edge nodes. There are several industrial systems to schedule containerized applications, including Swarm [99], Kubernetes [71], and Mesos [35]. Among them, Kubernetes is the most popular and has established its leadership [10]. The default scheduler of Kubernetes (K8S) [71] is an online scheduler that implements

a greedy multi-criteria decision-making (MCDM) algorithm. MCDM scores the available nodes with pre-defined rules and selects the highest scoring node for scheduling. K8S is originally designed for cloud computing. Recently, there are some works that aim to tailor K8S to fit the edge computing environment. K3s [51] aims to simplify K8S and provide lightweight K8S distribution. KubeEdge [114] and OpenYurt extend the K8S capability to the edge by enabling the virtual network connection between edge servers and VMs in the cloud. However, those solutions do not change the core idea of task scheduling of Kubernetes. They are not application performance sensitive.

Task Scheduling Algorithm. Scheduling in edge computing has some edge-native features. First, edge applications have service-level-objectives (SLOs) of the performance during execution. Traditionally, SLOs in cloud computing, defines measure capacity guarantee of applications, e.g., available memory to a provisioned VM or container. However, in edge computing, edge-native applications usually have high requirements of the service level objectives, such as latency, throughout, and privacy. Edge computing is promised to provide low latency and real-time response and services. Second, data are distributed. Different from cloud server and cloud storage, edge devices and data are geo-distributed, whose network connections are limited and dynamic. The data and edge device locality have great influence to the optimal scheduling. Third, resources on edge node are limited. Many intelligent edge applications such as object detection and tracing, video analytics are complex and resource greedy. When deploying those applications at edge, they must be partitioned and deployed on the geo-distributed edge nodes, which leads to frequent communication and intermediate data transmission over the low-bandwidth edge network. Hence, task scheduling algorithms in edge computing needs to jointly consider the coupled data, networking, and computing resources and make optimized task scheduling strategies to meet edge application SLOs.

Sundar et al. [100] proposed a heuristic algorithm for scheduling dependent tasks

in a generic cloud computing system by greedily optimizing the scheduling of each task subject to its time constraint. Wang et al. [108] developed a deep reinforcement learning-based task offloading scheme, which leverages the off-policy reinforcement learning algorithm with a sequence-to-sequence neural network to capture the task dependency of applications. Nevertheless, they fail to consider the orchestration of the network flows [69], which necessarily results in network congestion and prolonged task completion time. There are some works [89, 90] optimizing the average task completion time and jointly considering the task allocation and flow scheduling.

Due to the complexity of making joint task scheduling decisions, deep reinforcement learning has recently been applied to solve the scheduling problem. Chen et al. [74] adopted deep Q-Learning to obtain the offloading strategy in an ultradense network. However, this algorithm has the long-term cost of delay in computation offloading. To reduce the longterm delay, Chen et al. [13] proposed a double deep Qnetwork (DQN)-based strategic computation offloading algorithm. Dinh et al. [18] focused on multi-user multi-edge-node task offloading problems using Q-learning in MEC. To address the problem of high energy consumption, Huang et al. [39] considered the system computing performance under either partial or binary offloading policy in the MEC network and proposed a Deep-Q Network (DQN) based task offloading and resource allocation algorithm for the MEC.

2.2 Model Training at Edge

A significant problem of edge learning is to train accurate models with the distributed data on multiple edge devices. Federated learning (FL) is one of the most popular solutions, which enables collaborative model training among various edge devices without privacy violation. However, Federated Learning suffers from the straggler issue. In each training round, the FL server is required to wait for the updated parameters of all the participants, which may lead to a long time of the model train-

ing. The straggler issue is mainly caused by the following reasons: 1) Heterogeneous network connection. The status, i.e., bandwidth and distances of the network links connecting the edge devices and the server are usually versatile. 2) Heterogeneous computation capacity. The time of local training may be vastly different due to the computation capacities of the edge devices. Moreover, some edge devices with limited memory may not have the ability to burden the training tasks due to the complexity and great footprint requirements of the ML models. Recent work in accelerating model training at edge can be roughly divided into two categories: algorithm level, and architecture level.

To improve the communication efficiency, previous works usually focus on reducing the total amount of transmission bits, e.g., communication compression that compresses the communication data exchanged between the worker and servers. Sparsification [111] [53] and quantization [1] [94] are the two mainstream for compressing communication data. Sparsification [63] [101] is a selection method that only a fraction of the gradient is sent by the workers in each iteration. The top-k dimensions with the largest absolute value are sent by workers in [63]. Stich et al. propose the sparsified SGD with memory, a family of sparse schemes with both convergence and practical performance guarantees. In this approach, a concise convergence anslysis of SGD with k-sparsification or compression is given. Beyond these methods, DoubleSqueeze [101] further proposes sparsifying both the gradients sent from workere to the server and the model updates sent from the server to the workers. To reduce the uplink communication costs, the work in [52] proposed structured and sketched updates method, and compression techniques were adopted to reduce parameter dimension in this work. In [109], gradient selection and adaptive adjustment of learning rate were used for efficient compression.

Other works also consider accelerating federated learning in practical wireless edge network. [83] proposed a partial synchronization parallel schema to enable the parallel model training on the mobile clients in a relay-assisted wireless edge network. They

aim to optimize the communication efficiency for gradient aggregation and model synchronization among large-scale devices. [86] jointly optimize the batch size selection and communication resource allocation to accelerate DNN training in wireless federated edge learning systems. Apart from reducing the size of the transmission, [67] also adopts momentum gradient descent (MGD) to replace the commonly used stochastic gradient descent to improve the convergence rate. The author shows that the MGD can accelerate the convergence rate of the federated model training.

Model splitting is one of the representative architecture-level methods. Splitting learning [27] partitions the deep neural network into two parts, where the shallow part is trained on the client and the deep part is trained on the server. Such schema can leverage the heterogeneous computing capacity of the clients and server to reduce the model training time. However, the training is performed in a sequential manner. JointDNN [22] exploits the layer granularity of DNN architecture for run-time partitioning in edge-cloud environment. Recently, [105] integrates splitting learning and federated learning to support parallel and distributed model training. But they do not consider how to split the model for faster convergence and they do not consider the network model in practical federated learning system. Though, these works leverage the model splitting to handle the device heterogeneity for model training acceleration, they did not consider how to split the neural network models with respect to the heterogeneous computation and communication resources.

2.3 Model Inference at Edge

The objective of model inference at edge is to accelerate the inference with satisfied accuracy. Generally speaking, large and complex neural network models always show good performance than shallow models due to the powerful representation ability. However, complex neural networks are always resource-greedy and computation-intensive. It's challenging to deploy the complex AI models on resource-constraint

edge devices. To tackle the problem, several existing methods are proposed, e.g., model compression [16, 30, 36, 37, 42, 84, 95], model partition [50, 59, 132, 134], and model early exit [58,102,103], to reduce the complexity of the model executed on the edge devices.

Han et al. [30] learn important weights and connections in the neural network. They prune those redundant connections to get concise model based on the fact that many weights are closer to zero. [84] use weight quantization to represent the weights with only a few bits so that the size of stored weights can be reduced. Different from the aforementioned model-level work, [95] leverage knowledge-level distillation [36] to train a small DNN, which imitate a larger and powerful DNN but with less complex model structure and parameters. Model partition tries to utilize the heterogeneous computing capacity among edge devices and cloud server to accelerate the model inference. Generally, it split the model into two parts with the shallow part running on the edge device and the deep part executing on the cloud. There are generally two kinds of partition, horizontal partition and vertical partition. The horizontal partition [50] splits the neural network model in a layer-wise with the output of the former layers transmitted to the cloud for the execution of the latter layers. The selection point depends on both the model characteristics and the dynamic factors, such as computing capacity of the edge device, wireless network conditions. The vertical partition tries to explore the inference parallelism by distribute the computation tasks into several parts which can run simultaneously [134]. Model early exit leverage the trade-off between the model accuracy and the processing delay. Generally, a deeper model with more layers learns informative feature representations and output results with higher confidence. However, As DNNs grow larger and deeper, the computation costs become unaffordable for edge devices to run real-time and energy-sensitive DL applications. In this case, model early exit inserts some branches at the exit point of the neural network model. If the confidence at a exit point is higher than a threshold, the model will not execute the rest part and exit from the inserted branches. Li

et al. [58] proposed a three-stage end-edge co-inference framework to automatically and intelligently selects the best partition point and exit point of a DNN model to maximize the accuracy while satisfying the requirement on the execution latency. They first train a regression model to predict the performance of different types of DNN layer and train the branch network in the offline training stage. At the online optimization stage, a DNN optimizer selects the best partition point and early-exit point of DNNs to maximize the accuracy while providing performance guarantee on the end-to-end latency. Finally, at the co-inference stage, the edge server will execute the layer before the partition point and the rest will run on the mobile device. Compared to directly offloading DL computations to the cloud, this approach has lower communication costs and can achieve higher inference accuracy then those of the pruned or quantized DL models on edge devices.

Though the proposed method has largely reduce the complexity of the AI models, Few of them consider the run-time dynamics, e.g., the variation of the communication bandwidth and computation resources of the edge devices. Constraint and dynamic resources are key characteristics of the edge devices, which affects the run-time performance of the model inference a lot. For instance, the fluctuations on communication bandwidth between edge and cloud may make the workload offloading strategy sub-optimal, incurring additional inference delay. The changes of data characteristics, e.g., switching from bright scene to dark scene for an image object detector, will cause dynamic end-to-end delays for the model early exit method since the inference path may exit at different branches.

Extending the static model inference acceleration methods, several efforts [23, 24, 65, 133] proposed dynamic neural networks that allow selective execution to improve DNN compute efficiency. NestDNN [24] considers multi-tenant DL models running on an edge device, which leads to dynamic resource competition. With the model pruning and recovery scheme, it transforms the DL model into a set of descendant models, in which the descendant model with fewer resource requirements shares its model

parameters with the descendant model requiring more resources, making itself nested inside the descendent model requiring more resources without taking extra memory space. In this way, the multi-capacity model provides variable resource-accuracy trade-off with a compact memory footprint, hence ensuring efficient multi-tenant DL on the resource-constrain edge device. D2NN [65] optimizes dynamic resource-accuracy trade-offs, while its complicated network structure incurs significant memory overhead, making it ill-suited for resource-constrained platforms. Existing approaches usually requires careful consideration of model architecture and the optimization of a large set of model parameters, e.g., the model partition point in workload offloading mechanism and compression level in model compression. Such optimization usually leads to high computation overhead, which makes them less feasible to the online manner. To tackle the problem, EdgeML [133] leverage AutoML [25,118] to automate the decision making in model partition and model early exit for achieving desirable latency-accuracy-energy trade-off under dynamic run-time conditions. They train a reinforcement learning model [75] offline with the profiled branch insertion and model partition points, as well as the response time and the energy consumption. On the online inference stage, the reinforcement learning model dynamically control the model execution, i.e., determining the partition and exit points, to fit the run-time dynamics.

Though the above work considers some characteristics of the run-time dynamics, most of them only consider the edge-cloud collaboration, e.g., offloading the computation tasks to the cloud servers. They neglect the edge-edge collaboration, where multiple edge devices can collaborate with each other by sharing both data and computation resources. Edge-edge collaboration can reduce the data transmission latency and avoid the privacy violation in some degree. It leaves an open research area to jointly consider the computation, data, and networking resources among multiple edge devices.

Chapter 3

Edge-native Task Scheduling System

In this chapter, we design and develop an edge-native task scheduling system for collaborative edge computing. This chapter is organized as follows. We present an overview of this work in Section 3.1. Section 3.2 provides further motivation for our work by discussing the significance of collaborative task scheduling, which jointly orchestrates the coupled edge resources. Section 3.3 overviews the design goals and main components of the system. Section 3.4 presents the design details. In Section 3.5, we formulate the collaborative task scheduling problem and propose online solutions for a video analytics application. Section 3.6 shows the evaluation results. Finally, Section 5.6 concludes this chapter.

3.1 Overview

Collaborative edge computing (CEC) is a popular and new edge computing paradigm enabling sharing of data, computation, and networking resources among geo-distributed and heterogeneous edge nodes, including edge servers, edge gateways, and mobile phones [126]. CEC is promising and beneficial because it provides higher reliability and lower latency and facilitates collaboration among different stakeholders [78].

Task scheduling is a fundamental problem of collaborative edge computing, which refers to the arrangement of the user-generated application tasks to the heterogeneous edge nodes by deciding when, where, and how to offload the tasks and how to manage and utilize the underlying computation, storage, and networking resources [72]. Many works have investigated the task scheduling problems in collaborative edge computing [74]. Recently, there has been a trend of scheduling containerized application workloads among the geo-distributed and heterogeneous edge infrastructure [122]. This is because the container technology provides lightweight resource virtualization and enables fast application development and flexible service deployment over heterogeneous edge nodes.

There are several solutions to orchestrate containerized applications, such as Swarm [99], Kubernetes [71], and Mesos [35]. Among them, Kubernetes has established its leadership [10]. Many works have studied optimizing the Kubernetes scheduler for the cloud environment, where cloud servers with abundant computation resources are interconnected with a high-bandwidth and stable network in a data center [9]. However, Kubernetes is designed not dedicated to edge computing, neglects the unique features of edge nativeness, and lacks adequate support for edge-native applications [31].

First, edge-native applications are usually performance-aware, demanding high throughput, low latency, and strict privacy. The Kubernetes scheduler is mainly designed to ensure resource provision of workloads, such as the capacity of requested memory and CPU cores. It lacks support to meet the performance requirements of edge-native applications. Second, edge-native applications are with inner dependencies. Many intelligent edge applications are resource-greedy and complex, consisting of lots of inter-dependent components which are usually deployed to multiple edge nodes considering the constraint resource of a single node. However, the Kubernetes scheduler fails to consider the application's inner structure. Third, the data, computation, and networking resources are heterogeneous and coupled with each other. Application deployed on heterogeneous edge nodes experiences distinct performance, and the

coupled resources require joint orchestration. However, Kubernetes concentrates on orchestrating computation resources without jointly considering the data locality and networking resources, which may lead to underutilized resources and poor performance of workloads. Though some works [88] [112] consider the inner dependencies of workloads and the computation resources among edge nodes for optimizing the application performance, they fail to consider the data locality and resource heterogeneity.

In this chapter, we designed and developed ENTS, the first edge-native task scheduling system, to manage the geo-distributed and heterogeneous resources of edge infrastructures and enable efficient task scheduling among distributed edge nodes to optimize application performance. ENTS is developed based on Kubernetes, allowing Kubernetes to collaboratively schedule computation and networking resources considering both job profile and resource status. Specifically, to parse the inner dependencies of the user-submitted jobs, we adopt a data flow programming model, where each task in a job is programmed as a functional module. A profiler is designed to profile the job's execution time on heterogeneous edge nodes. The job profile information will later be used to facilitate efficient task scheduling. We also developed a network manager to manage the networking resources, which collaborates with the Kubernetes original components to jointly orchestrate the coupled resources under the coordination of a newly designed collaborative online scheduler. The scheduler runs the intelligent scheduling algorithms to generate the task scheduling policies to optimize the application performance.

To showcase the efficacy of ENTS, we formulate a joint task allocation and flow scheduling problem for data streaming applications as a case study. The problem is a mixed integrated non-linear problem proven to be NP-hard [69]. We design two online algorithms to solve the problem, which decides how to partition the job, where to allocate the tasks, and how to allocate the routing path and bandwidth for intermediate data flow to optimize the average job throughput. The efficacy of the

proposed system is illustrated by developing a real-world testbed for a representative edge video analytics application, namely, object attribute recognition. We develop a real-world hybrid testbed with both physical and virtual edge nodes to evaluate the system even in large scale. Online jobs will continuously arrive and be partitioned and scheduled among the edge nodes. We have comprehensively evaluated the performance of the designed system by comparing it with the state-of-the-art regarding different metrics, including average job throughput and average waiting time. The evaluation results show that our edge-native task scheduling approach improves the performance significantly.

The main contributions of this work are as follows:

- We design and develop ENTS to manage the data, computation, and networking resources in the heterogeneous geo-distributed edge infrastructure. ENTS is the first work to jointly manage coupled edge resources for optimizing the performance of edge-native applications.

- We formulate a joint task allocation and flow scheduling problem for data streaming applications and propose two online algorithms to solve the problem.

- We evaluate the performance of proposed solutions in a real-world testbed with a video analytics application. The experimental results indicate the superiority of ENTS over the baseline approaches in terms of higher job throughput and lower latency.

3.2 Background and Motivations

In this section, we introduce some background knowledge of the Kubernetes scheduler and illustrate the motivations for designing ENTS through some concise examples.

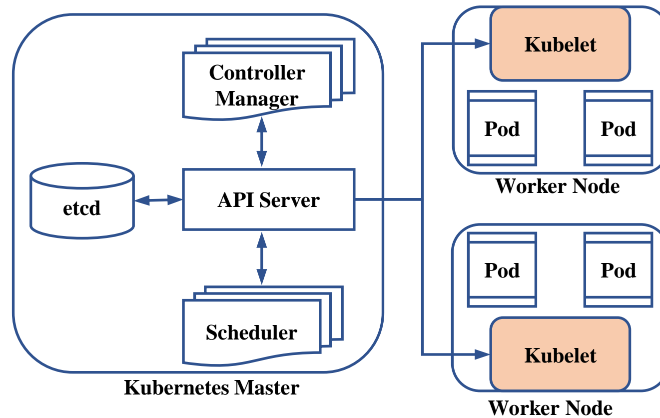


Figure 3.1: Components of Kubernetes System

3.2.1 Kubernetes Scheduler

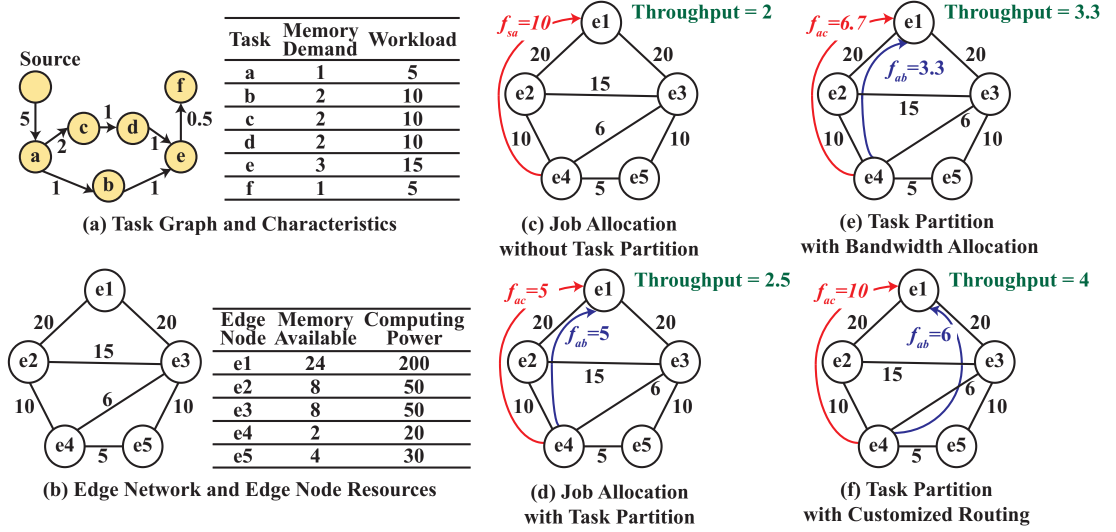
Fig. 3.1 depicts the components of Kubernetes with a master-client architecture. There is at least one centralized master managing resources and scheduling containerized workloads across multiple worker nodes (clients). The pod is the basic unit of Kubernetes to schedule the workload. A pod can contain one or more containers. There are mainly four components in the master node. The API server is an entry point to manage the whole cluster, providing services via Restful APIs. Components communicate and interact with each other through the API server. Etcd is a key-value pair distributed database that records the cluster status, such as node resource availability, location, states, and namespace. The scheduler is responsible for scheduling pods. It parses the operational requirements of pods and binds a pod to the best fit node. The controller manager is responsible for monitoring the overall state of the cluster. It launches a daemon running in a continuous loop and is responsible for collecting cluster information. Kubelet is the node agent in the clients. It is responsible for reporting events and resource usage and managing containers.

When scheduling user-submitted workloads, the scheduler first takes a pod pending to be scheduled from the etcd database and then binds the pod to the corresponding client node according to the pre-defined scheduling policies. The scheduling policy

is sent to Kubelet on the client nodes via the API server. After receiving the policies, Kubelet lunches the pods and monitors the pods' execution status. Kubernetes scheduler adopts a multi-criteria decision-making algorithm in two stages. The first stage is node filtering, where the scheduler will select candidate nodes capable of running the pods by applying a set of filters, such as memory and storage availability. Those filters are also known as predicates. The second stage is node scoring. It scores all the candidates based on one or more strategies, such as LeastRequestedPriority, which allocates pods to the nodes with the least computation resource consumption, and BanlancedResourceAllocation, which balances the resource consumption among edge nodes. Those strategies are known as priorities. The scheduler will allocate a pod to the node with the highest score.

3.2.2 A Motivating Example

As shown in Fig. 3.2, this section presents a motivating example articulating the benefits of collaborative task scheduling, which jointly considers the coupled data, computation, and networking resources in edge computing scenarios. The problem is to allocate the application with dependent tasks, shown in Fig. 3.2(a), to a set of edge nodes, shown in Fig. 3.2(b), such that the job throughput is maximized. Fig. 3.2(a) shows the task graph of the job modeled as a directed acyclic graph. There are 6 tasks in the job, and the weight of the link between tasks indicates the volume of the dependent data. Fig. 3.2(a) also shows the memory demand and workload of each task. We assume that the total memory demand and workload are the sum of tasks, i.e., 11 and 55, respectively. Note that the job is a streaming application, where input data continuously arrives from the source, i.e., edge node $e4$. The amount of the input data is 5. In Fig. 3.2(b), there are 5 edge nodes $\{e1, e2, e, e4, e5\}$. The weight of the link between the edge nodes indicates the bandwidth. Similarly, the table in Fig. 3.2(b) shows the available memory and computing power of the edge nodes in the network.



Task	Memory Demand	Workload
a	1	5
b	2	10
c	2	10
d	2	10
e	3	15
f	1	5

Edge Node	Memory Available	Computing Power
e1	24	200
e2	8	50
e3	8	50
e4	2	20
e5	4	30

Figure 3.2: A Motivating Example of Collaborative Task Scheduling

Fig. 3.2(c) shows the job allocation strategy without task partition, where the job is scheduled to node $e1$ and the input data is transmitted from the source node $e4$ to $e1$ indicated by data flow f_{sa}, whose allocated bandwidth is 10 and routing path is $e4 \to e2 \to e1$. The throughput is calculated by $1/\max\{5/10, 55/200\} = 2$. Strategy in Fig. 3.2(c) is known as LeastRequestPriority, which are extensively used in Kubernetes. Differently, Fig. 3.2(d) partition the job, where task a is allocated to source node $e4$ and the rest tasks are allocated to node $e1$. Hence there are two data flows indicated by f_{ac} and f_{ab} with the same routing path $e4 \to e2 \to e1$. By default, two data flows equally share the bandwidth of link $< e2, e4 >$. The throughput of the job using this strategy is 2.5, which is better than strategy in Fig. 3.2(c) as the raw data transmission in (c) becomes the bottleneck. Further, Fig. 3.2(e) improves (d) with the throughput 3.3 due to the optimized bandwidth sharing policy, where the bandwidths allocated to flow f_{ac} and f_{ab} are proportional to the amount of dependent data. Fig. 3.2(f) shows a throughput of 4 with customized routing policy, where the flow f_{ac} selects the routing path $e4 \to e2 \to e1$ with the allocated bandwidth 10 and the flow f_{ab} selects the path $e4 \to e3 \to e1$ with the allocated bandwidth 6.

From the above examples, we can see that joint consideration of the coupled resources

by optimizing the task allocation strategies, the bandwidth allocation, and flow routing policies can improve the application performance. In the rest of this section, we build ENTS system to orchestrate coupled edge resources and design optimal collaborative task scheduling algorithms by jointly considering the data, computing, and networking resources of the geo-distributed edge nodes.

3.3 System Overview

This section gives an overview of the design goals and the system components. ENTS is designed based on Kubernetes to manage the resources and schedule the workloads over the geo-distributed, large-scale, and heterogeneous edge environment. It has two main objectives: 1) Jointly manage and orchestrate the coupled and distributed data, computation, and networking resources; 2) Enable effective distributed task execution to achieve better performance of applications.

3.3.1 Design Goals

The design of ENTS obeys the principles as follows.

- *Scalability.* The system can be scaled to a large number of devices and services retaining its high performance.
- *Collaboration.* The different edge nodes can collaborate to manage the distributed and heterogeneous resource regarding data, computation, and networking.
- *Universality.* The system supports execution of various kinds of tasks and workloads.

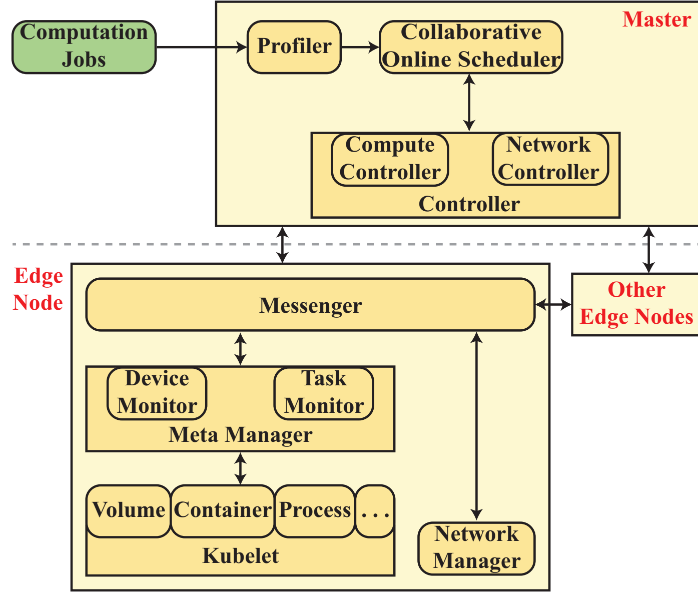


Figure 3.3: Architecture of the ENTS System

3.3.2 System Architecture

In Fig. 5.1, we show a birds-eye view of ENTS's system architecture and functional workflow. The system adopts the server-client architecture and is built based on Kubernetes with a master node to manage the distributed resources and schedule the tasks among edge nodes. Kubernetes components are used to manage the computation and storage resources of edge nodes. However, Kubernetes lacks support to profile the job's inner-dependency and execution time on heterogeneous edge nodes and orchestrate networking resources. Hence, we develop new components to enhance the ability of Kubernetes to orchestrate coupled resources considering the job profile. The system follows the principles of service-oriented architecture, where functions of the components are developed as services and can be called with APIs.

The components of the system are listed below.

- *Profiler* parses the input job and profiles the execution time of tasks on heterogeneous edge nodes. The job profile will be used to support intelligent task scheduling.

- *Scheduler* accesses the system information, such as CPU and GPU usage, network conditions, and job profile. On this basis, it generates the policies of task execution and resource allocation that optimizes job performance.
- *Compute controller* manages the computation and storage resources at the edge nodes. It leverages the Kubernetes components API server and controller manager to orchestrate the computation resources.
- *Network controller and manager* manage the networking resources of edge nodes, such as bandwidth allocation, routing and forwarding of data flows.
- *Messenger* handles the message between the edge node and the master. We extend the messaging of Kubernetes between the master and clients because it lacks support for orchestrating network resources.
- *Kubelet* manages pods, containers, and data volumes. It is Kubernetes original component, whose primary responsibility is for task execution.
- *MetaManager* is responsible for monitoring and storing device status and application status. Specifically, the device and task monitors are responsible for storing and retrieving metadata (device status and task execution status) to and from a lightweight database. Such information will be sent to the master node for supporting task scheduling.

ENTS is based on Kubernetes and reuses the key components of Kubernetes. It equips Kubernetes with the ability to jointly orchestrate the networking and computation resources to optimize the performance of edge-native applications. The general workflow of the system is described as follows. The profiler first parses the user-submitted job and profiles the execution time of each task of the job on heterogeneous edge nodes. The job profile information, including the inter-dependencies of tasks and task execution time, will be used for later decision-making of task scheduling. The scheduler generates the task execution policies by jointly considering the job profile

information, the data locality, available computation and networking resources of the edge nodes. Specifically, the policies decide which node to allocate tasks, the bandwidth allocation and the routing path of dataflows. The policies will be managed by the network controller and the compute controller together, and then be executed by the Kubelet and the network manager on the client nodes. The run-time characteristics of tasks and the nodes' status will be sent back to the controller in the master and used for later task scheduling.

3.4 System Design

This section illustrates the details of the ENTS system workflow, including job profiling, collaborative task scheduling, and distributed task execution.

3.4.1 Application Development and Profiling

To easily parse the user-submitted job and facilitate efficient distributed task execution, we adopt the data flow programming model [48], where each task in a job is programmed as a function module. Tasks are loosely coupled with intermediate data transmission. Note that many modern applications are modeled in such a way. Those applications are complex in nature, structured on microservices architecture style, consisting of a large number of inter-dependent and loosely coupled modules. Besides, to support various kinds of workloads, the programming model is non-intrusive to the user programming language. As shown in Fig. 3.4, we only require developers to declare the tasks in the submitted job without intruding on the main functions of the applications. Users can use any programming language to implement their applications. Compared with those programming models, which require users to learn lots of pre-defined operations, such as Hadoop, Spark, and Flink, ENTS is easier to learn and use.

```
1  # Start definition of tasks
2  def task_0(input):
3    # User code here...
4    return True, output
5
6  def task_1(input):
7    # User code here...
8    return True, output
9
10 def task_2(input):
11   # User code here...
12   return True, output
13
14 def task_3(input):
15   # User code here...
16   return True, output
17
18 # Export functions as job
19 job = [task_0, task_1, task_2, task_3]
```

Figure 3.4: Code Snippet of User Application

Users are required to submit the job configuration so that the system can profile the job and perform efficient task scheduling. As shown in Fig. 3.5, the configuration explicitly defines the data source, dependencies among the tasks, and the resource demand of each task. Particularly, the job consists of 4 tasks. The first task $task0$ demands 2GB memory and has subsequent tasks $task1$ and $task2$. After the user submits the job configuration, ENTS will start the profiling. The objective of job profiling is to estimate the running time of each task of the submitted job on heterogeneous edge nodes, which will then be used to support the collaborative task scheduling. Since it may take much time to profile the job, depending on the complexity of the job, we do the profiling offline. Specifically, the profiler will send the job configuration to the edge nodes that meet the resource requirements of the job. Each edge node will profile the job by executing the tasks under the requested resource and send the job profile information back to the scheduler. Offline profiling is reasonable for those long-running jobs, such as video analytics [128] and virtual reality [130]. Other methods can be used to measure the computing capability of edge nodes and

estimate the workload of the application in advance, which is more suitable for online application profiling [55] [80]. We will study them in the future and incorporate the mechanisms into ENTS.

```
1  {
2    job: "test",
3    image: "userid/ents:ubuntu",
4    total_memory_request: "4GB",
5    source: "",
6    input_size: "10"
7  },
8  {
9    id: "task0",
10   downstream: "['task1','task2']",
11   memory_resource: "2GB"
12 },
13 {
14   id: "task1",
15   downstream: "['task3']",
16   memory_resource: "2GB"
17 },
18 {
19   id: "task2",
20   downstream: "['task3']",
21   memory_resource: "2GB"
22 },
23 {
24   id: "task3",
25   downstream: "",
26   memory_resource: "2GB"
27 }
```

Figure 3.5: Code Snippet of Application Configuration

3.4.2 Collaborative Task Scheduling

After a job is profiled, it will be added to a Job Queue and pending to be scheduled, as shown in Fig. 3.6. The job-related information, including task dependencies and requested resources, the available computation resource of edge nodes, and the status of the network will be sent to the scheduler to support the collaborative task scheduling decisions. We will elaborate on the scheduling algorithms in Sec. 3.5.

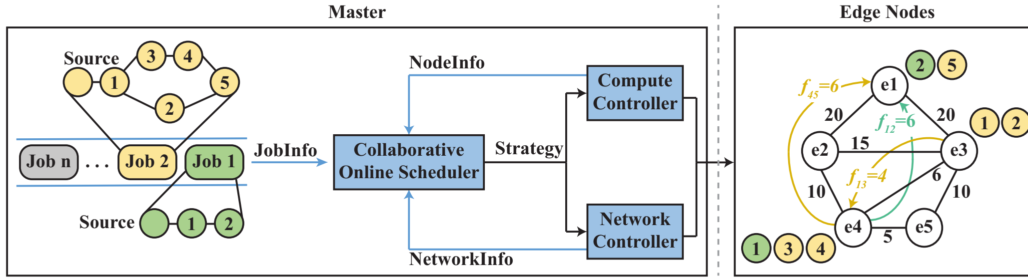


Figure 3.6: ENTS Task Scheduling Workflow

The scheduler generates the collaborative task scheduling strategy, which decides where to allocate each task, how much the allocated bandwidth is, and the routing path together with the communication port for each data flow. As shown in Fig. 3.7, the job shown in Fig. 3.5 is partitioned into 3 tasks, where task0 and task1 are allocated to edge nodes $e1$ and $e2$, respectively. Task2 and task3 are both allocated to $e3$. The bandwidth of data flow f_{01} and f_{02} is restricted to 15Mbps and 10Mbps, respectively. The source node port and destination port of flow f_{01} are set to be 8089 and 8090, respectively. The routing path of flow f_{13} is determined as $\{e2, e3, e4\}$.

Once the task scheduling strategy has been determined, they will be maintained by the compute controller and network controller, respectively, and sent to the edge nodes for execution. Specifically, the computation resource-related strategies, such as where to allocate the task and how many resources are assigned to the task, will be managed by Computer Controller, which interacts with the Kubelet on edge nodes to ensure the start, status monitoring, and stop of the containerized task. The networking resource-related strategies, such as port, bandwidth, and routing path of data flow, are managed by the network controller, which interacts with the network manager on edge nodes to ensure the communication and data transmission among edge nodes. The two controllers jointly manage the edge resources and ensure the correct execution of the collaborative task scheduling strategies with the coordination of the scheduler.

```
1  {
2    job_name: "test",
3    tasks: {
4      task_1: {
5        task_id: "0",
6        source_node: "edge_1",
7        source_node_port: "8089",
8        previous_node: "",
9        next_node: "edge_2 edge_3",
10       next_node_ports: "8090 8091",
11       bandwidth: "15Mbps 10Mbps",
12       routing: ""},
13     task_2: {
14       task_id: "1",
15       source_node: "edge_2",
16       source_node_port: "8090",
17       previous_node: "edge_1",
18       next_node: "edge_3",
19       next_node_ports: "8092",
20       bandwidth: "10Mbps",
21       routing: "edge_2 edge_4 edge_3"},
22     task_3: {
23       task_id: "2 3",
24       source_node: "edge_3",
25       source_node_port: "8091",
26       previous_node: "edge_1 edge_2",
27       next_node: "",
28       next_node_ports: "",
29       bandwidth: "",
30       routing: ""},
31   }
32 }
```

Figure 3.7: Collaborative Task Scheduling Strategy

3.4.3 Distributed Task Execution

When the messenger receives the task execution policy, it will decompose the policies into computation-related and networking-related policies. The computation-related policies will be forwarded to and maintained by the Kubelet, while the networking-related ones will be forwarded to and maintained by the network manager. Kubeetl and network manager work together to ensure the proper execution of the assigned task.

One important role of the network manager is to manage and orchestrate the networking resources. In this work, we are mainly concerned with the bandwidth allocation and customized routing of the cross-node data flows. For cross-node communication, Kubernetes usually adopts a flannel network [20]. As shown in Fig. 3.8, a data

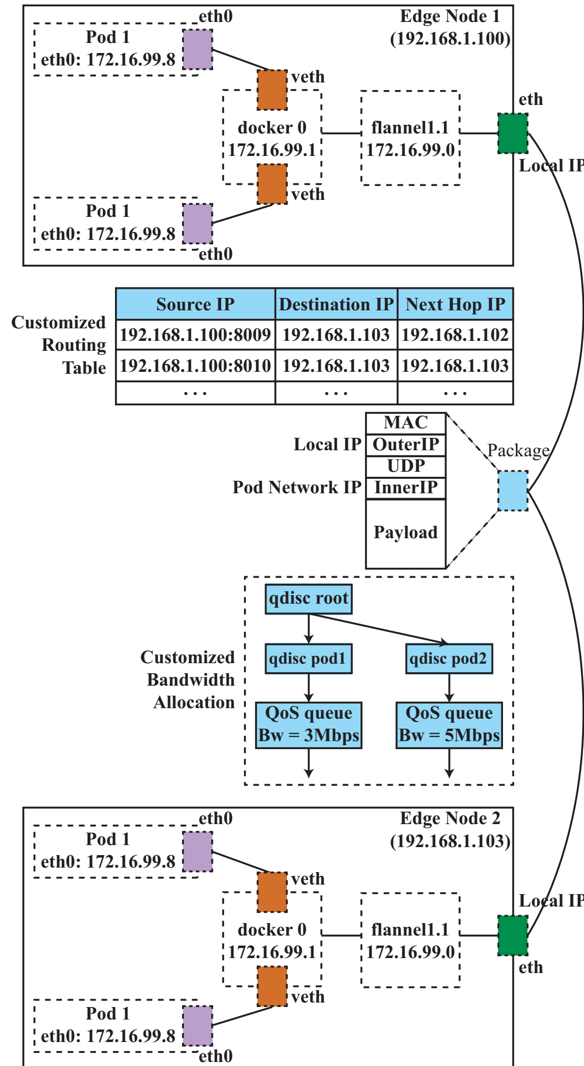


Figure 3.8: Bandwidth Allocation and Customized Routing of Network Manager

package from Pod1 to Pod3 will first be forward to docker0 and then to the flannel interface. The package will go through eth on edge node A and be sent to edge node B, where a reverse process will be performed to analyze the Internal IP of the package and route the package to the destination, i.e., Pod 3. To achieve the bandwidth allocation and customized routing of data flow, for each data flow in a scheduled job, the network manager will specify the *{source_ip, source_ip_port, bandwidth_limit, destination_ip, destination_ip_port}*, as shown in Fig. 3.7. Through this information, the network manager leverages the Linux kernel functions, i.e., Traffic Control and

Iproute [40], to shape the bandwidth between two edge nodes and customize routing for data packages. Traffic control creates Classful Queuing Disciplines (qdisc) to filter and redirect network packages to a particular quality-of-service queue before sending them out. The network manager also maintains the routing table of each assigned task. As shown in Fig. 3.8, the data package going through port 8009 from edge node 1 will be forwarded to another edge node rather than go directly to the destination, i.e., edge node 2. Also, the bandwidth of data flow from Pod1 of edge node 1 will be shaped to 3Mbps.

After the network configuration takes effect, the kublet will launch the pod according to the assigned computation-related policies, such as CPU and memory requests. The device monitor and task monitor will consistently and continuously monitor the status of the devices and the task.

3.5 Collaborative Task Scheduling with Data Streaming Applications

In this section, we showcase the collaborative task scheduling of ENTS with representative data streaming applications, namely edge video analytics. We first introduce the system model. Then, we formulate a joint task allocation and flow scheduling problem for a single job scheduling and illustrate the proposed algorithms. On this basis, we further propose two online scheduling algorithms to schedule multiple continuous arriving jobs to maximize the average job throughput.

3.5.1 System Model.

Edge video analytics [128] [119] [127] is a killer application of edge computing. The network and application model used in formulating the problem is described as follows.

Network Model. The communication network is a mesh network of edge nodes connected using a multi-hop path. The network is modelled as an undirected graph $G = (V, E)$, where V is the set of edge nodes, $V = \{j|1 \leq j \leq M\}$, and E is the set of links connecting different edge nodes, $E = \{l_{u,v}|u, v \in V\}$. Here, M is the total number of edge nodes. The computing capacity, maximum resource and available resource of edge node j is PS_j, R^j_{max} and R^j_{avail}, respectively. The bandwidth of link l is represented by B_l. The network can be heterogeneous in terms of the computation capacity of edge nodes and link bandwidth.

Application Model There will be multiple jobs submitted to the ENTS system by the edge nodes. Each job is modeled as a directed acyclic graph $J = (T, P)$, where T is a set of dependent tasks and P represents the set of dependencies between the tasks in the job. Pd_i denotes the predecessor tasks of task T_i. The computation workload and resource demand of task j is C_j and R^j_{req}. The amount of dependent data between task j and task i is $D_{i,j}$. The input data source of job J is assumed to be located at an edge node $s_J|s_J \in V$.

3.5.2 Problem Formulation

The objective of the single job scheduling is to maximize the throughput of the job by deciding where to allocate each task of the job, the routing path and bandwidth allocation of each data flow caused by the intermediate data transmission. If two dependent tasks are allocated to the same edge node, there will be no intermediate data transmission and thus no data flow. The joint task allocation and flow scheduling problem denoted as P_1 is formulated as follows.

$$\max \left\{ TP = \frac{1}{t_p} \right\} \tag{3.1}$$

$$t_p = \max \left\{ \max_{i \in T}(t^i_{comp}), \max_{i \in T, j \in Pd_i}(t^{i,j}_{comm}) \right\} \tag{3.2}$$

$$t_{comp}^{i} = X_i^u \cdot \frac{C_i}{PS_u} \tag{3.3}$$

$$t_{comm}^{i,j} = X_i^u \cdot X_j^v \cdot \frac{D_{i,j}}{B_{l_{u,v}}}, j \in Pd_i \tag{3.4}$$

$$X_i^u \in \{0, 1\}, \forall i, u \tag{3.5}$$

Eq. 3.3 indicates the computation time of task i, where X_i^u is a binary variable. X_i^u equals to 1 if task i is allocated to edge node u, otherwise X_i^u equals to 0. Eq. 3.4 shows the transmission time of the intermediate data between dependent task i and j. The throughput is $TP = \frac{1}{t_p}$, where t_p is constraint by the maximum transmission and computation time as indicated by Eq. 3.2. P_1 is a mixed Integrated Non-linear problem (MINLP), which is proven to be NP-hard in literature.

3.5.3 Proposed Solution

To solve the problem P_1, we decompose it into two sub-problems, i.e., allocate each task of the job P_2 and decide the routing path and bandwidth allocation of all the data flows P_3. To solve P_2, we use a greedy algorithm to allocate each task to the edge node, which can provide the least execution time, including the computation time and the dependent data transmission time. To solve P_3, we first relax it into a convex problem, which can be solved by convex optimizers, and then derive the solution for P_3.

Solving Problem P_2. The algorithm to solve P_2 is shown in Algo. 1. For each task in the job, the algorithm traverses all the edge nodes with satisfied resource capacity and allocates the task to the edge node with the minimum execution time, including both computation time and intermediate data transmission time (Line 3-13). For calculating $t_{comm}^{i,j}$, we set the bandwidth between two edge nodes as the average bandwidth of all routing links. This is reasonable because the intermediate data flow can have multiple choices to avoid network congestion. Later, we will adjust

the allocated bandwidth and the routing path of the data flows in a more fine-grained way in problem P_3.

Algorithm 1: Task Allocation

Input: network $G = (V, E)$, job $J = (T, P)$,

Output: the task allocation policy $T_{i,j}$, the data flows FL

1 Initialize $T_{i,j} \leftarrow 0$ for all i, j;

2 Query the available resource R_j of all edge nodes;

3 **for** *task* T_i *in job* $J = (T, P)$ **do**

4   **for** *edge node* j *in network* $G = (V, E)$ **do**

5   **if** $R^j_{avail} > R^i_{req}$ **then**

6   Calculate the computation time $t^i_{comp} = C_i \div PS_j$;

7   Calculate the intermediate data transmission time $t^i_{comm} = \max t^{i,j}_{comm}$ using Eq. (4);

8   Calculate the execution time $t^j_{exec} = t^j_{comp} + t^j_{comm}$;

9   **end**

10   **end**

11   Allocate task T_i to node $j^* = min_J\{t^j_{exec}\}$;

12     $T_{i,j^*} \leftarrow 1$;

13   Update R_{j*} for node j^*;

14 **end**

15 Calculate data flow $f_i =< source, destination, datasize >$ with $T_{i,j}$;

16 Add f_i to data flows FL;

17 **return** $T_{i,j}$, FL

Solving Problem P_3. After solving P_2, we get the data flows FL, where we can know the number of data flows Nf, the source, destination, and data volume of each data flow f_i. We then solve P_3 to decide the routing path and bandwidth allocation of each data flow. The P_3 is formulated as follows.

$$\min \max_{i=1,\ldots,Nf} \left\{ \frac{V_i}{b_i} \right\} \tag{3.6}$$

$$\sum_i \sum_{k:l \in P_i^k} b_i y_i^k \leq B_l, \forall l \tag{3.7}$$

$$\sum_k y_i^k = 1, \quad \forall i \tag{3.8}$$

$$y_i^k \in \{0, 1\}, \forall i, k \tag{3.9}$$

where V_i is the size of flow f_i and b_i is the bandwidth allocated to flow f_i. P_i^k is the collection of all the possible routing paths of flow f_i. y_i^k is a binary variable. y_i^k equals to 1 if flow f_i chooses the kth routing path of P_i^k. Note that Eq. 3.7 indicates that the sum of allocated bandwidth of all data flows going through link l cannot exceed its capacity. Eq. 3.8 and Eq. 3.9 ensure that a data flow can only choose one routing path.

The problem P_3 is still a MINLP problem. Therefore, we resort to relaxing the integer variable y_i^k to a real variable $y_i^k \geq 0$. We name the relaxed problem $P_3 - \text{Relax}$. Due to the existence of term $b_i \cdot y_i^k$, the $P_3 - \text{Relax}$ problem is still a non-linear programming problem which is hard to solve. In the following, we transform the $P_3 - \text{Relax}$ problem into an equivalent convex optimization problem.

An Equivalent Convex Problem. First, we introduce an variable TH such that $TH = \max_{i=1,\ldots,Nf} \left\{ \frac{V_i}{b_i} \right\}$. Furthermore, we introduce another variable q_i such that $q_i = TH \cdot b_i$, and variable $m_i^k = q_i \cdot y_i^k$. Then, the equivalent problem $P_3 - \text{Relax-Cvx}$ is formulated below.

$$\min TH \tag{3.10}$$

$$\sum_i \sum_{k:l \in P_i^k} m_i^k \leq B_l \cdot TH, \forall l \tag{3.11}$$

$$\sum_k m_i^k = q_i, \quad \forall i \tag{3.12}$$

$$m_i^k \geq 0, \forall i, k \tag{3.13}$$

$$q_i \geq V_i, \forall i \tag{3.14}$$

All constraint in the $P_3 - \textsc{Relax-Cvx}$ is affine, and the objective function is convex. Therefore, the $P_3 - \textsc{Relax-Cvx}$ problem is a convex optimization problem which can be solved using convex optimizers [8].

However, since we relax the binary integer constraint, the solution may be that some y_i^k are decimal factions. To solve the problem, we route the ith data flow to a path k^* such that $m_i^{k^*} = \max_k m_i^k$. When the routing path is determined, the optimal bandwidth allocation policies is given by

$$b_i^* = \min \left\{ \frac{V_i}{\sum_i \sum_{k:l \in P_i^{k^*}} V_i y_i^{k^*}} \right\}, l \in P_i^{k^*} \tag{3.15}$$

The algorithm to solve P_3 is shown in Algo. 2.

3.5.4 Online Scheduling

Algo. 1 and Algo. 2 study the task scheduling for one job. However, in a practical ENTS system, jobs constantly arrive and share the resource in the network. Our goal is to maximize the average job throughput. Motivated by this, we propose two online scheduling algorithms, which run in the ENTS online scheduler and periodically schedule all arrived jobs.

The online scheduler maintains two job queues: 1) a queue of jobs that are running, denoted by Q_{run}, and 2) a queue of jobs that are waiting to be scheduled, denoted by Q_{wait}. The two online scheduling algorithms are: 1) schedule the job in Q_{wait} one by one, and 2) schedule the job in Q_{wait} one by one but readjust the routing and bandwidth sharing strategy by considering all the existing and coming data flows in the edge network.

Algorithm 2: Joint Routing and Bandwidth Allocation (JRBA)

Input: network $G = (V, E)$, data flows FL,

Output: the routing policy y_i^k, the bandwidth allocation policy b_i, and job throughput JTH

1 Solve $P_3 -$ Relax-Cvx and get $\{T^*, q_i^*, m_i^{k^*}\}$;
2 **for** *flow* f_i *in* FL **do**
3   Initialize $y_i^k \leftarrow 0$ for all k;
4     $k^* \leftarrow arg_k \max m_i^k$;
5     $y_i^{k^*} \leftarrow 1$;
6 **end**
7 Calculate b_i^* using Eq. 3.15;
8 Update B_l according to $y_i^{k^*}$, b_i^*;
9 $JTH \leftarrow \max_{i=1,\ldots,N} \left\{ \frac{V_i}{b_i} \right\}$;
10 **return** y_i^k, b_i, JTH

The first algorithm (OTFS) is shown in Algo. 3. For each job in the queue Q_{wait}, the algorithm first sorts the job in descending order of waiting time and schedules the jobs in sequence (Line 6-9). During scheduling, the algorithm calls the procedure Task Allocation (Algo. 1) and JRBA (Algo. 2) in turn (Line 9-13).

The second algorithm (OTFA) is shown in Algo. 4. Different from OTFS, which makes task scheduling decisions based on the current status of the computation and networking resource in the edge network, OTFA jointly manages the existing data flows and the coming data flows. It first allocates the computation resources for arriving jobs and then readjusts the networking resources for all data flows (Line 10-15).

Algorithm 3: OTFS: Online Task Allocation and Flow Scheduling

Input: current time $curT$, network $G = (V, E)$, Q_{wait}

1 $J_{finish} \leftarrow$ all jobs finishing at $curT$;
2 **if** $J_{finish} \neq \emptyset$ **then**
3   Release all computing resource and bandwidth allocated to J_{finish};
4   Update R_j and B_l for network;
5 **end**
6 **if** *there are jobs arriving at curT* **then**
7   Add jobs arriving at $curT$ to Q_{wait};
8 **end**
9 Sort Q_{wait} in descending order of waiting time;
10 **for** *job J_i in Q_{wait}* **do**
11   Call the Task Allocation procedure to get $\{T_{i,j}, FL\}$;
12   Call the JRBA procedure;
13 **end**

Algorithm 4: OTFA: Online Scheduling Task Allocation Joint Flow Adjustment

Input: current time $curT$, network $G = (V, E)$, Q_{wait}, Q_{run}

1 $J_{finish} \leftarrow$ all jobs finishing at $curT$;
2 **if** $J_{finish} \neq \emptyset$ **then**
3 Release all computing resource and bandwidth allocated to J_{finish};
4 Update R_j and B_l for network;
5 **end**
6 **if** *there are jobs arriving at curT* **then**
7 Add jobs arriving at $curT$ to Q_{wait};
8 **end**
9 Sort Q_{wait} in descending order of waiting time;
10 **for** *job J_i in Q_{wait}* **do**
11 Call the Task Allocation procedure to get $\{T_{i,j}, FL\}$;
12 **end**
13 Release all bandwidth allocated to data flows FL_{run} in Q_{run};
14 Add FL to FL_{run};
15 Call the procedure JRBA with FL_{run};

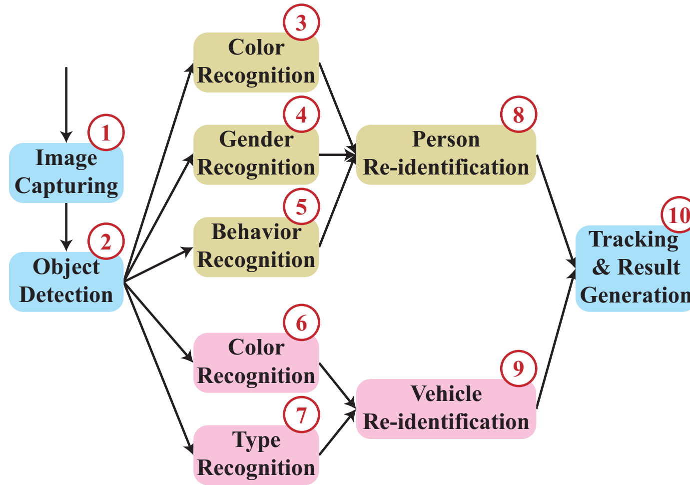


Figure 3.9: Application Graph of Object Attributes Recognition

3.6 Experimental Results

3.6.1 Experimental Setup

Benchmarks. To evaluate the ENTS system, we use a real-world live video analytics application, i.e., object attribute recognition [120], which is extensively used in surveillance of public safety. The application graph is shown in Fig. 3.9, where we have 10 functional modules. For modules 2 to 9, each of them is implemented with a computing-extensive and resource-greedy DNN model [43] [117]. The application takes the surveillance video as input and recognizes the attributes of pedestrians and vehicles in the video, such as the color of cloth, gender of pedestrians, and type of vehicles. Specifically, we use MobileNet-V2 [92] as the backbone network for object detection in module 2. For attribute recognition and object re-identification, i.e., module $3-9$, we use Resnet-50 [34] as the backbone network. We use the Kalman filter to track the objects in module 10. The resolution of the video is 1920x1080 with 30fps and the size of each video frame is about 6MB. The application is implemented with Python.

Baselines. We compared the proposed method with three state-of-the-art baselines

as follows.

- *LeastRequestPriority (LR).* It schedules the whole job to the edge node with the least resource consumption. The LR policy is frequently used in Kubernetes.
- *BalancedResourceAllocation (BR).* It schedules the whole job to the edge node, which can balance the resource consumption among the edge nodes. BR is used in Kubernetes to achieve workload balancing.
- *Task Partition (TP).* It partitions the job and schedules each task to the edge nodes with the least execution time, including the transmission time and the computation time. We adopt the default shortest path to transfer the intermediate data. When multiple data flows go through the same link, all flows equally share the link bandwidth.

Metrics. We employ two metrics as follows.

- *Average Job Throughput.* It is the average throughput of all submitted jobs. It is an important metric to measure the performance of the scheduling algorithms.
- *Average Waiting Time.* It is the average waiting time of all submitted jobs, i.e., the time from the job submitted to the job scheduled. It is a metric reflecting the effectiveness of the scheduler and system overhead.

Testbed Implementation. To test the system on a large scale geo-distributed edge environment, we developed a hybrid testbed with both physical and virtual edge nodes, as shown in Fig. 4.7. We use virtual machines to emulate virtual edge nodes. While numerous virtual edge nodes enable us to test in a large-scale and network-flexible testing environment, the incorporation of physical nodes guarantees the fidelity of the testbed. We leverage Linux Traffic Control to configure the network topology and bandwidth among the edge nodes. We vary the network link bandwidth, e.g., from $1Mbps$ to $10Mbps$, to emulate the physical distance among edge nodes. The

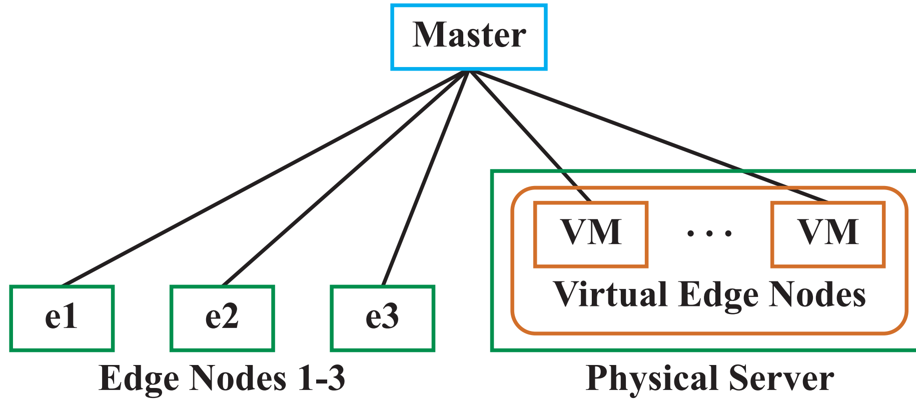


Figure 3.10: Test Environment of ENTS

intuition is that the bandwidth should be low if two nodes are far away. Similar idea is also adopted in [54].

Specifically, we randomly generate the network connection among edge nodes with the average node degree as 3. We also enable routing and forwarding on each node so that each node is both a compute node and a router. We use 4 raspberry pi, 2 Nvidia Jetson Nano, and 2 Nvidia Jetson Xavier NX to represent physical edge nodes. A PC equipped with four Intel Cores i9-7100U with 20GB RAM to act as the master node to manage the edge nodes. Two servers are leveraged to host virtual machines acting as virtual edge nodes. One is equipped with Intel(R) Xeon(R) Gold 6128 CPU with 192GB Memory, another is Intel(R) Core(TM) i9-10900F CPU with 64GB memory. The specifications of the physical devices are shown in TAB. 4.2.

3.6.2 Results and Analysis

We test the performance of the ENTS system and the proposed online scheduling algorithms under various situations.

Effects of Number of Edge Nodes. We evaluate the influence of the number of edge nodes on the average job throughput and average waiting time to test the scalability of ENTS. In this experiment, a total of 50 jobs are submitted by the

Table 3.1: Specifications of Physical Devices

Name	CPU	Memory	Performance
Raspberry Pi	1 core	1GB	Low
Jetson Nano	6 cores	4GB	Low
Jetson Xavier NX	6 cores	8GB	Medium
Edge Server-1	64 cores	64GB	High
Edge Server-2	128 cores	192GB	High

edge nodes to the master with the arriving rate following a Poisson distribution with $\lambda = 0.5/second$.

As shown in Fig. 3.11(a), TR, OTFS, and OTFA perform much better than LR and BR, with higher average throughput. The average throughput of LR and BR does not exceed 1. It is because LR and BR do not partition the job, which leads to the transmission of source video data over a low-bandwidth edge network. It becomes the bottleneck of the job throughput. Unlike LR and BR, the other three methods, i.e., TP, OTFS, and OTFA, partition the job and enable distributed job execution, avoiding raw data transmission. OTFA performs best with the highest throughput among TP, OTFS, and OTFA. TP shares the bandwidth equally and assigns the shortest routing path for network flows, which usually leads to traffic congestion when multiple data flows pass through the same network link. Instead, OTFS and OTFA optimize the networking resources by enabling optimal bandwidth sharing and routing path selection concerning the end-to-end job throughput. OTFA goes further. It considers all the available data flows in the network, which can improve the average job throughput compared to OTFS.

We also observe that the average throughput does not show a linear growth with an increasing number of edge nodes. Generally, when the number of edge nodes increases, the network will have more resources and higher job throughput. However,

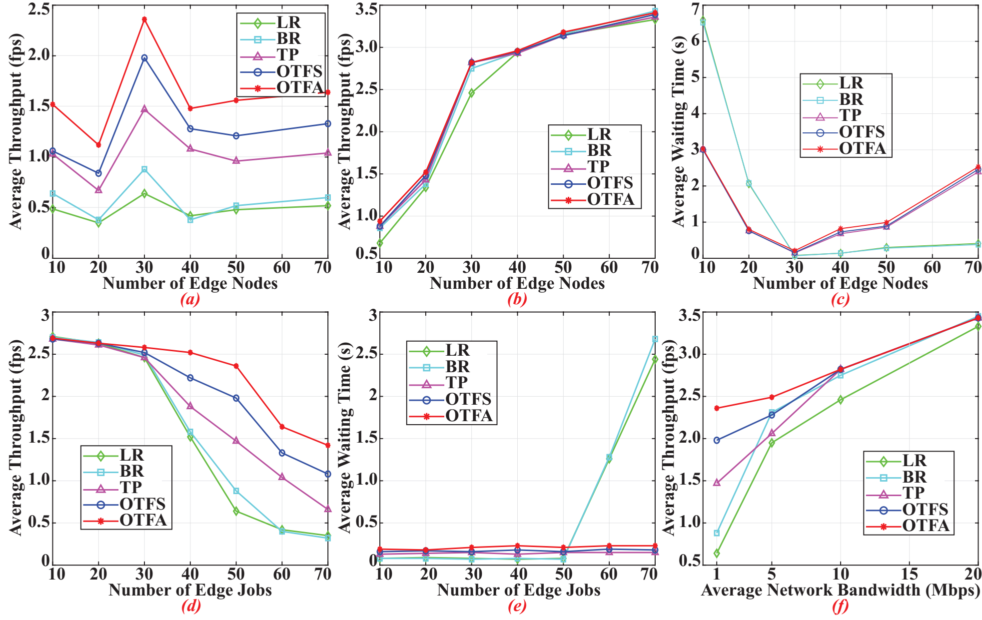


Figure 3.11: a) Impact of the number of edge nodes on average throughput with average bandwidth 1Mbps. b) Impact of the number of edge nodes on average throughput with average bandwidth 10Mbps. c) Impact of the number of edge nodes on average waiting time. d) Impact of the number of submitted jobs on average throughput. e) Impact of the number of submitted jobs on average waiting time. f) Impact of average bandwidth on average throughput.

the throughput decreases slightly when the number of edge nodes increases from 10 to 20 and 30 to 40. It is because of the limited network bandwidth, i.e., 1Mbps with a variance of 0.3 in our experiment. When the number of edge nodes increases, the number of hops and network links between two edge nodes also increases, resulting in more bottleneck communication paths. As shown in Fig. 3.11(b), when the average bandwidth of the edge network becomes 10Mbps, such fluctuation of the average throughput will no longer exist. More specifically, it shows a linear growth as expected. It is because the network bandwidth is not the bottleneck anymore, and there are fewer bottleneck communication paths.

Fig. 3.11(c) depicts the influence of the number of edge nodes on the waiting time. When the number of the edge nodes is below 30, the average waiting time of TP, OTFS, and OTFA is much smaller than that of LR and BR. The reason is that the

former scheduling policies partition the job and allocate the task into edge nodes with less abundant resources, improving resource utilization and the number of jobs executable among the geo-distributed edge nodes. When the number of edge nodes is above 30, the total resource is sufficient, where the average waiting time is dominated by the running efficiency of the scheduling algorithms. Compared with the LR and BR algorithms, TP, OTFS, and OTFA are required to traverse all the edge nodes for each task and solve the formulated optimization problem, which increases the average waiting time. However, we observe that when the number of edge nodes is below 50, the average waiting time is no more than 1 second, and about 2.5 second when the number of edge nodes is 70, which is still at a low level.

Effects of Number of Submitted Jobs. We evaluate the performance of the average job throughput and wait time with a changing number of submitted jobs. We set the average bandwidth as 1Mbps with a variance of 0.3. The number of edge nodes is 30. The arriving rate of the submitted jobs follows a Poisson distributed with $\lambda = 0.5/second$.

As shown in Fig. 3.11(d), when the number of submitted jobs is no more than 30, our method performs similarly to the baseline. In such cases, the edge resources are relatively abundant, and the proposed methods, i.e., OTFS and OTFA, tend to yield similar decisions compared with the baseline methods. However, when there are more jobs, the average throughput of LR and BR declines dramatically. It is because multiple jobs compete for limited networking and computation resources. Without partitioning the submitted jobs and optimizing the bandwidth allocation and routing path of flows, LR and BR easily suffer from network congestion and fragmented computation resource usage, degrading the average job throughput significantly. OTFA performs the best. Compared to TR and OTFS, OTFA considers optimal bandwidth sharing and routing path for incoming in addition to existing data flows, which can further improve the averaging job throughput with better resource utilization when there are more jobs.

Fig. 3.11(e) depicts similar trends concerning the performance in average waiting time. When the number of submitted jobs is below 50, the average waiting time for all the mentioned methods is low, i.e., no larger than 0.5 without apparent fluctuation. We can also see that the waiting time of LR and BR is shorter than that of TP, OTFS, and OTFA. It is because the latter three approaches have to traverse all the edge nodes for each task, which leads to more waiting time for scheduling jobs. When the number of submitted jobs exceeds 50, TP, OTFS, and OTFA show consistent average waiting times while the performance of LR and BR increases significantly. The reason is that there are no available resources to schedule the new-coming jobs. The rest of the jobs are required to wait in the job queue, which results in an increased average waiting time. Compared to TR, OTFS, and OTFA, the other two methods, i.e., BR and LR, do not partition the submitted job, which may easily lead to fragmented resource consumption and thus serve fewer jobs.

Effects of Average Bandwidth. We also evaluate the performance of the average job throughput with the variance of the average bandwidth of the edge network. We set the number of edge nodes as 30 in this experiment and the number of submitted jobs as 50 with the arriving rate following a Poisson distributed with $\lambda = 0.5/second$.

As shown in Fig. 3.11(f), the average throughput of all the methods increases with the average bandwidth. More specifically, when the average bandwidth of the edge network is no more than 5Mbps, OTFA outperforms other methods significantly because it jointly considers and optimizes the data locality, the networking, and computing resources of edge nodes. However, when the average bandwidth is above 10Mbps, baselines and proposed methods tend to have similar performance. It is because the bandwidth is relatively abundant now. However, OTFS and OTFA are slightly better than LR and BR, as they optimize the bandwidth allocation and routing selection for data flows in the edge network. BR outperforms LR as it aims to achieve balanced resource consumption, enabling the powerful edge nodes to service more jobs.

In a nutshell, we evaluated and compared the performance of ENTS with the state-

of-the-art and proposed online algorithms for scheduling streaming jobs. Benefiting from the ability to consider task dependencies and jointly optimize the limited coupled computation and networking resources, ENTS achieves a $43\% - 220\%$ improvement in average throughput. Although the proposed solutions introduce additional overhead in making the scheduling strategies, they can serve more jobs when resources of the edge network are limited, which leads to less averaging waiting time.

3.7 Conclusion

In this chapter, we designed and developed ENTS, the first edge-native task scheduling system, to manage geo-distributed and heterogeneous edge resources in collaborative edge computing. ENTS extends Kubernetes with the ability to jointly orchestrate computation and networking resources to optimize the application performance. ENTS comprehensively considers both the application characteristics and edge resource status. We show the superiority of ENTS with a case study on data streaming applications, in which we formulate a joint task allocation and flow scheduling problem and propose two online scheduling algorithms. Experiments on an object attribute recognition application on a large number of edge nodes show ENTS achieves improved performance.

Chapter 4

Scheduling Model Training Tasks

In this chapter, we design and develop an edge-native task scheduling system, namely EdgeSplit, for collaborative edge computing. This chapter is organized as follows. We present an overview of this work in Section 4.1. We further introduce the motivation of proposing EdgeSplit in Section 4.2. We then elaborate the training process of EdgeSplit and the methodology to reduce the model training time in Section 4.3. Section 4.5 shows the evaluation results. Finally, Section 4.6 concludes this chapter.

4.1 Overview

In recent years, due to the great advancement of deep learning, AI models and algorithms has been extensively used in various applications, including object detection, natural language processing, and autonomous driving [56]. Traditionally, AI models are trained on the cloud with the data collected from end devices due to its extensive resource consumption. However, the cloud-based solution suffers from high communication costs, long response time, and privacy concerns [97]. Recently, edge empowered AI, namely Edge AI, has been proposed to support the training and deployment of AI models on edge devices (e.g., edge servers, edge gateways, and mobile phones) at

the network edge closer to the data sources [11, 135].

A significant problem of Edge AI is to train accurate models with fast convergence using the distributed data on edge devices. Federated learning (FL) [61, 73] is a popular and promising solution, which enables collaborative model training without privacy violation. In FL, edge devices first use their local data to train the AI model. They then send the model updates, e.g., model weights or gradients, to the FL server for aggregation. The FL server will send back the aggregated parameters for the next round of local training. These steps are repeated multiple rounds until a desirable accuracy is achieved. FL has seen recent success in various applications, such as Google's Gboard [32], health AI [87], and across government collaboration [70].

However, fast training of federated learning on edge devices faces two challenges: 1) constrained edge resources. Edge devices are usually with limited memory and computation capabilities. Some edge devices, e.g., mobile phones, may not have the ability to burden the training tasks due to the complexity and great memory footprint requirements of large AI models; 2) heterogeneous edge resources. There are various edge devices, such as mobile phones, edge servers, and raspberry pis. The computing and networking capabilities of those devices are usually vastly different. The FL server is required to wait for the updated parameters of all the participants, which may lead to long waiting time.

The two challenges are not well addressed in existing works. Some works [28, 29, 64, 111] reduce the communication overhead for acceleration by compressing the transmitted data. However, they lead to accuracy loss. Other works try to adaptively select participants [12, 79, 110] and optimize model aggregation frequency [109, 116] among heterogeneous edge devices. But they neglect the constraint edge resources, where an edge device may not be able to perform the local training of a large AI model. Recently, split learning [81, 107] was proposed to enable the model training on low-resource mobile devices by splitting the full model between server and clients. However, it is a sequential training paradigm and not suitable for the parallel fed-

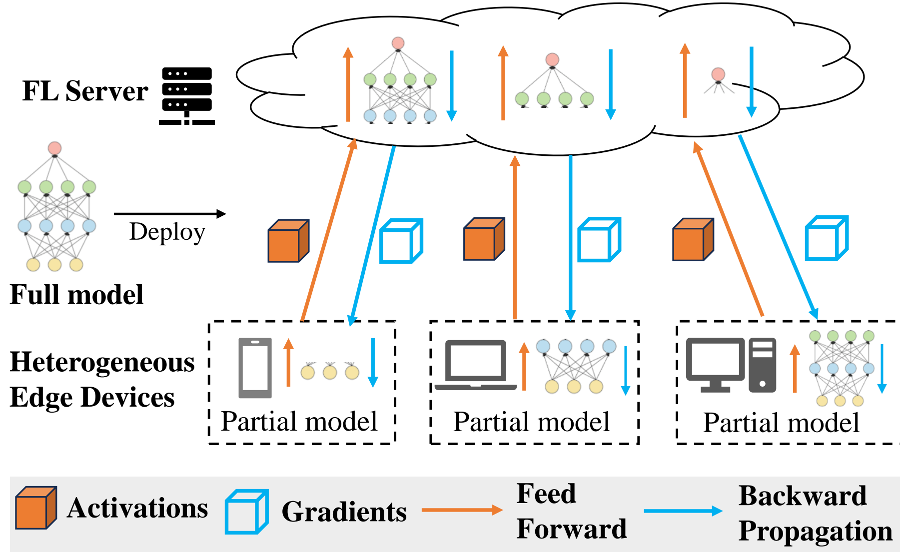


Figure 4.1: Overview of EdgeSplit. Edge devices train part of the full model with different depths adapting to local resources and offload the rest of model training task to the FL server for acceleration.

erated learning. Moreover, it does not take into account the resource heterogeneity among edge devices.

In this thesis, we rethink the problem from the perspective of training task scheduling. We propose a novel training framework, named EdgeSplit, to accelerate federated learning on heterogeneous and resource-constraint edge devices by enabling efficient task offloading among edge devices and the FL server. As shown in Fig. 4.1, a full model is partitioned into a set of shallow models with different depths at layer-granularity, adapting to the local edge resources. Each device trains part of the full model and partially offloads the rest of the training task to the FL server. Hence, the local training on edge devices is divided into four steps, i.e., local forward, server forward, server backward, and local backward. The server is responsible for both the rest of model training task and parameters aggregation. EdgeSplit provides an efficient and practical solution to train large models on low-resource edge devices and leverages the powerful computation resource of FL server to accelerate the distributed

model training in heterogeneous edge computing.

However, it is non-trivial to achieve efficient and practical EdgeSplit. First, it is difficult to decide the optimal model partition point for each edge device for minimizing the training time. The computation workload and size of parameters vary across the layers of a DNN model. Simple decoupling can hardly reduce the training time. To solve the challenge, we mathematically formulate a training task scheduling problem to decide the model partition points and the bandwidth between an edge device and the server, considering both devices' computing capabilities and network bandwidth. The problem is a mixed integer non-linear problem proven to be NP-hard. We hence propose an efficient alternating algorithm to quickly find the optimal solution by iteratively optimizing the model splitting and bandwidth allocation strategy. Second, handling the partial model training tasks offloaded from multiple edge devices leads to high memory consumption of the FL server. We thereby design an SplitPipe mechanism to alleviate the memory overhead by pipelining and reusing the duplicated layers of the partial models on the server, making it more suitable to the resource-constraint edge computing environment.

EdgeSplit is different from Split Learning. First, Split learning adopts fixed model partition points for all edge devices, neglecting resource heterogeneity. Second, Split learning does not address the memory overhead of the server. The efficacy of EdgeSplit is illustrated by developing a self-developed real-world testbed with both physical and emulated edge devices to evaluate the system in large scale. We have comprehensively evaluated the performance of EdgeSplit by comparing it with vanilla federated learning and baselines under various settings. The evaluation results show that our proposed EdgeSplit can achieve up to 5.5x acceleration. In a nutshell, our contributions are three folds:

- To the best of our knowledge, this is the first work that accelerates federated learning on heterogeneous and resource-constraint edge devices by adaptively

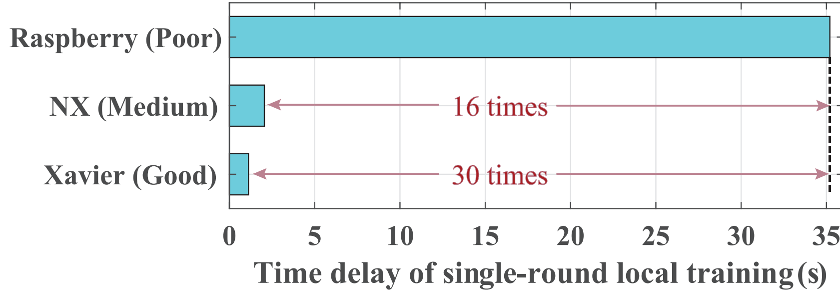


Figure 4.2: Federated learning on heterogeneous edge devices suffers from the straggler issue.

splitting and scheduling the training task. EdgeSplit significantly improves training efficiency without compromising accuracy.

- We propose novel solutions to address the challenging issues for achieving efficient and practical EdgeSplit, including joint model splitting and bandwidth allocation for fast training and SplitPipe mechanism to reduce memory overhead.

- We evaluate the performance of EdgeSplit on a real-world testbed with various benchmark models and datasets. The results indicate the superiority of EdgeSplit in accelerating federated learning.

4.2 Motivations

Federated learning suffers from the straggler issue on heterogeneous edge devices. There are two paradigms of federated learning. One is synchronous, and another is asynchronous. In the former paradigm, the FL server waits for the model updates from all participants and then performs the model aggregation. A large portion of time is used to wait for the weights updates from other less powerful edge devices, which is known as the straggler issue. It is usually caused by the heterogeneous computing and networking capabilities of edge devices. In asynchronous federated

learning, the server doesn't need to wait for the model updates from all participants. It performs server-side model aggregation once it receives an update. Though it has no straggler issue, the model convergence usually cannot be guaranteed [7]. Synchronous federated learning is still the mainstream training paradigm. Fig. 4.2 shows the time for one-round model training on three edge devices with different computing capabilities. We train a LeNet model on raspberry pi and two different NVIDIA Jetson family platforms (i.e., NX AGX, and Xavier AGX). As shown in Fig. 4.2, there is obvious heterogeneity among those devices. The single-epoch training on raspberry pi with poor computation capability consumes about 30 times training latency than that of Xavier AGX with good computation capability, and about 16 times training latency than that of NX AGX with medium computation capability.

Edge devices with limited memory footprint cannot perform local model training. Memory in neural network training is required to store input data, weight parameters, and activations as an input propagates through the network. In training, activations from a forward pass must be retained until they can be used to calculate the gradients in the backward pass. We observe that a 50-layer ResNet model requires 12GB memory if we use a 32-bit floating-point value to store model parameters. However, mobile phones are usually equipped with 8GB RAM. They can hardly hold the full ResNet model. Moreover, neural networks are getting larger and larger, such as transformers [62] and foundation models [6]. It is challenging to perform federated learning on edge devices to train large models with localized data. EdgeSplit is hence a promising way, which solves the resource constraint of edge devices by splitting the model and can utilize other heterogeneous resources to accelerate the FL training.

4.3 Framework of EdgeSplit

We first introduce the training framework and the benefits of EdgeSplit. We then illustrate the challenges to achieve efficient EdgeSplit.

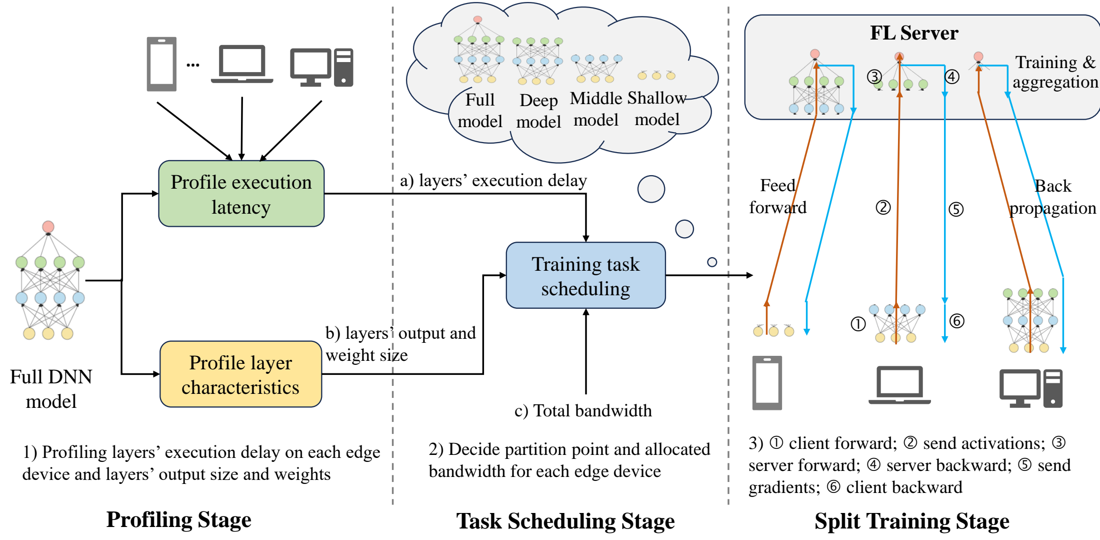


Figure 4.3: Framework of EdgeSplit. It consists of three stages: model and device profiling, task scheduling optimization, and online split edge training.

4.3.1 EdgeSplit Framework

Fig. 4.3 shows the workflow of the EdgeSplit framework, which consists of three stages: model and device profiling, task scheduling optimization, and online edge split training.

At the profiling stage, EdgeSplit mainly performs two steps: 1) profiling the execution time of each layer of the neural network on different edge devices and the FL server; 2) profiling the size of output for each layer of the trained model. More specifically, we perform one training epoch on each edge device and the server, and then record the execution delays and output size of different DNN layers. We repeat this process dozens of times and then calculate the average delay for a more accurate and stable value. Note that the output size of each layer in the model is fixed and just needs to be profiled once. Those profiling information will then be used to support intelligent task scheduling strategies, which decide how to partition the model and allocate the bandwidth between edge devices and the server.

At the task scheduling optimization stage, the EdgeSplit scheduler determines the best

partition point for each edge device and the bandwidth allocated to each edge device. The optimization problem aims to minimize the training time for accelerating the federated learning. It takes the following as inputs for the task scheduling algorithm: 1) the profiled average execution time of different model layers in all edge devices and the FL server; 2) the profiled size of output for each layer of the model; 3) the total bandwidth between the server and edge devices.

At the online split edge training stage, after getting the best partition points of the given DNN model, the partial model for each edge device are determined. The server first performs weights initiation and sends partial weights to edge devices, which will then conduct partial model training with localized data. After finishing the local feed forward process, edge devices will send the outputs of partial models to the server, which is responsible for the rest of feed forward training. The server also performs back propagation and sends back the gradients to edge devices for local back propagation and local weights updates. The edge devices then use the updated weights to perform the next batch of local feed forward. The process will be repeated several times, subjecting to the number of batches in an epoch. Finally, edge devices send back the updated weights, which will be aggregated in the server and go for the next round of training until the model gets convergence. The steps of one-round training of EdgeSplit are shown in Algo. 5.

Benefits. EdgeSplit is suitable for accelerating federated learning on heterogeneous edge devices for three reasons.

- **Resource-efficient.** Edge devices only need to train part of the full model, subjecting to the local computing capabilities and the bandwidth between an edge node and the server. It enables training resource-greedy AI models on resource-constraint edge devices, such as raspberry pi, which has only 1GB memory.
- **Communication-efficient.** In EdgeSplit, only activations of the partition

Algorithm 5: Procedures of a single round of EdgeSplit Training

Input: A Server S, DNN model W to be trained, M edge devices $E = \{e_1, e_2, ..., e_M\}_{i=1}^{M}$

1 The server decides the model partition points for each edge device and sends the initial weights $L_{e_1}, L_{e_2}, ..., L_{e_M}$;
2 **for** *epochs* **do**
3   **for** *batches* **do**
          `// Feed forward`
4   Each edge device performs feed forward of the partial model with localized data;
5   Each device shares the outputs/activations of the partial models to the server;
6   Server S finished the rest of the training;
          `// Back propagation`
7   Server S performs back propagation and sends back the gradients to edge devices;
8   Each edge device gets the gradients and does local back propagation and updates its weights;
9   **end**
      `// Weights aggregation`
10   Each edge device sends the updated partial weights to the server for aggregation;
11   Server S collects and aggregates all updated weights from edge devices and sends back the aggregated weights $L^u_{e_1}, L^u_{e_2}, ..., L^u_{e_M}$ to edge devices;
12 **end**
13 **return** Model W

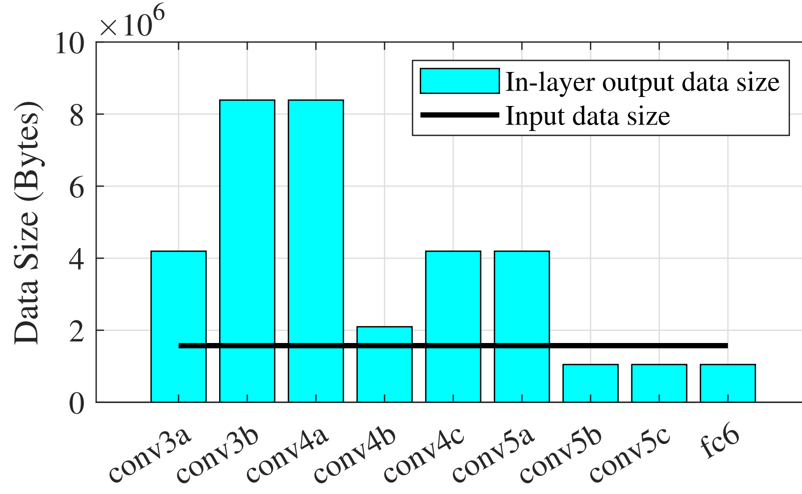


Figure 4.4: Output data size varies across DNN layers. Random model partition can hardly generate optimal execution latency.

layer and partial of full weights are transmitted between edge devices and the server, which reduces network pressure and achieves fast training.

- **No Accuracy Loss.** EdgeSplit only offloads partial training tasks to the FL server. It does not compress the data or modify any hyper-parameters of the training. There is no accuracy loss of EdgeSplit compared to the original FL.

4.3.2 Challenges

Despite its great benefits, there are still several challenges to achieve efficient and practical EdgeSplit.

First, determining the best partition points for a model on heterogeneous edge devices is non-trial. As shown in previous works [50, 60], simple decoupling of a deep neural network can hardly reduce its final execution latency in an edge-cloud deployment. This is because the in-layer output data, to be transferred across the network, is vastly different across the layers of a DNN model and is usually much larger than the original input data, due to the existence of the convolutional layers. Fig. 4.4 shows the data

size of some layers of a VGG16 model and the input data size with a 128 batch size on the CIFAR-10 dataset. Moreover, the overall training time is not only related to the execution latency of a single edge-server pair, but also largely influenced by other edge devices with heterogeneous resources. The overall training time is determined by the straggler. Furthermore, edge devices share the bandwidth. The allocated bandwidth to each edge device will affect the efficiency of data transmission.

Second, high memory consumption of the server leads to scalability issue. When there are $1,000$ edge devices participating in the federated learning, the server may have to load $1,000$ server-side models in the memory, which will be used to handle the activations from edge devices and complete the whole training process. Hosting such large number of DNN models will dramatically increase the burden of the FL server. It is suitable for cloud computing scenarios where the server is located in cloud data centers and can leverage the high flexibility of cloud servers to address the server-side pressure, i.e., allocating more cloud servers. However, for those less powerful servers, for example, base stations and edge gateways, it is still a concern.

4.4 Methodology

In this section, we detail the solutions to address the aforementioned challenges for achieving practical and efficient EdgeSplit.

4.4.1 Determine the Best Partition Points

The following shows details about the problem formulation and the task scheduling algorithms to decide the best model partition points for each edge device.

Table 4.1: List of notations

Symbol	**Descriptions**
$X_{i,j}$	binary variable, whether layer j of a model is the partition point for edge device i
$T^f_{i,j}$	forward time from layer 1 to layer j on device i
$T^b_{i,j}$	backward time from layer j to layer 1 on device i
$S^f_{i,j}$	forward time from layer $j+1$ to end layer on the server
$S^b_{i,j}$	backward time from end layer to layer $j+1$ on the server
M	total number of edge devices
N	number of partitionable layers of a DNN model
O_j	Output size of layer j of a given model
B_i	bandwidth allocated to edge device i
P_j	number of parameters from layer 1 to layer j

System Model. We consider a network consisting of M edge devices and a server. The edge devices have heterogeneous computation capabilities, and the server is located in the remote cloud and is much more powerful than edge devices in terms of computation capability. The edge devices and the server are interconnected, and the total bandwidth between the server and edge devices is denoted by B. Bandwidth between an edge device i and the server is B_i, $1 \leq i \leq M$.

DNN models are usually with a layered architecture. Different layers have different sizes of parameters and output data. We assume the model is with N feasible partition layers. O_j represents the size of output/activations of layer $j, 1 \leq i \leq N$. P_j is number of parameters from layer 1 to layer j. Other notations used in this thesis is shown in Table. 4.1.

Problem Formulation. There are three main contributing components to the training time in each round: 1) forward computation time of local edge devices and the

server; 2) back propagation time of the server and local edge devices; and 3) intermediate data transmission time to send/receive activations/gradients and weights between edge devices and the server.

In the forward phase, the computation time T^f_{batch} for edge device i in a batch is calculated by

$$T^f_{batch} = \sum_{j=1}^{N} T^f_{i,j} * X_{i,j} + \sum_{j=1}^{N} S^f_{i,j} * X_{i,j} \tag{4.1}$$

where $X_{i,j}$ equals to 1 if layer j of the model is selected as the partition point for edge device i. Otherwise, $X_{i,j}$ equals to 0. In Eq. (4.1), $\sum_{j=1}^{N} T^f_{i,j} * X_{i,j}$ represents the feed forward time on local edge device i, and $\sum_{j=1}^{N} S^f_{i,j} * X_{i,j}$ represents the rest of feed forward time performed on the server.

In the backward phase, the computation time T^b_{batch} for edge device i is calculated by

$$T^b_{batch} = \sum_{j=1}^{N} (S^b_{i,j} + T^b_{i,j}) * X_{i,j} \tag{4.2}$$

The communication time to send the activations to and get gradients from the server is calculated by

$$T^g_{batch} = \frac{\sum_{j=1}^{N} Oj * X_{i,j}}{B_i} * 2 \tag{4.3}$$

where $Oj * X_{i,j}$ is the amount of activations (gradients) to be transmitted if layer j is the partition layer for device i, i.e., $X_{i,j} = 1$. This is only one partition point for a model on an edge device, hence $\sum_{j=1}^{N} Oj * X_{i,j}$ indicates the amount of activations/gradients for device i. The activations and gradients are assumed to have the same size, as they are decided by the shape of the partition layer, i.e., the number of neurons in this layer.

Note that, edge device i also has to receive the initial weights from the server at the beginning of the training and sends back the updated weights when finishing a round of training. The communication time to receive and send the weights is calculated by

Eq. (4.4), where $\sum_{j=1}^{N} Pj * X_{i,j}$ indicates the amount of weights to be sent for edge device i.

$$T_{batch}^{w} = \frac{\sum_{j=1}^{N} Pj * X_{i,j}}{B_i} * 2 \tag{4.4}$$

Hence, the time for a round of training of edge device i is calculated by Eq. 4.5, where b is the number of batches.

$$T_i^r = b * (T_{batch}^{f} + T_{batch}^{b} + T_{batch}^{g}) + T_{batch}^{w} \tag{4.5}$$

Objective. The problem of minimizing the training time of EdgeSplit by jointly deciding model partition and bandwidth allocation is formulated as follows. We denote this problem as $P1$. As shown in the objective function Eq. (4.6), the overall training time is determined by maximum training time of edge devices, and our objective is to minimize the maximum training time for acceleration. Eq. (4.8) is the constraint. It ensures that there is one and only one partition point for the model on an edge device. Eq. (4.9) indicates the allocated bandwidth to edge devices should not exceed the total bandwidth.

$$\textbf{P1:} \quad \min_{X_{i,j}, B_i} \max \{T_1^r, T_2^r, ..., T_i^r\}_{i=1}^{M} \tag{4.6}$$

Subject to constraints:

$$X_{i,j} \in \{0, 1\}, \quad \forall i, j \tag{4.7}$$

$$\sum_{j=1}^{N} X_{i,j} = 1, \quad \forall i \tag{4.8}$$

$$\sum_{i=1}^{M} B_i \leq B \tag{4.9}$$

Problem Solution. There are both binary variable $X_{i,j}$ and continuous variable B_i in $P1$. The problem is hence a mixed integer non-linear problem, which is proven to be NP-hard in literature and is hard to solve. To solve the problem, we first simplify

the original problem $P1$ and then propose an efficient alternating algorithm to solve it. T_i^r can be simplified and rewritten as follows:

$$T_i^r = \sum_{j=1}^{N} (A_{i,j} + \frac{C_j}{B_i}) * X_{i,j} \tag{4.10}$$

where $A_{i,j} = b * (T_{i,j}^f + S_{i,j}^f + S_{i,j}^b + T_{i,j}^b)$ and $C_j = (b * O_j + P_j) * 2$. In Eq. (4.10), $A_{i,j}$ and C_j are deterministic and there are two variables $X_{i,j}$ and B_i. We observe that once fix B_i, there is an analytical solution for $X_{i,j}$, and once fix $X_{i,j}$, the original problem $P1$ becomes a convex problem. Instead of relaxing $X_{i,j}$ to a continuous variable, which is usually adopted, we propose an alternative minimization method to solve the problem, which alternatively search and optimize $X_{i,j}$ and B_i. There are three steps of the method.

Step 1: Fix B_i, then $P1$ can be solved directly. The corresponding analytical solution of $X_{i,j}$ is given by Eq. (4.11). It indicates that the selected partition layer j should be able to minimize the one-round training time of an edge device.

$$X_{i,j} = \begin{cases} 1, & j = \arg\min_j (A_{i,j} + \frac{C_j}{B_i}) \\ 0 & \text{otherwise.} \end{cases} \tag{4.11}$$

Step 2: Fix $X_{i,j}$ getting from step 1, and the original problem $P1$ is transformed to following problem $P2$.

$$\begin{aligned} \textbf{P2:} \quad & \min_{B_i} \max \{T_1^r, T_2^r, ..., T_i^r\}_{i=1}^{M} \\ & \sum_{i=1}^{M} B_i \leq B \end{aligned} \tag{4.12}$$

Note that, in problem $P2$, T_i^r is convex and the objective function $\max \{T_1^r, T_2^r, ..., T_i^r\}_{i=1}^{M}$ is the stagnation point maximum function of M convex functions, which is also convex. The constraint $\sum_{i=1}^{M} B_i \leq B$ is linear. Hence, problem $P2$ is a convex problem, which can be solved by convex optimizers [17].

Algorithm 6: Joint model partition and bandwidth allocation

Input: Profiled data $T^f_{i,j}$, $S^f_{i,j}$, $S^b_{i,j}$, $T^b_{i,j}$, $S^u_{i,j}$, $T^u_{i,j}$, O_j, P_j; total bandwidth B; D and batch size b

Output: the model splitting strategy $X_{i,j}$ and bandwidth allocation strategy B_i

```
1  Initialize B_i for all edge devices;
2  Initialize a large one-round training time T_opt ← INF;
3  for iterations do
       // Step 1
4      Calculate X*_{i,j*} by solving Eq. (4.11);
5      Fix model splitting strategy by assigning X*_{i,j*} ← 1;
       // Step 2
6      Solve convex problem P2 and get optimal objective value T^{r*}_{i*} and B*_i for all
         edge devices;
7      if T_opt > T^{r*}_{i*} then
8          T_opt ← T^{r*}_{i*};
9          Fix B*_i for all edge devices;
10     else
11         break;
12     end
13 end
14 return X*_{i,j*}, B*_i
```

Step 3: We then repeat the alternative minimization process, i.e., step 1 and step 2, for several times, and the solution of original problem $P1$ will be found.

The algorithm is shown in Alg. 6. More specifically, the input includes the profiled model characteristics. We first randomly initialize B_i and T_{opt} with an infinite value. We then performs Stage 1 by fixing B_i (line 1-5). After getting $X_{i,j}$, we solve problem $P2$ through a convex optimizer and get the solution of B_i (line 6). Finally, if the T_{opt}

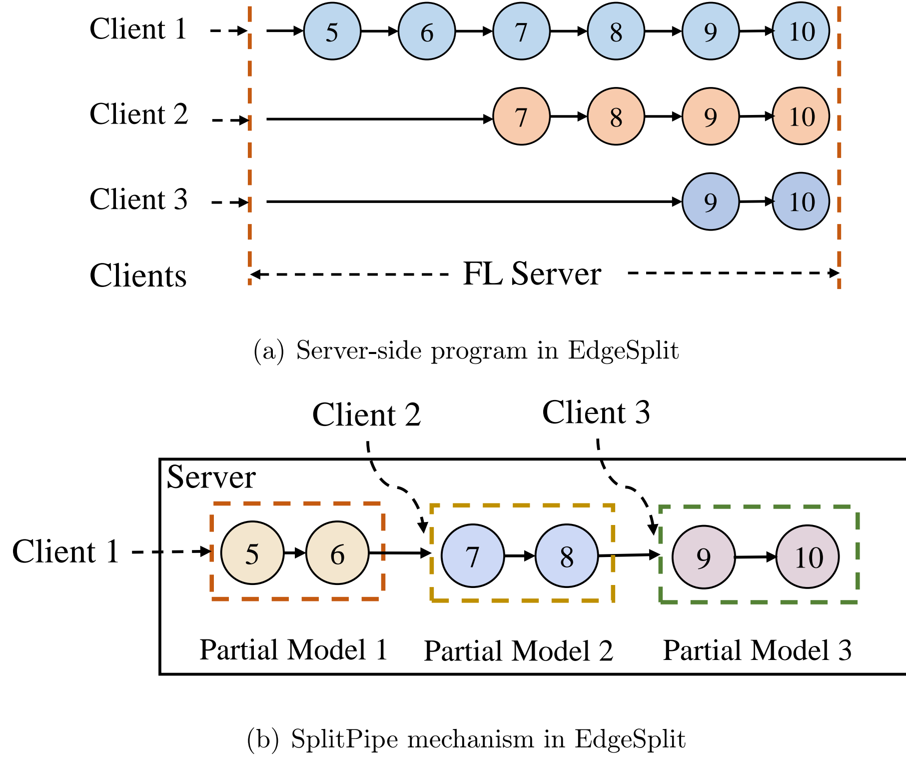
keeps decrease, we continue iterate this process (line 7-12). Otherwise, we break this process to get the optimal model splitting and bandwidth allocation strategy.

4.4.2 Alleviate Memory Overhead

In EdgeSplit, the partial models have to be loaded in the FL server, which will perform the rest of the model training tasks. As shown in Fig. 5.2(a), suppose there are 3 edge devices and the full model to be trained has 10 layers. The partition points of the 3 edge devices are $4, 6, 8$, respectively. The FL server thereby has to load three partial models with layer 5-10, 7-10, and 9-10. Hosting those partial models and their corresponding weights in the FL server leads to high memory consumption. We observe that when training 20 ResNet50 models at the server, the active peak memory goes up to 220GB. Further, the computation resource of the server is not fully utilized. For the partial model with layer $5-10$, when layer 5 is executed, layer $6-10$ consume the memory but wait for execution.

We notice that for those partial models, there are overlaps among the layers loaded into the memory. For example, layer $7-10$ are duplicated for partial model 1 and partial model 2, and layer $9-10$ are duplicated for partial model 2 and partial model 3. Hence, we propose a SplitPipe mechanism to reduce the memory consumption of the FL server by pipelining the partial model and reusing the duplicated layers on the server.

As shown in Fig. 5.2(b), the deepest partial model with layer $5-10$ is split into several small partial models, i.e., layer $5-6$, layer $7-8$, and layer $9-10$. Those partial models concurrently handle the tasks from client $1-3$. When layer $5-6$ finishes the computation tasks from client 1, it forwards the activations of layer 6 to layer $7-8$, which completes the computation task from client 2 and forwards the activations of layer 8 to layer $9-10$. Layer $9-10$ finishes the task request from client 3 and then handles the request from client 2. In such case, only layers $5-10$ are fed into the



(a) Server-side program in EdgeSplit

(b) SplitPipe mechanism in EdgeSplit

Figure 4.5: EdgeSplit with SplitPipe mechanism to reduce memory overhead by pipeling and reusing duplicated layers of partial models.

memory and can greatly reduce the memory. The partial models L_P to be loaded in the server is indicated by Eq. (4.13), where X^*_{i,j^*} is the model partition point for edge device i determined by Alg. 6 and $Index(\cdot)$ is a function returning the index of non-zero elements of a list.

$$L_P = Index(\sum_{i=1}^{M} X^*_{i,j^*}) \tag{4.13}$$

However, a partial model under the SplitPipe has to serve multiple task requests concurrently. For example, in Fig. 5.2(b), layer $7-8$ may have to handle the request both from client 2 and the request from its preceding layers, i.e., layer $5-6$. Different task requests correspond to different model parameters. To avoid loading multiple partial model parameters simultaneously, which may lead to high memory

Layer 9-10	F3	F2	F1	B3	B2	B1		
Layer 7-8	F2	F1				B2	B1	
Layer 5-6	F1							B1

(a) SplitPipe with First-in-First-Serve

Layer 9-10	F3	B3	F2	B2	F1	B1		
Layer 7-8	F2	F1			B2		B1	
Layer 5-6	F1							B1

(b) SplitPipe with Backward-First

Figure 4.6: Different task execution strategy of SplitPipe. Backward-First strategy reduces memory overhead by releasing memory space of cached intermediate variables.

consumption, we adopt the dynamic model loading technique, where a partial model saves the parameters into disk after handling a task request and then loads the new parameters from disk to serve the new coming task request. We denote this strategy as *SplitPipe-RW*.

SplitPipe-RW may lead to prolonged task executing time as the model parameters need to be written into the disk and loaded into the memory. We additionally design a *SplitPipe-BF* mechanism to reduce memory overhead to avoid frequent I/O operations, by optimizing the task execution procedure. An ideal case is that, after finishing the request from client 2, the request from layer $5-6$ just arrives. However, in practical systems, it may not be true, and the concurrent tasks may wait for execution. Fig. 4.6(a) shows a naive task execution strategy, which adopts a first-in-first-serve principle. F1, F2, and F3 represent the forward calculation of client 1, client 2, and client 3, respectively. Similarly, B1, B2, and B3 indicate the back propa-

gation. In this case, partial models in Fig. 5.2(b) will maintain a task queue and serve the requests in the queue one by one according to request arrival time. To further reduce the memory overhead, the SplitPipe-BF mechanism adopt a backward-first principle, as shown in Fig. 4.6(b). In SplitPipe-bf, the backward task has the high priority. This is because in the forward propagation stage during model training, the intermediate results, e.g., weights and activations of each layer, are required to be cached and used for gradient calculation during back propagation. Those accumulated intermediate results will lead to high active peak memory. The backward-first strategy can release unnecessary intermediate variables in time and reduce memory consumption. Taking F2 of Layer 9-10 (the forward calculation of the client 2 of layer 9-10) as an example. Before F2 is calculated, the reverse B3 of client 3 (the reverse calculation of layer 9-10) is performed first. After the calculation is completed, the intermediate variables of F3 can be released, so that F2 can reuse the memory space of the intermediate variables of F3.

SplitPipe can efficiently reduce the memory consumption in the FL server, adapting to cases when a resource less-abundant edge server at the network edge acts as the FL server rather than the cloud servers in a data center. Moreover, SplitPipe is suitable for the scenarios when there are multiple FL servers or one server with multiple computing hardware, e.g., GPUs. In such scenarios, each partial model can easily feed into a computing device, and fully utilize the computing resources.

4.5 Experimental Evaluation

4.5.1 Experimental Setup

Testbed. To test the system on a large-scale edge environment, we build a large-scale hybrid testbed with high fidelity with both physical and virtual edge devices, as shown in Fig. 4.7. We use Docker containers [5] to emulate virtual edge devices.

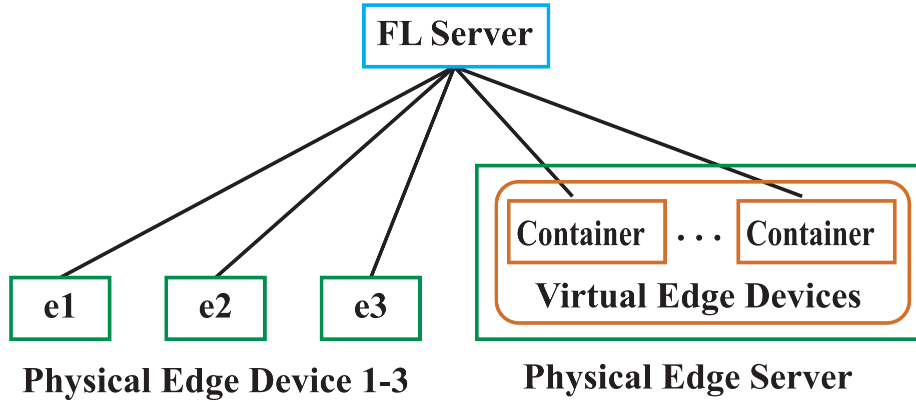


Figure 4.7: Hybrid testbed with physical and emulated devices

The Docker engine enables us to allocate different CPU resources to emulate resource heterogeneity. While numerous virtual edge devices enable us to test in a large-scale and network-flexible testing environment, the incorporation of physical nodes guarantees the fidelity of the testbed. We leverage Linux Traffic Control to configure the network topology and bandwidth among the edge devices and the FL server. We vary the network link bandwidth, e.g., from $1Mbps$ to $10Mbps$, to evaluate EdgeSplit under different network conditions.

Regarding edge devices, we use raspberry pi, Nvidia Jetson Nano, Nvidia Jetson Xavier NX, and Nvidia Jetson Xavier AGX to represent physical devices. Note that the Jetson devices are equipped with embedded GPUs and are much more powerful than raspberry pi. Among them, Xavier AGX has the best computation capability. A powerful server equipped with Intel(R) Core(TM) i9-10900F CPU with 64GB memory is used to act as the FL server to manage the edge devices and the training process. In terms of virtual edge devices, we use an edge server to host Docker containers acting as virtual edge devices. The edge server is equipped with Intel(R) Xeon(R) Gold 6128 CPU with 192GB memory and 128 cores. Docker enables us to flexibly configure the number of CPU cores and the memory of a container. The more CPU cores, the more powerful the container is in terms of computation capability. The specifications of the physical devices are shown in Table. 4.2.

Table 4.2: Specifications of heterogeneous physical devices

Name	CPU	Memory	Performance
Raspberry Pi	1 core	1GB	Low
Jetson Nano	6 cores	4GB	Low
Jetson Xavier NX	6 cores	8GB	Medium
Jetson Xavier AGX	12 cores	32GB	Good
Edge Server	128 cores	192GB	-
FL Server	64 cores	64GB	High

Benchmark. The performance of experiments is highly relevant to the input data and the models to be trained. To show the generality of EdgeSplit, we test various DNN models, including LeNet [57], VGG-16 [98], ResNet-50 [33], and ResNet-101 [113]. They are classical and representative models and extensively used in both academia and industry. Moreover, they have a variant depth of neural network from 5 to 101 layers, with distinct sizes of parameters and computation workloads.

We use three commonly used datasets, i.e., Fashion-MNIST, MNIST, and CIFAR-10. MNIST dataset consists of handwritten digit images of 0 to 9 (i.e., 10 classes). Each image has 28 $\times$ 28 pixels with a pixel value ranges from 0 to 255. It has a total of 60, 000 training and 10, 000 test samples. FashionMNIST consists of images of ten clothing, including T-shirts, trousers, pullover, dress, and coat. Each sample has 28 $\times$ 28 pixel grayscale images the number of training and test samples equal to the MNIST dataset. CIFAR10 consists of color images of ten objects (classes), including airplane, cat, dog, bird, automobile, horse, and ship. It has 50, 000 training and 10, 000 test samples. Each sample in CIFAR10 has 32 $\times$ 32 pixels. For each dataset, there are ten classes. We randomly assign 5 different classes for each edge device.

Baselines. We compare EdgeSplit with three methods.

- **Vanilla Federated Learning (Vanilla FL)**. In this case, all training tasks are

performed on local edge devices, and the server is only responsible for weights aggregation as in FedAvg [73]. The bandwidth is equally shared among edge devices.

- **FL with adaptive bandwidth allocation (Adaptive FL)**. Adaptive bandwidth allocation for accelerating FL is used in [86, 109]. In this case, training tasks are also done locally. However, the bandwidth allocation is decided by solving problem P2 in § 4.4.1. Adaptive FL is actually a special case of EdgeSplit, where the partition points for all edge devices are the last layer, indicating the model is trained locally.

- **EdgeSplit with fixed partition point (EdgeSplit-Fix)**. In this case, partition points for all the edge devices are the same, as in SplitFed [104]. We allocate half of the layers on the edge devices and half of them on the FL server. The bandwidth allocation strategy is the same as Adaptive FL.

4.5.2 Results and Analysis

We test the performance of the EdgeSplit and baselines under various situations. In all the following experiments, the client selection ratio is set to be 1, which means all edge devices participate in the training. The batch size is set to be 128 and the local epoch is 1.

Overall Evaluation. We first test the one-round training time with four physical devices, i.e., raspberry pi, jetson nano, jetson xavier NX, and jetson xavier agx. The results are shown in Table. 4.3. We can see that, due to the memory constraints, raspberry pi, jetson nano, and jetson xavier NX cannot train large DNN models, i.e., VGG16, ResNet50, and ResNet101. We observe that when training the ResNet50 and ResNet101, the memory footprint can go up to 11GB, which exceeds the capacity of those devices. However, EdgeSplit enables large model training on resource-constraint

devices by partitioning the model. Moreover, EdgeSplit also achieves training acceleration compared to vanilla FL and Adaptive FL. EdgeSplit achieves 1.52x speed acceleration of one-round training when training LeNet on the MNIST dataset.

Table 4.3: One-round training time on physical devices (s)

Name	Vanilla FL	Adaptive FL	EdgeSplit	Acceleration
LeNet	370.2	369.4	243.4	1.52x
VGG16	-	-	614.6	-
ResNet50	-	-	3407.6	-
ResNet101	-	-	2203.8	-

– baselines cannot perform training due to limited memory

Table 4.4: Comparison of per-round training time. Best partition points and acceleration ratio to vanilla FL are given.

Model	Vanilla FL	Adaptive FL	EdgeSplit-Fix	Ratio	EdgeSplit	Best Partition Points	Ratio
LeNet	237.9	234.2	164.3	1.4x	134.8	[1,1,1,1,2,2,4,4]	1.76x
VGG16	969.6	953.2	384.2	2.5x	243.4	[1,1,3,3,13,13,13,13]	3.9x
ResNet50	1806	1800.4	696.6	3.0x	330.2	[1,1,1,1,2,2,49,49]	5.5x
ResNet101	1355.7	1352.3	693.6	1.9x	308.1	[1,1,1,1,1,1,8,8]	4.4x

We then use the hybrid testbed to emulate those heterogeneous edge devices to quantitatively study the performance of EdgeSplit. We want to know how much acceleration EdgeSplit can achieve. We set the memory of those emulated devices (i.e., containers) as 12GB to train those large models. Moreover, we use the CPU cores ranging from 1 core to 5 core to emulate three types of virtual edge devices with heterogeneous computation capabilities.

Table. 4.4 shows the per-round training time with 8 edge devices under $30Mbps$ bandwidth. We can see that model partition methods, i.e., EdgeSplit-Fix and EdgeSplit, show obvious acceleration. This is because splitting the model can reduce local training time and offload partial computational tasks to the powerful FL server for

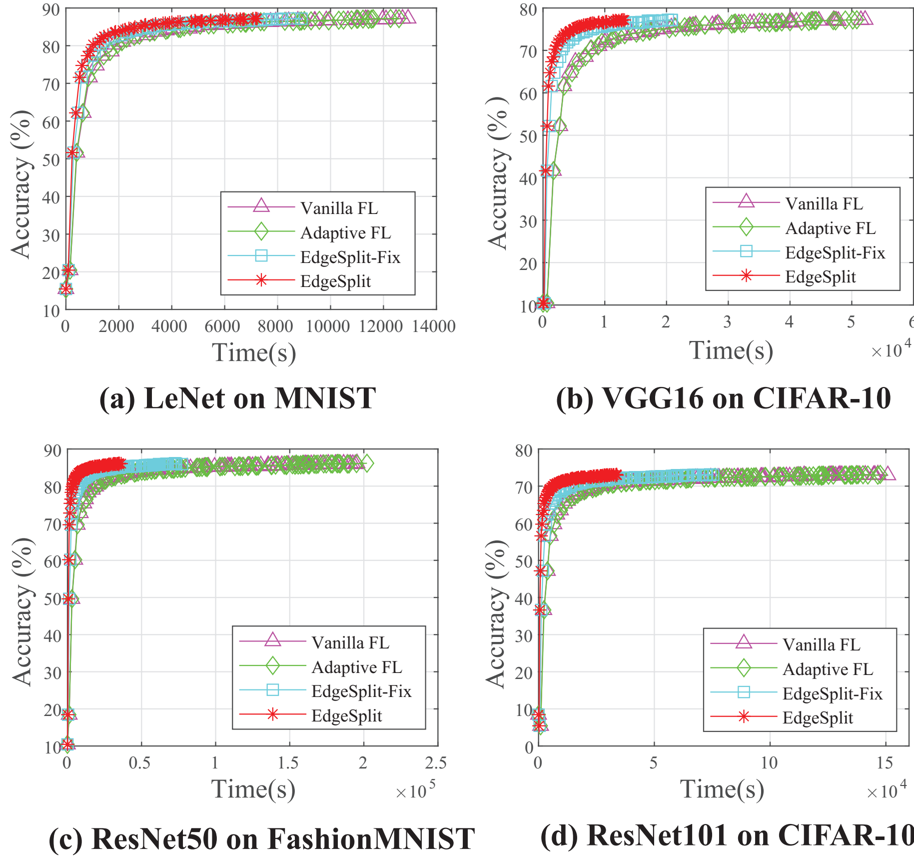


Figure 4.8: Convergence Time v.s. Accuracy. EdgeSplit achieves fast convergence without accuracy loss

acceleration. EdgeSplit outperforms EdgeSplit-Fix, achieving up to 5.5x acceleration when training ResNet50, as it can adaptively adjust the partition points for each edge device considering the heterogeneous local resources. We also observe Adaptive FL achieves similar performance with Vanilla FL. This is because the bandwidth is relatively abundant in this case, and the main bottleneck is from the computation. More study about the influence of bandwidth is shown in later experiments. We also train the models into convergence. Fig. 4.8 shows the accuracy changes with the training time. EdgeSplit shows less training time in each round and can achieve fast convergence without accuracy loss, compared to the various baseline methods.

Effects of Number of Edge Devices. We set the total bandwidth as 30Mbps. As shown in Fig. 4.9, the one-round training time of baseline methods and EdgeSplit

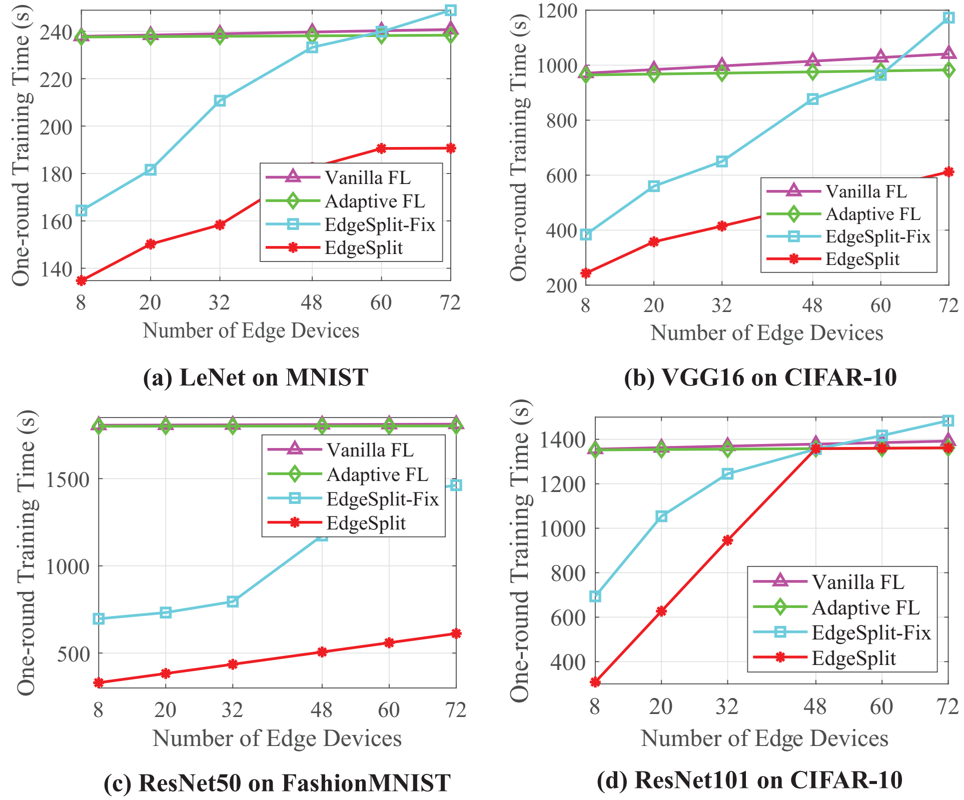
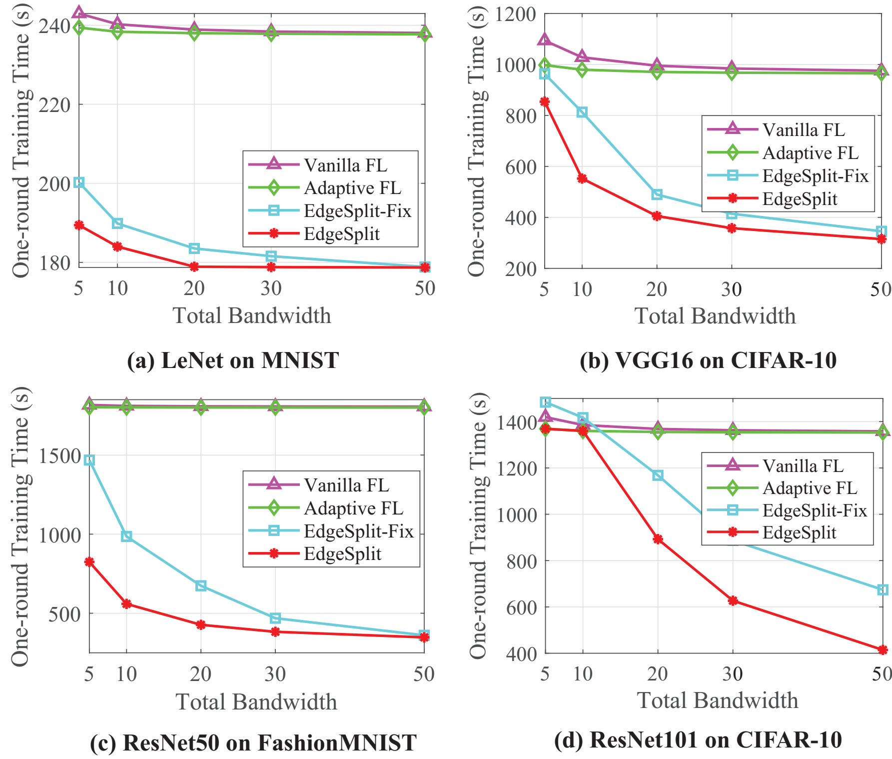


Figure 4.9: Impact of Number of Edge Devices

increases with the number of edge devices from 8 to 72. In this case, the average bandwidth for each edge device decreases with the increasing number of edge devices, which leads to prolonged parameter transmission and increased one-round training time. However, compared to vanilla FL and adaptive FL, the increment of one-round training time for EdgeSplit and EdgeSplit-Fix is much more apparent. This is because they need to frequently transmit activations and gradients between edge devices and the FL server. Massive edge devices lead to transmission congestion and thus cause an increase in one-round training time. Things become worse for EdgeSplit-Fix. As shown in Fig. 4.9 (a), (c), and (d), EdgeSplit-Fix consumes more time to train the model when there are 72 edge devices. In this case, edge devices are expected to burden more computational tasks as the average bandwidth is relatively limited. However, EdgeSplit-Fix cannot adjust the amount of local computation as the partition points are fixed.



(a) LeNet on MNIST

(b) VGG16 on CIFAR-10

(c) ResNet50 on FashionMNIST

(d) ResNet101 on CIFAR-10

Figure 4.10: Impact of Total Bandwidth

We can also see from Fig. 4.9(d) that the one-round training time of EdgeSplit and adaptive FL is the same when the edge devices are more than 48. It shows that the two methods yield the same model splitting and bandwidth allocation strategy. This is because the data transmission cost is much higher in this case. EdgeSplit tends to choose local model training rather than splitting the model. It also shows that adaptive FL is a special case of EdgeSplit, where the whole model is trained locally. Overall, the performance of EdgeSpit should not be worse than adaptive FL.

Effects of Total Bandwidth. We set the number of edge devices as 20. As shown in Fig. 4.10, the one-round training time decreases with the increasing total bandwidth as the data transmission cost becomes smaller. The speed acceleration ratio becomes larger as the total bandwidth increases. For example, the acceleration ratio is about 1.3x under 5Mbps and then goes up to 3.1x under 50Mbps condition for VGG16. A similar trend is also observed for ResNet50 and ResNet101. We also observe that in

some cases, the acceleration ratio is about 1x, i.e., for ResNet101 when bandwidth is below 10Mbps. In such a case, the local computation time is the dominant factor, and EdgeSplit tends not to split the model. It means the model training is performed locally and the improvement is due to the bandwidth adjustment. Moreover, we see that adaptive FL is much better than vanilla when the bandwidth is relatively low. The gap between the two methods becomes small with the increasing total bandwidth. For example, the gap in Fig.4.10(b) changes from 96s to 10s with the bandwidth increasing from 5Mbps to 50Mbps. This is because adaptive FL can efficiently utilize the bandwidth and allocate the bandwidth across edge devices when the bandwidth is low. Compared to adaptive FL, EdgeSplit not only optimizes the bandwidth allocation but also optimally decides the model splitting, which shows apparent training acceleration.

Effects of SplitPipe We study the effects of SplitPipe on the memory usage and training performance of EdgeSplit. We set the number of edge devices as 8 and the total bandwidth as 30Mbps.

Table 4.5: Performance of EdgeSplit with different serverpipe mechanisms. Memory: the active peak memory footprint within the training; Round-Time: the average round time in the training.

	LeNet		VGG16		ResNet50		ResNet101	
	Memory(GB)	Round-Time(s)	Memory(GB)	Round-Time(s)	Memory(GB)	Round-Time(s)	Memory(GB)	Round-Time(s)
Split-NonPipe	0.2	124.6	8.81	218.1	87.9	283.5	14.336	285.6
SplitPipe-RW	0.038	136.1	1.47	245.2	15.44	333.4	2.46	312.6
SplitPipe-BF	0.163	134.8	6.93	243.4	64.4	330.2	10.12	308.1

We compare the performance of EdgeSplit with three mechanisms. Split-NonPipe indicates EdgeSplit without SplitPipe approach, where the partial models are separately loaded in the Fl server, as shown in Fig. 5.2(a). SplitPipe-RW represents EdgeSplit with SplitPipe mechanism adopting dynamic model loading strategy, and SplitPipe-BF indicates Backward-First principle. As shown in Table 4.5, the memory consumption of SplitPipe-BF and SplitPipe-RW is much lower than that of Split-NonPipe. SplitPipe-BF achieves around 18.6%, 21.3%, 26.7%, and 29.4% less active peak mem-

ory consumption than SplitPipe-Non when training LeNet, VGG16, ResNet50, and ResNet101, respectively. This is because the SplitPipe approach only needs to load sequential partial models into the memory, and it adopts a Backward-first principle when handling the concurrently arriving task request, which releases unnecessary intermediate results in the training and hence reduces memory consumption. We also observe that though the active peak memory of SplitPipe-RW and SplitPipe-BF is obviously less than that of Split-NonPipe, performance of SplitPipe-RW is better than SplitPipe-BF. This is because SplitPipe-RW adopts dynamic model loading strategy whose memory consumption is approximate to that of the longest partial model with the most layers.

In terms of per-round training time, Split-NonPipe outperforms SplitPipe methods. Split-NonPipe achieves about 7%, 10.3%, 14.1%, and 7.3% less average round-time than SplitPipe-BF when training LeNet, VGG16, ResNet50, and ResNet101, respectively. The reason is that a partial model in the SplitPipe schema has to burden the training task of multiple edge devices simultaneously. The high workload may cause the tasks in the queue waiting to be processed, and thus lead to a long per-round training time. Though SplitPipe-RW has to frequently write and read model weights from the disk, its performance is only slightly worse than SplitPipe-BF, achieving around 1.3s, 1.8s, 3.2s, and 4.5s more average round-time. This is because the model parameters saving and loading time are much lower than the training time. We observe that the model saving and loading time is about 0.2s for the ResNet50 model in the server.

Overhead The main overhead of the system comes from the running time of the task scheduling algorithm. We test the algorithm execution time with the change of number of edge devices. The algorithm is running on the FL server, which generate the model splitting and bandwidth allocation strategy. As shown in Fig. 4.11, for all four models, the algorithm execution time increases with the number of edge devices. Our algorithm is simple and efficient. When there are 30 edge devices, the running

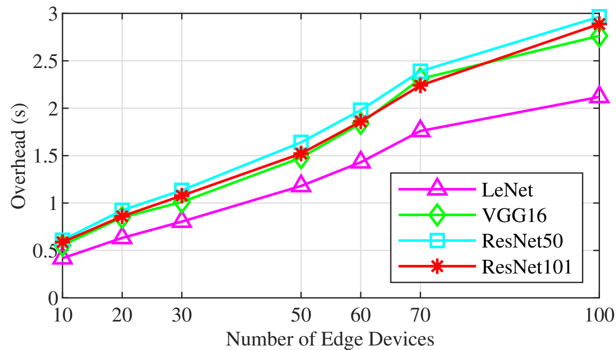


Figure 4.11: Impact of number of edge devices on system overhead.

time is approximately 1s. Even though there are 100 edge devices, the execution time will not exceed 3s, which can be ignored compared to the training time of the four models.

4.6 Conclusion

In this work, we propose EdgeSplit to accelerate federated learning on heterogeneous and resource-constraint edge devices. Through partitioning the full DNN model adapting to the heterogeneous edge resources, EdgeSpit enables large model training on low-resource edge devices and offloads part of the training task to the powerful FL server to reduce model training time. Experiments under various settings show the performance of EdgeSplit is not worse than vanilla FL and its variant. When there is relatively abundant average bandwidth, EdgeSplit can achieve apparent acceleration by jointly deciding the optimal model partition point and bandwidth allocation for each edge device.

Chapter 5

Scheduling Model Inference Tasks

In this chapter, we design and develop an edge-native task scheduling system for collaborative edge computing. This chapter is organized as follows. We present an overview of this work in Section 5.1. Section 5.2 summarizes the related work and articulates the motivations of this work. Section 5.3 presents the system design of distributed and collaborative edge intelligence for trustworthy and real-time video surveillance. Section 5.4 formulates the joint stream mapping and task scheduling problem and elaborates on the proposed heuristic algorithm. Section 5.5 shows the system implementation and performance evaluation. Finally, Section 5.6 concludes this chapter.

5.1 Overview

Nowadays, cameras have been widely deployed in public and private areas, such as traffic intersections, campuses, and grocery stores [129]. Driven by the recent breakthrough in machine learning, especially deep learning, we can perform advanced video analytics from the camera streams with deep learning models [76]. Collaborative video surveillance aims to analyze the live video streams from the distributed cameras to

support a wide range of applications, including traffic control [68], security monitoring, and object re-identification [4]. Because many of these applications have high requirements for real-time responses, it is highly demanded to achieve low-latency and high-throughput video stream processing.

There are many existing cloud-based video surveillance solutions [49]. However, these solutions are centralized and reactive [125]. More specifically, video streams collected from different cameras are sent to a centralized cloud server for storage and analytics [26]. Due to the massive data transmission among cameras and servers, such a cloud-based approach incurs bandwidth congestion, single point of failure, low scalability, and high latency. Furthermore, the cloud-based solutions raise privacy concerns, especially when the videos are confidential, e.g., smart home and warehousing [19].

In the past few years, there have been emerging works of edge computing-based video surveillance that can reduce latency, improve scalability, and provide better services compared to centralized surveillance solutions. This is because the computation tasks are pushed to network edges closer to the data sources [91]. Some existing edge computing-based video analytics solutions have considered leveraging the spatial and temporal correlations among different cameras to fully discover the correlation among distributed cameras [45]. There are also proactive video surveillance solutions for vehicle tracking using edge devices [115]. Some works further considered the aspects related to workload balancing to improve the performance of each edge device and the whole video analytics applications [120] [41].

To summarize, existing works mainly consider sending the video streams to a nearby edge computing device, which may lead to overloading and degrade application performance dramatically [123]. This is because the video analytic applications are usually computation-intensive while multiple video streams share the limited computation resources of a single edge device [124]. Some works handle the resource constraints of a single edge device by offloading some computation tasks to the remote cloud [49] [14]. However, it bears unpredictable latency and limited bandwidth,

Table 5.1: Comparison of the related work of video analytics

Video Analytics Solutions	Architecture	Adaptive Workload	Model Offloading	Stream Scheduling
VideoStorm [121]	Edge-Cloud	✗	✗	✗
Chameleon [46]	Edge-Cloud	✗	✗	✗
NoScope [49]	Edge-Cloud	✓	✓	✓
STVT [115]	Edge-Cloud	✓	✓	✓
LAVEA [119]	Edge-Edge	✓	✓	✓
VideoEdge [41]	Edge-Edge	✓	✓	✓
Distream [120]	Edge-Edge	✓	✓	✓
BCEI (this work)	Edge-Edge	✓	✓	✓

and privacy concerns [26]. Instead, this work leverages distributed and collaborative edge intelligence, where geo-distributed edge devices collaborate by sharing computation and data resources in an edge network to accomplish the video surveillance tasks.

This thesis proposes a distributed and collaborative edge intelligence (DCEI) approach to support trustworthy and real-time video surveillance. The proposed solution enables proactive video surveillance, where multiple cameras stream the video to different edge devices sharing computation resources and data. The key challenge is how to design approaches that can efficiently sense the status of underlying edge resources and intelligently assign the computation tasks among the edge devices. We thereby design an online collaborative scheduler, taking the status of the resources of the distributed edge devices and the task characteristics as input and generating the distributed task execution policies. We formulate a joint stream mapping and task scheduling problem, which maps the video stream and distributes the computation tasks to multiple edge devices. By solving the problem, the scheduler decides

which edge device the video stream should be assigned, how the tasks should be partitioned, and where to execute the distributed tasks. Our system enables efficient computation offloading and significantly improves resource utilization for edge computing-based video surveillance. We have deployed a real-world prototype of pedestrian re-identification to examine the practicability and high efficiency of DCEI. More specifically, we deploy 7 Internet Protocol (IP) cameras at different locations in an indoor environment to run the video surveillance task. These cameras send video streams to a cluster of geo-distributed edge devices. The results show that the proposed system can achieve nearly real-time performance. We have compared DCEI with several benchmark solutions concerning various performance metrics, including throughput and latency. The results show that DCEI outperforms the state-of-the-art significantly.

The main contributions of this section are as follows:

- We propose distributed and collaborative edge intelligence (DCEI) approach, which is the first to study the collaboration among geo-distributed edge devices and generic for applications demanding trustworthiness and low latency.

- We apply DCEI for trustworthy and real-time video surveillance, in which we have studied a joint stream mapping and task scheduling problem for the first time and solved it using a heuristic algorithm.

- We deploy a real-world prototype of trustworthy and real-time video surveillance and conduct extensive performance evaluation. The experimental results indicate the superiority of DCEI over the benchmark approaches in terms of latency reduction and throughput improvement.

5.2 Related Work

Real-time video analytics has been considered a killer application of edge computing [3]. Existing edge computing-based video analytics solutions mainly rely on the cooperation among cameras, edge devices, and cloud servers to accomplish real-time data analytics for computation-intensive and bandwidth-hungry video surveillance applications. Among them, solutions based on edge to cloud collaboration [14,26,49,125] and edge to edge collaboration [41, 77, 119] have attracted most attention.

Edge-Cloud collaboration. Edge-cloud collaboration-based solutions can provide amplest computation resources owing to the participation of the cloud platform. Many initial efforts on video analytics have considered edge-cloud collaboration. Zhang et al. [121] proposed VideoStorm, which leverages an online scheduler on a cloud server cluster to process queries of thousands of video streams sent by edge devices. Other works also study workload partition between edge and cloud. [14] divided the ResNet Model into three parts, which are deployed at the end, edge, and cloud, respectively. The three parts collaboratively perform the inference tasks submitted by users. [125] adopted serverless-based infrastructures to facilitate fine-grained and adaptive partitioning of cloud-edge workloads.

Although edge-cloud collaboration benefits from the paradigms of both cloud computing and edge computing, it still suffers from unpredictable latency because of the limited bandwidth between edge and cloud. Moreover, sending the confidential video data to the cloud may cause privacy issues. Such solutions do not explore the potential of collaboration among edge devices to share resources.

Edge-Edge collaboration. Some works in literature have considered edge-to-edge collaboration, where each edge device collaborates with others to provide surveillance services jointly. Neff et al. [77] proposed REVAMP^{2}T, a pedestrian re-identification algorithm that works on an encoded feature representation for each identified individual. Because no raw figure of pedestrians is shared among edge devices, the concern

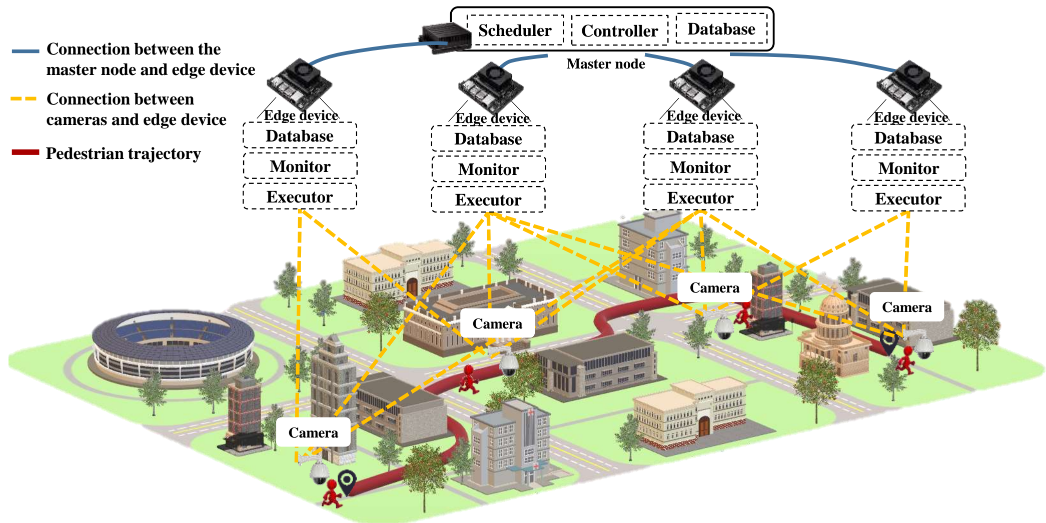


Figure 5.1: System model of distributed and collaborative edge intelligence system for video surveillance

of privacy is dramatically reduced. Another work in [44] also showed the advantage of leveraging spatial-temporal correlation to scale the video analytics systems to large camera deployments. The work in [47] presented the abstraction of camera clusters to provide video analytics service. However, those existing works only support data-level cooperation, and the distributed computation resources are not fully utilized.

Few attempts have been made to share computation resources among multiple edge devices for video analytics. Yi et al. have designed LAVEA, a low-latency video edge analytic system that leverages nearby edge devices to reduce the overall task completion time [119]. However, this work fails to consider collaborative video surveillance, in which there are multiple video streams from different locations. Recent work Distream [120] adaptively balances the workloads across multiple intelligent cameras and partitions the workloads between the smart cameras and the centralized edge cluster. However, the edge devices in the cluster are not geo-distributed and deployed in a centralized way, which ignores the data transmission cost and network topology among edge devices.

Compared to existing works, this work enables collaboration among geo-distributed

edge devices by sharing both computation and data resources. Moreover, it considers the stream transmission cost in the edge network and optimizes the real-time performance of video analytics applications by jointly scheduling the camera streams and the machine learning (ML) models. Table 5.1 shows a comparison of related works on video analytics with our method.

5.3 System Design

This section first introduces the system model to deploy trustworthy and real-time video surveillance applications. Then, we discuss the system's key components to manage edge resources and distribute the computation tasks among geo-distributed edge devices.

5.3.1 System Model

Fig. 5.1 depicts the overall architecture of our proposed distributed and collaborative edge intelligence approach. Multiple cameras are deployed to perform trustworthy and real-time video surveillance tasks in a large-scale area. Unlike the specialized intelligent cameras that can perform some computation-intensive video analytical tasks, we use low-cost commodity-off-the-shelf cameras whose computation capabilities are meager. Several geo-distributed edge devices with heterogeneous computation capabilities are deployed near the cameras to reduce the data transmission cost and provide near real-time response. Edge devices and cameras are interconnected within a network so that each camera can stream the videos to any edge device. Also, the computation tasks and data can be shared within the edge cluster.

Unlike cloud-based and traditional edge-based solutions, our system creates a shared resource pool with geo-distributed edge devices and cameras within a network. The benefits of the shared resource pool are as follows:

- *Efficient resource sharing and load balancing.* With a shared resource pool, the computation tasks can be migrated among edge devices to achieve load balancing and reduce the risk of single-node failure.

- *Flexible access to input video streams and intermediate data.* Instead of connecting one edge device with one camera, each device in the cluster can process multiple video streams, which facilitates effective data sharing.

- *Optimized scheduling to accelerate inference.* The computation tasks can be partitioned and scheduled among edge devices to reduce the execution time by jointly considering the computation and networking resources.

5.3.2 System Components

Our system enables distributed intelligence among geo-distributed edge devices, which share both computation resources and data to accomplish the video surveillance tasks. The key is to sense the distributed edge resources and intelligently distribute the computation tasks among the resource-constraint edge devices, considering the underlying edge resources' task characteristics and status.

The system adopts a master-client architecture, where the master is responsible for monitoring the edge resource status and scheduling decisions on partitioning and distributing the computation tasks among the edge devices. The key components of the system and their roles are described below.

- *Scheduler.* The scheduler runs at the master node and accesses the cluster status by communicating with the controller, which continuously monitors the workload and available resources of the cluster, such as network topology, data transmission rate, idle computation resources, and storage capacities, of each edge device. The scheduler generates task partition and resource allocation policies with a scheduling algorithm running inside.

- *Controller.* It monitors the edge resources by communicating with the monitors on edge devices. Also, it manages the edge resource by messaging the task partition and resource allocation policies to edge devices.
- *Executor.* The executor receives the task execution policies from the controller and executes the assigned tasks.
- *Monitor.* It monitors the networking and computation status of the edge device and sends it to the controller.
- *Database.* Each edge device has its local database instead of directly sending the inference results to the master node. The local database should be synchronized periodically with the global database on the master node to facilitate the distributed task execution and data sharing.

The general workflow of the system is described as follows. The computation tasks will be submitted to the scheduler, which generates the task execution policies by jointly considering the performance metrics of the task, available computation, storage, and networking resources of the edge devices. The policies decide how the task should be partitioned and where the task should be executed. More details will be presented in Sec. 5.4. The run-time characteristics of tasks and the resource status will be sent back to the controller by the monitor and used for future decision-making.

5.4 Joint Stream Mapping and Task Scheduling for Pedestrian Re-identification

We showcase the DCEI system with an edge video analytics application, namely pedestrian re-identification, empowered by deep learning models. We formulate a joint stream mapping and task scheduling problem to optimize the application latency, considering where to schedule the video stream and deploy the deep learning models.

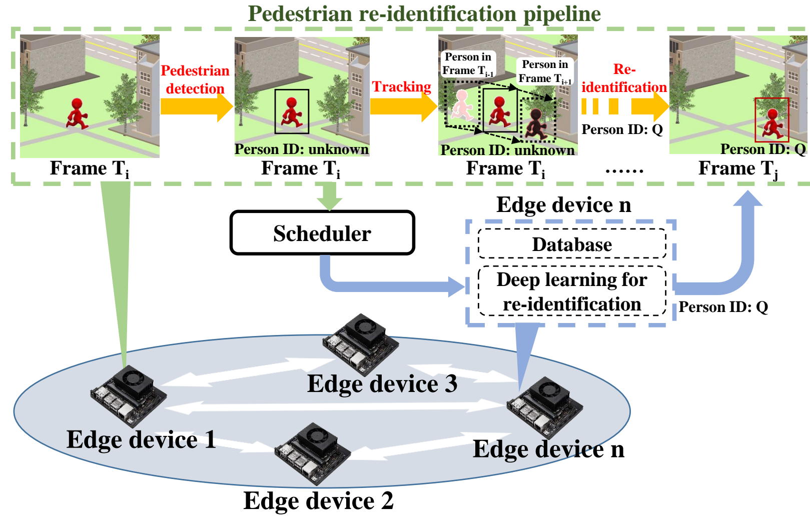


Figure 5.2: Model pipeline and task offloading of pedestrian re-identification

In this section, we first introduce the pedestrian re-identification pipeline. Then, we illustrate the benefits of jointly considering stream mapping and task scheduling. We mathematically formulate the problem and introduce the optimization algorithms based on this.

5.4.1 Pedestrian Re-identification Pipeline

We target those applications that employ state-of-the-art neural network models to conduct various challenging video analytic tasks. Examples include object attribute recognition and object re-identification, human activity recognition, and many others [131]. Those applications typically adopts a cascaded architecture that leverages a pipeline of neural network models to accomplish the video analytic task [120]. The pipeline usually consists of an object detection model followed by some task-specific models to perform various tasks on the detected objects within a video frame, e.g., object type, color, and shape.

Without loss of generality, we use the pedestrian re-identification application as an example. Pedestrian re-identification aims to associate images of the same person taken

from different or the same cameras at different times. It is an essential technique in video surveillance and has been widely applied in security areas such as pedestrian tracking, criminal event detection, and children remote monitoring [77] [4]. As shown in Fig. 5.2, a general process of pedestrian re-identification can be abstracted into a pipeline consisting of three phases, i.e., pedestrian detection, tracking, and re-identification. A detection model will first process an input image to detect the individual's location, and the identified individual is tracked by monitoring the trajectory. Afterward, discriminative features are extracted from the picture of tracked pedestrians for comparison and re-identification. The pedestrian is then annotated with the re-identification result during the later continuous tracking. The pipeline shows that the re-identification model is not frequently called because the individual will be assigned an ID and continuously tracked by the tracker once an individual is re-identified.

Fig. 5.2 also shows an example of distributed task execution policies. The pedestrian re-identification application is partitioned into pedestrian detection, tracking, and re-identification tasks. While the detection task and tracking are executed on edge device 1, the re-identification task is executed on edge device n. Edge device 1 sends the intermediate data to edge device n to accomplish the overall task.

We use two deep learning models for pedestrian detection and re-identification. The execution time of pedestrian tracking is much less than detection and re-identification as we do not adopt the computation-intensive deep learning (DL) models for tracking. Hence, we only consider the detection and re-identification models in the problem formulation part.

5.4.2 Motivations of Joint Stream Mapping and Task Scheduling

This section describes a concise example to show the motivations of joint scheduling of the video streams and deploying ML models on the edge devices. We consider a network consisting of three edge devices that are fully connected. The application model consists of two dependent tasks, i.e., pedestrian detection and re-identification, where we assume each task takes 1 unit of time to be executed on each device. There are four cameras that can stream the videos to any edge device. The first scheduling problem is to decide on the edge device where each camera stream the video. Once the video is streamed to an edge device, the device will use the ML model for the detection task to detect any individual within the frame. The other ML model is to identify the detected individual for the re-identification task. More specifically, the second scheduling problem is to decide which edge device to offload the re-identification ML model. The scheduling problems' decisions are based on resource availability, bandwidth constraints, and computation workload.

The scheduling result shown in Fig. 5.3(a) is based on randomly deciding the mapping of video streams from cameras to edge devices and locally executing the ML models in the pedestrian re-identification application. Such random scheduling of camera streams and no offloading leads to the possibility of multiple cameras streaming the video to one overloaded edge device, while other edge devices can either be lightly loaded or even have no workload. In our example scenarios, three cameras (a, b, c) stream video to edge device 1, while camera d streams video to edge device 2, and there is no video streamed to edge device 3. The completion time for this random scheduling is high (6 units in our example) and not optimal as it leads to inefficient usage of device resources. An improvement over the random scheduling is shown in Fig. 5.3(b), where we schedule the video streams from cameras while executing the ML models locally on the edge devices. The scheduling of camera streams shown in Fig. 5.3(b) leads to a

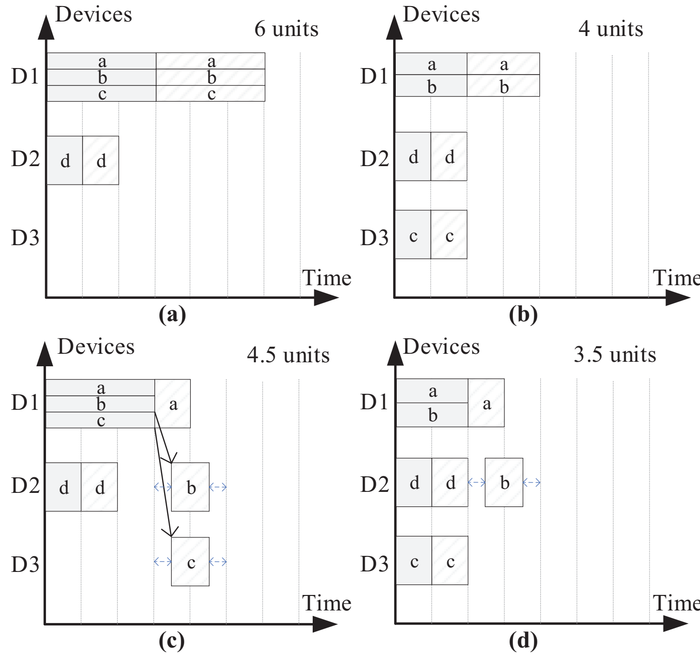


Figure 5.3: A motivation example of joint stream mapping and task scheduling: (a) random camera streams and local execution; (b) scheduling camera streams and local execution; (c) random camera streams and offloading ML models; (d) joint scheduling of camera streams and offloading ML models.

reduction in the completion time of the application from 6 to 4 units as it considers the transmission time between cameras and edge devices. Another alternative approach for improvement over the random scheduling is to make a scheduling decision on offloading the ML model depending on the resource availability instead of locally executing all the ML models as shown in Fig. 5.3(c). The scheduling result, shown in Fig. 5.3(c), also leads to a reduction in completion time from 6 to 4.5 units compared to random scheduling.

Although scheduling camera streams or offloading ML models separately leads to some improvements in terms of task completion time, there are still further spaces for efficiently utilizing the geo-distributed edge resources. Fig. 5.3(d) shows a joint approach of scheduling camera video streams and offloading the ML models, which leads to the best performance in terms of application completion time compared to the random or other scheduling approaches. We have designed an online scheduler in the

proposed DCEI system to enable joint scheduling of camera streams and offloading ML models in the pedestrian re-identification application scenario. The following sections show the problem formulation and proposed solution for this joint scheduling problem in a dynamic distributed edge computing environment.

5.4.3 Problem Formulation

A random mapping of the camera video streams to edge devices leads to an unbalanced workload and inferior performance. We formulate a joint stream mapping and task scheduling problem to improve resource utilization and boost application performance.

We assume a quasi-static slotted time model, where it is assumed that in each time slot t, the controllers are aware of the different network and device metrics required to make the decision. The metrics for different resources are assumed to be constant in a time slot. The system consists of K cameras. Each camera can stream the video to any edge device in the cluster. The network and application model can be defined as:

Network model. The network is modelled as a graph $G = (V, E)$, where $V = \{i|1 \leq i \leq M\}$ is the set of edge devices and $E = \{e_{ij}|i, j \in V\}$ is the set of links connecting different devices. The weight of each device is PS_i, representing the computation capacity of the device i. Each device also has a maximum resource of R^i_{max} and the available resource is indicated by R^i_{avail}. We only consider the memory in this section for the various resources of edge devices, e.g., CPU, memory, and storage. The weight of link e_{ij} represents the bandwidth between devices i and j. The devices and network links can be heterogeneous in computation capacity and bandwidth capacity, respectively. The data rate for transmission between any two edge devices is R_{ij}. The video transmission time between camera k and edge device i is T_{ik}.

Application model. The application model for the pedestrian re-identification applica-

tion studied in this section is a sequential DAG of two dependent tasks, i.e., detection and re-identification. The computation load for detection and re-identification tasks corresponding to camera stream k is $CL_{1,k}$ and $CL_{2,k}$ respectively. The dependent data between the two tasks is $D^k_{1,2}$. Resource request of the detection model and re-identification model is R^{det}_{req} and R^{reid}_{req}, respectively.

Decision variables. Two decision variables correspond to mapping camera streams and scheduling DL models. The first decision variable x_{ik} is binary, which is equal to 1 if camera video stream k is scheduled to device i. It also indicates that the detection task is deployed on device i. Another decision variable y_{ik} is also binary, which is equal to 1 if the re-identification task corresponding to camera video stream k is scheduled to device i.

Cost model. For each camera stream, either both the detection and re-identification tasks can be locally executed, or the re-identification task can be offloaded to another device. The time for executing the models on each device is dependent on the overall workload and available computation capacity.

The total resource request on device i, notated as R_i, can be calculated as follows:

$$R_i = \sum_{k=1}^{K} (x_{ik} \cdot CL_{1,k} + y_{ik} \cdot CL_{2,k}) \tag{5.1}$$

The overall processing time for camera stream k, i.e. L_k, can be calculated as:

$$\begin{aligned} L_k = & \sum_{i=1}^{M} x_{ik} \cdot T_{ik} + \sum_{i=1}^{M} x_{ik} \cdot \frac{CL_{1,k}}{PS_i} + \sum_{j=1}^{M} y_{jk} \cdot \frac{CL_{2,k}}{PS_j} \\ & + \sum_{i=1}^{M} \sum_{j=1}^{M} x_{ik} \cdot y_{jk} \cdot \frac{D^k_{1,2}}{R_{i,j}} \cdot \sigma(i-j) \\ & \forall j \in V, \quad k \in \{1, 2, \cdots, K\} \end{aligned} \tag{5.2}$$

where $\sigma(\cdot)$ is an indicator function. Only when $\cdot$ is zero, $\sigma(\cdot)$ equals to 1, otherwise

$\sigma(\cdot)$ equals to 0.

Objective function. The objective function of the problem is to minimize the sum of completion time for all applications from camera streams.

$$\min_{x_{ik},y_{ik}} \sum_{k=1}^{K} L_k \tag{5.3}$$

Constraints:

$$R_i \leq R^i_{max}, \quad \forall i \in V \tag{5.4}$$

$$\sum_{i=1}^{M} x_{ik} = 1, \quad \forall j \in V, \quad k \in \{1, 2, \cdots, K\} \tag{5.5}$$

$$\sum_{j=1}^{M} y_{ik} = 1, \quad \forall i \in V, \quad k \in \{1, 2, \cdots, K\} \tag{5.6}$$

$$x_{ik} = \{0, 1\}, \quad \forall i, j \in V, \quad k \in \{1, 2, \cdots, K\} \tag{5.7}$$

$$y_{ik} = \{0, 1\}, \quad \forall i, j \in V, \quad k \in \{1, 2, \cdots, K\} \tag{5.8}$$

Eq. 5.4 indicates that the resource request on an edge device cannot exceed its maximum resource. Eq. 5.5 and Eq. 5.6 show that the detection model and the re-identification model of a stream can only be deployed on one edge device. The formulated optimization problem is nonlinear integer programming (NLP) problem. The problem is NP-hard because the offloading problem can be reduced to a generalized assignment problem, proven to be NP-hard in literature.

5.4.4 Optimization Solution

We have proposed a joint stream mapping and task scheduling heuristic algorithm (JSTSH) that determines where to schedule each video stream and which edge device to allocate the detection and the re-identification task. The algorithm is developed considering two main principles.

Algorithm 7: Joint Stream Mapping and Tasks Scheduling Heuristic (JSTSH)

Input: Video stream $V = \{k_i\}_{i=1}^{K}$, $\{PS_i\}_{i=1}^{M}$ and $\{R_{avail}^{j}\}_{i=1}^{M}$

Output: Video and task allocation policy x_{ik} and y_{ik}

1 Create index I of streams in descending order of workload;
2 **for** $t \leftarrow 1$ *to* K **do** // Stage 1
3     $k \leftarrow I(t)$;
4   **for** *each device* $i \leftarrow 1$ *to* M **do**
5   **if** $R_{avail}^{j} > R_{req}^{det}$ **then**
6   Calculate the execution time $t_{exec}^{i} = T_{ik} + \frac{CL_{1,k}}{PS_i}$;
7   **end**
8   **end**
9   Calculate device i^* with the shortest execution time $i^* = min_i\{t_{exec}^{j}\}$;
10     $x_{i^*,k} \leftarrow 1$, update R_{i*} for device i^*;
11 **end**
12 **for** $t \leftarrow 1$ *to* K **do** // Stage 2
13     $k \leftarrow I(t)$;
14   **for** *each device* $i \leftarrow 1$ *to* M **do**
15   **if** $R_{avail}^{j} > R_{req}^{reid}$ **then**
16   Calculate the intermidiate data transmission time $t_{comm}^{i} = x_{ik} \cdot \frac{D_{1,2}^{k}}{R_{i,j}}$;
17   Calcualte the inference time of detection model $T_{reid}^{i} = \frac{CL_{2,k}}{PS_i}$;
18   Calculate the execution time $t_{exec}^{i} = t_{comm}^{i} + t_{reid}^{i}$;
19   **end**
20   **end**
21   Calculate device i^* with the shortest execution time $i^* = min_i\{t_{exec}^{j}\}$;
22   Add i to candidate list $M_{deployed}$;
23   **if** $V_{rest} \neq \emptyset$ **then**
24   Schedule remaining streams to the device that satisfies Eq. 5.9;
25   **end**
26   Update R_{i*} for device i^* $y_{i^*,k} \leftarrow 1$;
27 **end**
28 **return** x_{ik}, y_{ik}, $i = 1, 2, \cdots, M$, $k = 1, 2, \cdots, K$

- *High workload first.* Generally, the latency of handling video streams with high workloads is much larger than those with moderate workloads. Suppose we allocate edge resources to the video streams with a moderate workload first. In that case, those video streams with a high workload may suffer from prolonged latency when the rest of the edge resources are inadequate, further leading to increased average latency of all video streams. Hence, we leverage a priority list to rank all video streams according to their workloads.
- *Reuse re-identification model.* Following the pedestrian detection task, a re-identification model is used to perform the re-identification task. In our pedestrian re-identification pipeline, the re-identification model is not frequently called because an individual will be continuously tracked once the individual's ID is determined. For each video stream, if we jointly consider the deployment of the detection model and the re-identification model, it will lead to high resource consumption. Hence, we consider the deployment of the two models separately. When the available resources are constrained, the re-identification model can be reused, which means a re-identification model can handle multiple video streams.

Based on the two principles, we solve the problem in two stages. The first stage is to schedule the video stream and the detection task. The second stage is to schedule the re-identification task. We first determine the priority of stream scheduling by sorting the video streams according to their workloads. Then in the first stage, we filter the candidate edge devices with abundant resources to allocate the detection model. The video stream is allocated to the edge device with minimum execution time, including the raw video transmission time and the inference time of the detection model.

$$\min_i(x_{ik} \cdot \frac{D^k_{1,2}}{R_{i,j}} + \frac{CL_{2,k}}{PS_i}), \quad i \in M_{deployed}, k \in V_{rest} \tag{5.9}$$

After determining x_{ik}, we then allocate the re-identification models in stage 2. We

use a similar greedy idea to allocate the re-identification models as stage 1 does. The difference is that we consider the scenario that idle computation resources may be less abundant for deploying a re-identification model for each stream. Hence, we reuse the re-identification model, where a deployed model can serve multiple streams. To determine where to deploy the re-identification model, we allocate the re-identification models for those streams with high workloads. When the available resources cannot support the deployment of new re-identification models, we reuse the re-identification model. The rest of the streams V_{rest} will be allocated to the edge device, which satisfies Eq. 5.9. By solving Eq. 5.9, the rest of the streams will be scheduled to the deployed re-identification model that can provide the least intermediate data transmission and inference time.

5.5 Implementation and Performance Evaluation

This section presents the system implementation, evaluation metrics, and extensive experimental results of the proposed distributed and collaborative edge intelligence platform.

5.5.1 System Implementation

Testbed. We deploy the system in an indoor environment due to the privacy-preserving policies of the campus. The floor is shown in the upper right of Fig. 5.4, with the size $30 \times 15m^2$. We implement the system with 7 IP cameras, 5 edge devices, and 1 master device. We use 3 Jetson Xavier NX and 2 Jetson Tx2 as the edge devices. The Jetson Xavier NX has a 384-core Volta GPU with 8GB RAM, and the Jetson Tx2 has a 256-core GPU with 8GB RAM. The computation capacity of the former device is much stronger than the latter. Also, to emulate the heterogeneous resource, we constrain the memory of Jetson Tx2 as 4GB. We use a powerful PC equipped with

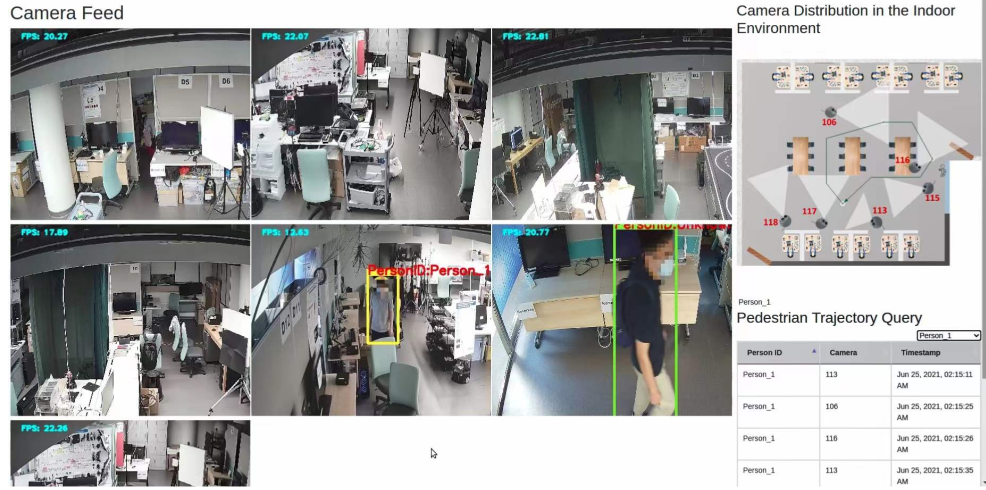


Figure 5.4: Demo of pedestrian re-identification (with mosaics for anonymization)

four Intel Cores i9-7100U with 32 GB RAM to emulate the master. Three routers connect the edge devices and cameras. To avoid single node failure, we use a backup server to synchronize the status of the master node and recover the system when the master node goes offline. To emulate the distance between the cameras and the edge devices, we use the Linux traffic control to configure the bandwidth between the cameras, edge devices, and edge devices.

Pedestrian re-identification pipeline. In the detection part, we use SSD-MobileNet-V2 [43] to process the input image and locate pedestrians. SSD-MobileNet-V2 is a lightweight deep convolution network that uses both depth-wise and point-wise convolutions to decrease model complexity, i.e., the number of parameters and operations. We uses Kalman filter-based tracker due to its low computation cost compared with DL-based approaches. Our target is to extract discriminative features for re-identification. Considering the constrained resources of edge devices, we build the feature extractor network with OSNet-AIN [2], which is light-weighted compared to ResNet-50 [34] and DenseNet [38]. It can learn the global representation of the individual's appearance and capture the subtle details required for the re-identification of individuals. We finetune the pretrained models of SSD-MobileNet-V2 and OSNet-AIN. All the models are developed following serverless principles and called Restful

APIs [15].

We test the accuracy of the deep learning models. We run the system for a whole workday, capturing around 346 pedestrians. Among which, 270 pedestrians are correctly re-identified with an overall accuracy of 78.3%. There are many false positives, i.e., different pedestrians are not differentiated and thus identified with the same identity, which is caused by differences in lighting and camera angles. More advanced approaches can be applied to improve accuracy. However, it is not the focus of this work.

Database. We use SQLite to record both the spatial and temporal information and the discriminative pedestrian features, as well as the video analytics results. The SQLite is a lightweight database often used in resource constraint embedded devices, such as mobile phones, cameras, and home electronic devices. Each edge device has its local database instead of directly sending the inference results to the master node, which is usually vulnerable to an unstable network connection. The local database should be synchronized periodically with the global database on the master node to facilitate collaborative inference and pedestrian tracing. The synchronization period is set to 10 seconds in our system. For example, when the data is not synchronized, the video analytics results are still in the edge devices. The master will forward the query request to edge devices and get the latest status from edge devices.

Fig. 5.4 shows the system interface. The camera views are on the left side of the interface, where we can monitor pedestrian behavior in real-time. Authorized users can also easily query the pedestrian trajectory, as shown on the right side.

5.5.2 Evaluation Metrics and Experimental Settings

We use the two following metrics to evaluate the performance of the proposed approach:

- *Throughput.* The cameras stream the input videos to the edge cluster continuously, in which high throughput is required to process these incoming video streams in real-time. Usually, Frames Per Second (FPS) measures the throughput. We define the system throughput as the average FPS for all streams.

- *Latency.* Live video analytics applications require producing analytics results within a short period. The task execution time for one stream includes video transmission time, local execution time, task offloading time, and remote execution time. Since there is multiple input video stream, we define the latency as the average task execution time for all streams.

The system schedules the video analytics tasks with stream mapping and task offloading (WMWO). We evaluate the system performance under different metrics and compare it against several benchmark solutions.

- No mapping no offloading (NMNO). The input video streams are randomly assigned to the edge devices, and all the computation tasks are executed locally.

- With mapping no offloading (WMNO). In this case, the input video streams are assigned to the edge devices with the shortest video transmission time, and all the computation tasks are executed locally.

- No mapping with offloading (NMWO). In this case, the input video streams are randomly assigned to the edge devices, and the re-identification task can be offloaded among the edge cluster.

NMNO and WMNO are non-offloading methods, and NMWO and WMWO are offloading methods. We test the performance of the proposed method and the baselines under various situations, i.e., the number of edge devices, the number of pedestrians in each input video stream, and different bandwidths of network links.

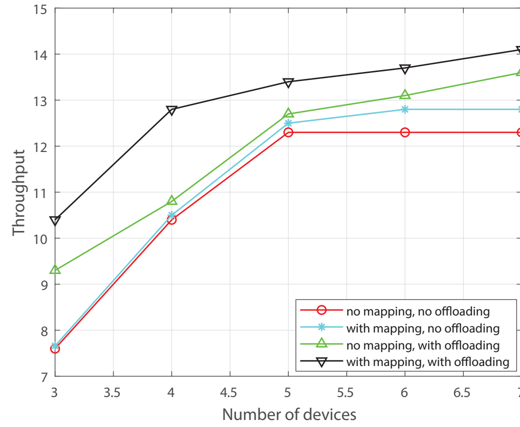


Figure 5.5: Throughput vs. number of devices

5.5.3 Influence of Number of Edge Devices

The number of the input video stream is set to be 5. We increase the number of edge devices from 3 to 7. As shown in Fig. 5.5, we can see that the proposed method, i.e., WMWO, achieves the highest throughput with the variation of the number of edge devices. When there are more edge devices than the input video streams, the NMNO method keeps the throughput consistent as it cannot utilize the computation resources of other edge devices. The WMNO achieves a slightly high throughput than NMNO as it maps the input video streams to edge devices, considering the heterogeneous computation capacities of the edge devices. Compared with WMNO, NMWO does not schedule the input video streams. However, it dynamically offloads the re-identification tasks, which can alleviate the computation workload on a single edge device and leverage the heterogeneous computation capacity of edge devices to improve the average throughput.

Though WMNO and NMWO can achieve higher throughput by either optimizing the stream mapping decision or the task offloading decision, their performance is not better than WMWO as WMWO jointly considers stream mapping and task offloading. WMWO achieves 14%-36% throughput improvement compared with baseline methods. It shows that our proposed method can integrate the heterogeneous resources

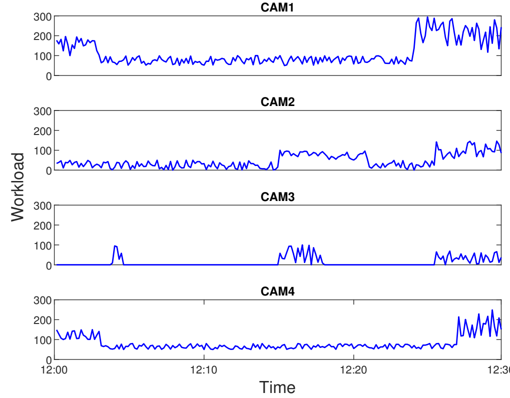


Figure 5.6: Workload dynamics in real-world deployment

of geo-distributed edge devices and improve application performance by jointly optimizing the mapping and offloading decisions to improve the application throughput.

5.5.4 Influence of Dynamic Workload

We study the performance of the proposed method and various benchmarks under dynamical workloads with a varying number of pedestrians in the input video streams. The number of video streams and edge devices is set as 4 and 5, respectively. Fig. 5.6 shows the workloads generated from the 4 cameras on a weekday between 12:00 to 12:30. The workload of each video stream varies with the number of pedestrians in a video stream and is dynamic as the content captured by each camera changes over time. We can also see that the workloads are different across cameras. CAM1 and CAM4 have a higher average workload than the other two cameras as they capture people entering or leaving out the doors.

As shown in Fig. 5.7, while the throughput of NMNO and WMNO fluctuate a lot over time, there is no noticeable change in offloading methods, i.e., throughput for WMNO and NMWO. NMNO and WMNO show apparent performance degradation when there is a high workload because edge devices with constrained resources may easily get overloaded, further leading to the deterioration of the average throughput. Offloading

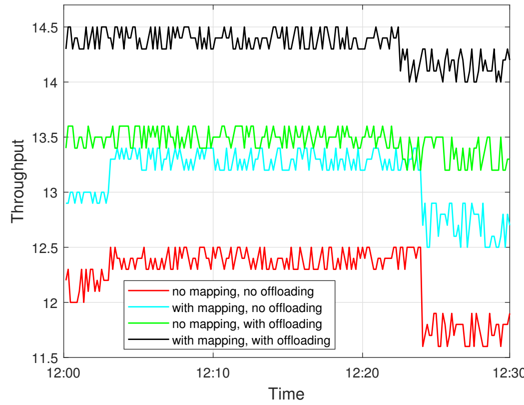


Figure 5.7: Throughput vs. dynamic workload

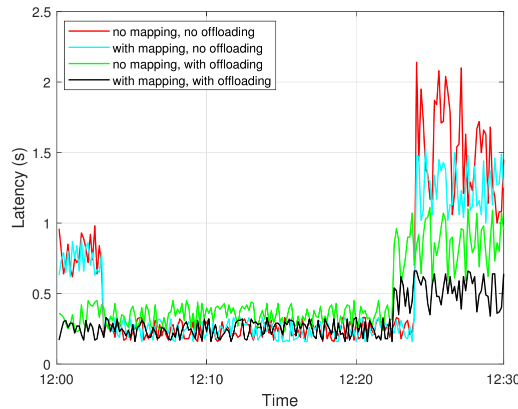


Figure 5.8: Latency vs. dynamic workload

methods show consistent performance because they leverage resource sharing and schedule the dynamic workloads among edge devices. Similar trends are also observed in terms of the latency of the application. From Fig. 5.8, we can also see that the latency of offloading the task is similar to that of without offloading when there is a low workload between 12:05 to 12:25, which is due to the data transmission latency of offloading the re-ID task and getting back the re-identification results. When there is a low workload, the data transmission latency is the main factor that constrains the end-to-end latency of offloading methods. As the workload increases, the transmission delay of intermediate data and model inference delay will increase. However, the inference latency caused by overloaded edge devices becomes the main factor in this

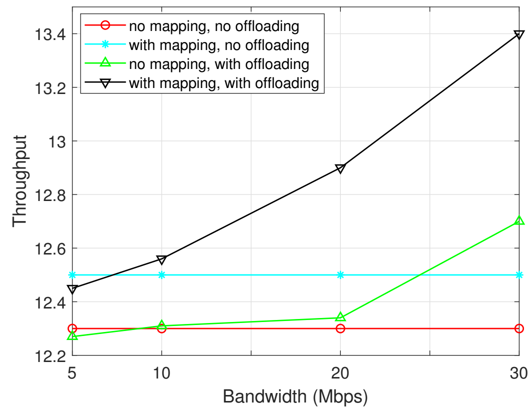


Figure 5.9: Throughput vs. bandwidth

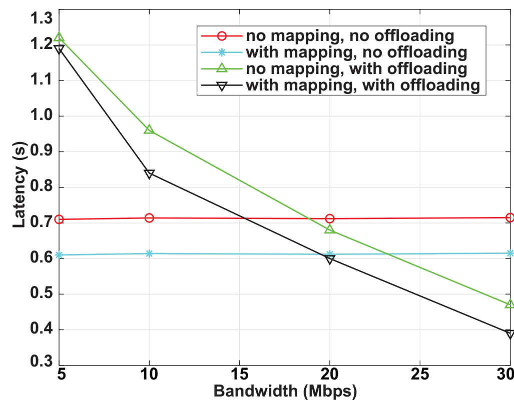


Figure 5.10: Latency vs. bandwidth

case. In this case, offloading methods show apparent superiority.

5.5.5 Effects of Bandwidth

We also investigate the performance of the system in different bandwidth conditions. Specifically, we leverage the Linux traffic control to manually set the bandwidth of each link and study the system throughput under the average bandwidth of 5Mbps, 10Mbps, 20Mbps, and 30Mbps, respectively. The variance of bandwidth of network links is 30%. For example, we increase the average bandwidth from 10 to 20 by doubling the bandwidth of each link in the network.

It can be seen from Fig. 5.9, while the throughput of non-offloading methods keeps consistent, the performance of offloading methods degrades with the decreasing bandwidth due to the increased transmission delay. When the average bandwidth is 30Mbps, the throughput of offloading methods is much better than that of the non-offloading methods. When the average bandwidths are 5Mbps or 10Mbps, the performance of offloading methods is similar to no-offloading methods. We find that the performance of offloading methods is highly affected by the transmission delay, as shown in Fig. 5.10. When there is a long transmission delay, the performance of offloading methods is approximate to non-offloading methods, which means that the optimal policy in such cases is to execute the re-identification tasks locally, i.e., no-offloading.

The experimental results indicate that our proposed distributed and collaborative edge intelligence (DCEI) approach can dramatically improve the real-time performance of video analytics applications, especially when the resources of edge devices are constrained. DCEI also shows superior performance in handling dynamic workloads as it integrates the geo-distributed resources and intelligently schedules the video streams and the inference tasks.

5.6 Conclusion

This chapter proposes a trustworthy and real-time video surveillance system with distributed and collaborative edge intelligence. We deploy the edge devices closer to the cameras and create a distributed edge devices cluster where computation and data resources can share within the cluster. We design a scheduler that jointly schedules the camera streams to edge devices and offload tasks in the application pipeline to improve resource utilization and performance. We have tested the efficacy of the proposed solution with a pedestrian re-identification on a real-world prototype.

Chapter 6

Conclusion and Future Directions

In this chapter, we conclude this thesis in Section 6.1 and present future research direction in Section 6.2.

6.1 Conclusion

In the past decade, due to the great advancement of deep neural networks, DL models and algorithms have been extensively used in various applications, including object detection, natural language processing, and autonomous driving. Traditionally, DL models are trained and deployed on cloud data centers, as they are usually computation-intensive and resource-greedy. Recently, edge computing has been proposed to enable the training and inference of DL models on edge nodes at the network edge closer to the data sources. However, the resource on edge nodes is usually constraint and may not be able to burden the training and inference tasks. Hence, scheduling and distributing those tasks among distributed computing nodes is urgently needed. However, scheduling those tasks in edge computing is vastly different from that in the cloud, as edge nodes are distributed and heterogeneous, connecting low-bandwidth and intermittent networks. It calls for new systems and algorithms.

In this thesis, we identify the challenging issues of scheduling AI tasks among edge nodes and cloud servers for high-performance training and inference, and address several critical issues. In Chapter 3, we design a edge-native task scheduling system to optimize the performance of DNN training an inference tasks by jointly considering the underlying edge resource status and DNN task characteristics. We then study the training task scheduling and inference task scheduling in Chapter 4 and Chapter 5, respectively. More specifically, we propose EdgeSplit, a novel FL training framework to accelerate FL on heterogeneous and resource-constraint edge devices without compromising the accuracy in Chapter 4. In Chapter 5, we designed a distributed and collaborative edge intelligence (DCEI) approach to enable geo-distributed edge devices collaborate sharing data and computation resources to perform computation-intensive video analytics tasks. The solutions presented in this thesis servers as a preliminary step towards ubiquitous intelligence and attract extensive attention from both the academia and industries. We believe AI capabilities are embedded in every edge and IoT device and are accessible everywhere.

6.2 Future Research

We close this thesis by providing some suggestions for future research. We vision that the following two directions are worth further exploration for achieving high-performance deep learning in edge computing.

- **Decentralized scheduling framework.** Traditional resource management architecture usually adopts a centralized method, where there is a centralized manager to sense the resource information of distributed client nodes and make decisions of task partition and scheduling. The edge-native task scheduling framework proposed in Chapter 3 is also based on centralized architecture. However, considering the fact that the edge nodes and cloud servers may belong to different stakeholders, it may be impossible to adopt centralized scheduling.

Hence, a decentralized architecture is needed, where a stakeholder centralized make task scheduling decisions among its belonging nodes, and a decentralized scheduling methods is adopted across stakeholders.

- **General programming model for efficient neural network partition.** In Chapter 3 and Chapter 5, we do partition the applications empowered by AI models. However, we do not partition the model itself. We partition the application DAG and pipeline respectively, where each module composed of DNN model is allocated to an edge node. In Chapter 4, we do partition the DNN model. However, the implementation is based on CNNs, which have layered neural network architecture. For RNN [96] and more sophisticated transformer model [106], they have more complex architecture, which usually cannot easily be partitioned.

  Current programming tools, such as Pytorch and Tensorflow, support manually partitioning the DNN models, e.g., specifying the partition points when programming the model. However, they do not support dynamically partition the model at runtime. For example, partitioning a model and distributing it to distributed edge nodes based on resources and communication conditions so that the training and inference performance can be optimized. We will seek more general methods to do the splitting. One possible direction is computational graph partition. In fact, current deep learning libraries, such as Pytorch and Tensorflow, will transform the neural networks into graphs during execution. Partition of the transformed graph is a promising solution [82].

References

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in Neural Information Processing Systems*, 30:1709–1720, 2017.

[2] Jon Almazan, Bojana Gajic, Naila Murray, and Diane Larlus. Re-id done right: towards good practices for person re-identification. *arXiv preprint arXiv:1801.05339*, 2018.

[3] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 50(10):58–67, 2017.

[4] Apurva Bedagkar-Gala and Shishir K Shah. A survey of approaches and trends in person re-identification. *Image and Vision Computing*, 32(4):270–286, 2014.

[5] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[6] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[7] K. A. Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé M Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In *SysML 2019*, 2019. To appear.

[8] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[9] Eric A Brewer. Kubernetes and the path to cloud native. In *ACM Symposium on Cloud Computing*, pages 167–167, 2015.

[10] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[11] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[12] Mingzhe Chen, Zhaohui Yang, Walid Saad, Changchuan Yin, H Vincent Poor, and Shuguang Cui. A joint learning and communications framework for federated learning over wireless networks. *IEEE Transactions on Wireless Communications*, 20(1):269–283, 2020.

[13] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Medhi Bennis. Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018, 2018.

[14] Yanming Chen, Tianbo Yang, Chao Li, and Yiwen Zhang. A binarized segmented resnet based on edge computing for re-identification. *Sensors*, 20(23):6902, 2020.

[15] Yifei Chen, Xiaolong Xu, and Weizheng Wang. Efficient web apis recommendation with privacy-preservation for mobile app development in industry 4.0. *IEEE Transactions on Industrial Informatics*, 2021.

[16] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[17] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[18] Thinh Quang Dinh, Quang Duy La, Tony QS Quek, and Hyundong Shin. Learning for computation offloading in mobile edge computing. *IEEE Transactions on Communications*, 66(12):6353–6367, 2018.

[19] Haohua Du, Linlin Chen, Jianwei Qian, Jiahui Hou, Taeho Jung, and Xiang-Yang Li. Patronus: A system for privacy-preserving cloud video surveillance. *IEEE Journal on Selected Areas in Communications*, 38(6):1252–1261, 2020.

[20] Rajdeep Dua, Vaibhav Kohli, and Santosh Kumar Konduri. *Learning Docker Networking*. Packt Publishing, 2016.

[21] Mohammed S Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. Edge computing meets millimeter-wave enabled vr: Paving the way to cutting the cord. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2018.

[22] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing*, 2019.

[23] Biyi Fang, Xiao Zeng, Faen Zhang, Hui Xu, and Mi Zhang. Flexdnn: Input-adaptive on-device deep learning for efficient mobile vision. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 84–95. IEEE, 2020.

[24] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127, 2018.

[25] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28, 2015.

[26] Philipp M Grulich and Faisal Nawab. Collaborative edge and cloud neural networks for real-time video processing. *Proceedings of the VLDB Endowment*, 11(12):2046–2049, 2018.

[27] Otkrist Gupta and Ramesh Raskar. Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116:1–8, 2018.

[28] Farzin Haddadpour, Mohammad Mahdi Kamani, Aryan Mokhtari, and Mehrdad Mahdavi. Federated learning with compression: Unified analysis and sharp guarantees. In *International Conference on Artificial Intelligence and Statistics*, pages 2350–2358. PMLR, 2021.

[29] Jenny Hamer, Mehryar Mohri, and Ananda Theertha Suresh. Fedboost: A communication-efficient algorithm for federated learning. In *International Conference on Machine Learning*, pages 3973–3983. PMLR, 2020.

[30] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[31] Yiwen Han, Shihao Shen, Xiaofei Wang, Shiqiang Wang, and Victor CM Leung. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. In *IEEE Conference on Computer Communications*, pages 1–10, 2021.

[32] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[34] Alexander Hermans, Lucas Beyer, and Bastian Leibe. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737*, 2017.

[35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[36] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[37] 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.

[38] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[39] Liang Huang, Suzhi Bi, and Ying-Jun Angela Zhang. Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks. *IEEE Transactions on Mobile Computing*, 19(11):2581–2593, 2019.

[40] Bert Hubert et al. Linux advanced routing & traffic control howto. *Netherlabs BV*, 1:99–107, 2002.

[41] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131, 2018.

[42] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[43] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[44] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14, 2019.

[45] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *IEEE/ACM Symposium on Edge Computing (SEC)*, pages 110–124, 2020.

[46] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 253–266, 2018.

[47] Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A Chien. Networked cameras are the new big data clusters. In *The 3rd Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 1–7, 2019.

[48] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[49] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[50] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[51] Vojdan Kjorveziroski and Sonja Filiposka. Kubernetes distributions for the edge: serverless performance evaluation. *The Journal of Supercomputing*, pages 1–28, 2022.

[52] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[53] Jakub Konečnỳ and Peter Richtárik. Randomized distributed mean estimation: Accuracy vs. communication. *Frontiers in Applied Mathematics and Statistics*, 4:62, 2018.

[54] Ondrej Krajsa and Lucie Fojtova. Rtt measurement and its dependence on the real geographical distance. In *2011 34th International Conference on Telecommunications and Signal Processing (TSP)*, pages 231–234. IEEE, 2011.

[55] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic performance prediction for smartphone applications. In *USENIX Annual Technical Conference*, pages 297–308, 2013.

[56] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[57] 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.

[58] En Li, Zhi Zhou, and Xu Chen. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, pages 31–36, 2018.

[59] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1):96–101, 2018.

[60] Hongshan Li, Chenghao Hu, Jingyan Jiang, Zhi Wang, Yonggang Wen, and Wenwu Zhu. Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution. In *2018 IEEE 24th international conference on parallel and distributed systems (ICPADS)*, pages 671–678. IEEE, 2018.

[61] Wei Yang Bryan Lim, Nguyen Cong Luong, Dinh Thai Hoang, Yutao Jiao, Ying-Chang Liang, Qiang Yang, Dusit Niyato, and Chunyan Miao. Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3):2031–2063, 2020.

[62] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 2022.

[63] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[64] Heting Liu, Fang He, and Guohong Cao. Communication-efficient federated learning for heterogeneous edge devices based on adaptive gradient quantization. *arXiv preprint arXiv:2212.08272*, 2022.

[65] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[66] 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.

[67] Wei Liu, Li Chen, Yunfei Chen, and Wenyi Zhang. Accelerating federated learning via momentum gradient descent. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1754–1766, 2020.

[68] Xinchen Liu, Wu Liu, Huadong Ma, and Huiyuan Fu. Large-scale vehicle re-identification in urban surveillance videos. In *IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, 2016.

[69] Yujiong Liu, Shangguang Wang, Qinglin Zhao, Shiyu Du, Ao Zhou, Xiao Ma, and Fangchun Yang. Dependency-aware task scheduling in vehicular edge computing. *IEEE Internet of Things Journal*, 7(6):4961–4971, 2020.

[70] Guodong Long, Yue Tan, Jing Jiang, and Chengqi Zhang. Federated learning for open banking. In *Federated Learning: Privacy and Incentive*, pages 240–254. Springer, 2020.

[71] Marko Luksa. *Kubernetes in action*. Simon and Schuster, 2017.

[72] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.

[73] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[74] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *IEEE Conference on Computer Communications*, pages 2287–2295, 2019.

[75] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[76] Rashmika Nawaratne, Damminda Alahakoon, Daswin De Silva, and Xinghuo Yu. Spatiotemporal anomaly detection using deep learning for real-time video surveillance. *IEEE Transactions on Industrial Informatics*, 16(1):393–402, 2019.

[77] Christopher Neff, Matías Mendieta, Shrey Mohan, Mohammadreza Baharani, Samuel Rogers, and Hamed Tabkhi. Revamp^{2}t: Real-time edge video analytics for multicamera privacy-aware pedestrian tracking. *IEEE Internet of Things Journal*, 7(4):2591–2602, 2019.

[78] Zhaolong Ning, Xiangjie Kong, Feng Xia, Weigang Hou, and Xiaojie Wang. Green and sustainable cloud of things: Enabling collaborative edge computing. *IEEE Communications Magazine*, 57(1):72–78, 2018.

[79] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE international conference on communications (ICC)*, pages 1–7. IEEE, 2019.

[80] Thanh-Phuong Pham, Juan J Durillo, and Thomas Fahringer. Predicting workflow task execution time in the cloud using a two-stage machine learning approach. *IEEE Transactions on Cloud Computing*, 8(1):256–268, 2017.

[81] 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.

[82] Fareed Qararyah, Mohamed Wahib, Doğa Dikbayır, Mehmet Esat Belviranli, and Didem Unat. A computational-graph partitioning method for training memory-constrained dnns. *Parallel computing*, 104:102792, 2021.

[83] Zhihao Qu, Song Guo, Haozhao Wang, Baoliu Ye, Yi Wang, Albert Zomaya, and Bin Tang. Partial synchronization to accelerate federated learning over relay-assisted edge networks. *IEEE Transactions on Mobile Computing*, 2021.

[84] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[85] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[86] Jinke Ren, Guanding Yu, and Guangyao Ding. Accelerating dnn training in wireless federated edge learning systems. *IEEE Journal on Selected Areas in Communications*, 39(1):219–232, 2020.

[87] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletari, Holger R Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N Galtier, Bennett A Landman, Klaus

Maier-Hein, et al. The future of digital health with federated learning. *NPJ digital medicine*, 3(1):119, 2020.

[88] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 159:161–174, 2020.

[89] Yuvraj Sahni, Jiannong Cao, and Lei Yang. Data-aware task allocation for achieving low latency in collaborative edge computing. *IEEE Internet of Things Journal*, 6(2):3512–3524, 2018.

[90] Yuvraj Sahni, Jiannong Cao, Lei Yang, and Yusheng Ji. Multi-hop multi-task partial computation offloading in collaborative edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1133–1145, 2020.

[91] Yuvraj Sahni, Jiannong Cao, Shigeng Zhang, and Lei Yang. Edge mesh: A new paradigm to enable distributed intelligence in internet of things. *IEEE Access*, 5:16441–16458, 2017.

[92] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[93] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[94] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[95] Ragini Sharma, Saman Biookaghazadeh, Baoxin Li, and Ming Zhao. Are existing knowledge transfer techniques effective for deep learning with edge devices?

In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 42–49. IEEE, 2018.

[96] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[97] 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.

[98] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[99] Fabrizio Soppelsa and Chanwit Kaewkasi. *Native docker clustering with swarm.* Packt Publishing, 2016.

[100] Sowndarya Sundar and Ben Liang. Offloading dependent tasks with communication delay and deadline constraint. In *IEEE Conference on Computer Communications*, pages 37–45, 2018.

[101] Hanlin Tang, Chen Yu, Xiangru Lian, Tong Zhang, and Ji Liu. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International Conference on Machine Learning*, pages 6155–6165. PMLR, 2019.

[102] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[103] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE*

37th International Conference on Distributed Computing Systems (ICDCS), pages 328–339. IEEE, 2017.

[104] Chandra Thapa, Pathum Chamikara Mahawaga Arachchige, Seyit Camtepe, and Lichao Sun. Splitfed: When federated learning meets split learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8485–8493, 2022.

[105] Chandra Thapa, Mahawaga Arachchige Pathum Chamikara, and Seyit Camtepe. Splitfed: When federated learning meets split learning. *arXiv preprint arXiv:2004.12088*, 2020.

[106] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[107] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564*, 2018.

[108] Jin Wang, Jia Hu, Geyong Min, Wenhan Zhan, Albert Zomaya, and Nektarios Georgalas. Dependent task offloading for edge computing based on deep reinforcement learning. *IEEE Transactions on Computers*, 2021.

[109] 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.

[110] Zhiyuan Wang, Hongli Xu, Jianchun Liu, Yang Xu, He Huang, and Yangming Zhao. Accelerating federated learning with cluster construction and hierarchical aggregation. *IEEE Transactions on Mobile Computing*, 2022.

[111] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. *arXiv preprint arXiv:1710.09854*, 2017.

[112] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE Conference on Computer Communications*, pages 1–9, 2021.

[113] Zifeng Wu, Chunhua Shen, and Anton Van Den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90:119–133, 2019.

[114] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *IEEE/ACM Symposium on Edge Computing*, pages 373–377, 2018.

[115] Zhuangdi Xu, Sayan Sinha, Shah Harshil S, and Umakishore Ramachandran. Space-time vehicle tracking at the edge of the network. In *The 3rd Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 15–20, 2019.

[116] Lei Yang, Yingqi Gan, Jiannong Cao, and Zhenyu Wang. Optimizing aggregation frequency for hierarchical model training in heterogeneous edge computing. *IEEE Transactions on Mobile Computing*, 2022.

[117] Lei Yang, Yanyan Lu, Jiannong Cao, Jiaming Huang, and Mingjin Zhang. E-tree learning: A novel decentralized model learning framework for edge ai. *IEEE Internet of Things Journal*, 8(14):11290–11304, 2021.

[118] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[119] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[120] Xiao Zeng, Biyi Fang, Haichen Shen, and Mi Zhang. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *ACM Conference on Embedded Networked Sensor Systems*, pages 409–421, 2020.

[121] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 377–392, 2017.

[122] Jiawei Zhang, Xiaochen Zhou, Tianyi Ge, Xudong Wang, and Taewon Hwang. Joint task scheduling and containerizing for efficient edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(8):2086–2100, 2021.

[123] Jiwei Zhang, Md Zakirul Alam Bhuiyan, Xu Yang, Amit Kumar Singh, D Frank Hsu, and Entao Luo. Trustworthy target tracking with collaborative deep reinforcement learning in edgeai-aided iot. *IEEE Transactions on Industrial Informatics*, 18(2):1301–1309, 2021.

[124] Jiwei Zhang, Md Zakirul Alam Bhuiyan, Xu Yang, Tian Wang, Xuesong Xu, Thaier Hayajneh, and Faiza Khan. Anticoncealer: Reliable detection of adversary concealed behaviors in edgeai assisted iot. *IEEE Internet of Things Journal*, 2021.

[125] Miao Zhang, Fangxin Wang, Yifei Zhu, Jiangchuan Liu, and Zhi Wang. Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 80–93, 2021.

[126] Mingjin Zhang, Jiannong Cao, Yuvraj Sahni, Qianyi Chen, Shan Jiang, and Tao Wu. Eaas: A service-oriented edge computing framework towards distributed intelligence. *arXiv preprint arXiv:2209.06613*, 2022.

[127] Mingjin Zhang, Jiannong Cao, Yuvraj Sahni, Qianyi Chen, Shan Jiang, and Lei Yang. Blockchain-based collaborative edge intelligence for trustworthy and real-time video surveillance. *IEEE Transactions on Industrial Informatics*, 2022.

[128] Qingyang Zhang, Hui Sun, Xiaopei Wu, and Hong Zhong. Edge video analytics for public safety: A review. *Proceedings of the IEEE*, 107(8):1675–1696, 2019.

[129] Tianzhu Zhang, Si Liu, Changsheng Xu, and Hanqing Lu. Mining semantic context information for intelligent video surveillance of traffic scenes. *IEEE Transactions on Industrial Informatics*, 9(1):149–160, 2012.

[130] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. Towards efficient edge cloud augmentation for virtual reality mmogs. In *ACM/IEEE Symposium on Edge Computing*, pages 1–14, 2017.

[131] Wuyang Zhang, Zhezhi He, Luyang Liu, Zhenhua Jia, Yunxin Liu, Marco Gruteser, Dipankar Raychaudhuri, and Yanyong Zhang. Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 201–214, 2021.

[132] Zhihe Zhao, Zhehao Jiang, Neiwen Ling, Xian Shuai, and Guoliang Xing. Ecrt: an edge computing system for real-time image-based object tracking. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 394–395, 2018.

[133] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. Edgeml: An automl framework for real-time deep learning on the edge. In *Proceedings of the Inter-*

national Conference on Internet-of-Things Design and Implementation, pages 133–144, 2021.

[134] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[135] 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. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.