Multi-Agent Reinforcement Learning for Cooperative Edge Cloud Computing

Shiyao Ding
Department of Social Informatics
Kyoto University, Japan

Multi-Agent Reinforcement Learning for Cooperative Edge Cloud Computing

Doctoral Thesis Series of
Ito Laboratory
Department of Social Informatics
Kyoto University

Copyright © 2022 Shiyao Ding

Abstract

The first general electronic computer in the world, ENIAC, was birthed in 1946 for calculating ballistic trajectories. Since ENIAC was able to process around 5000 operations per second, it could execute the ballistic calculation within hours, much faster than the several months needed by hand-cranked mechanical computers. This can be regarded as the first classical example of processors being developed to suit task requirements. With the rapid development of information technologies, many computing paradigms such as distributed computing, grid computing, cloud computing, and edge computing are being comprehensively developed to satisfy various emerging task requirements. Though the task requirements are ever-changing, the basic logic between server and task does not change. That is the server provides computation resources that can satisfy the various task requirements. Once the current computing paradigm fails to satisfy the new generation of tasks, a new generation computing paradigm will be needed.

In recent years, with advent of Internet of Things (IoT), edge cloud computing has become a fundamental component for the provision of sufficient computation resources to various IoT services. In order to achieve those services, a series of appropriate tasks must be allocated to the edge cloud

servers to resolve the task allocation problem. Most studies consider this problem within the context of self-interested servers, where each server desires to maximize its own interest. However, with the development of smart communities such as smart hospitals, smart campuses and smart factories, the servers usually belong to one organization and the need is to optimize the team's performance over the whole edge cloud computing system, rather than one single server.

The above scenario naturally corresponds to a cooperation relationship of servers making the traditional assumption of self-interested servers inappropriate. Our solution is to propose a cooperative edge cloud computing which is defined as follows. Given an edge cloud computing system with several edge and cloud servers and a certain period, the goal is to elucidate the task allocation policy that maximizes the team interest over the whole system within the period stipulated, while minimizing the performance metrics such as total delay/energy cost of all servers.

The goal of this thesis is to realize a series of task allocation methods that offer different attributes for cooperative edge cloud computing. Each server can observe its own status and make task allocation decisions based on the observations; these decision-makings will interact with each other. Thus, we adopt the multiagent reinforcement learning approach by regarding the servers as agents that can optimally realize task allocation. Once the agents learn well under the given edge cloud computing environment, they can be applied to guide the servers to optimal cooperation.

In this thesis, we address the following issues in studying the task allocation problem in cooperative edge cloud computing by multiagent reinforcement learning. 1) Tasks with high-workloads usually cannot be performed well

by just one single server. 2) The tasks have a dependency relationship where some tasks can only be performed after other tasks have been completed. 3) The edge servers are usually distributed among multiple areas and share one cloud server cluster.

To handle these issues, this thesis proposes three major contributions as follows.

1. Against the high-workload task allocation problem in cooperative edge cloud computing, we propose a coalitional reinforcement learning algorithm for guiding servers to dynamically form coalitions to perform tasks cooperatively. Although some traditional methods tackle the high-workload problem in distributed computing, they more focus on the tasks themselves and ignore server status. That is because the servers usually have enough computation resources that server status has only a small effect on task performance. However, edge servers usually have limited resources such that server status significantly impacts the performance of the tasks. Thus, our proposed methods can jointly perform high-workload tasks considering the dynamic features of both tasks and servers. Although this thesis applies some specific reinforcement learning (RL) methods like Q-learning in the proposals made, many other RL methods can be used to adapt different situations in edge cloud computing. This means our proposal provides a new direction in tackling high-workload task allocation in cooperative edge cloud computing by identifying a framework for server coalition formation.

2. Besides the high-workload issue, the tasks usually have a dependency relationship where some tasks can only be executed after finishing

some other tasks. Thus, we consider the dependent task allocation problem and our solution is to propose graph convolutional network based reinforcement learning methods for two cases: 1) one single job can be allocated to only one server; 2) multiple jobs can be allocated to multiple servers. In contrast to existing studies where the dependent tasks are allocated in a static process, we consider their deployment in a dynamic environment where the arriving tasks and server statuses change dynamically. Specifically, the graph convolutional network is used to embed the dependency information of the tasks, and an RL module is used to process decision-making for task allocation. We consider that our proposals are indispensable for developing models that consider the dynamic situation and task dependency in realizing advanced task allocation in edge cloud computing.

3. Against the distributed task allocation problem, unlike most existing work with the self-interested setting, we consider a cooperative setting in this thesis. Specifically, we consider this problem in two major cases: task offloading, where multiple tasks exist each of which can be performed by one server, and federated learning where only one single task exists that cannot be performed by one server. Since each edge server can only observe its own local status, it is difficult to achieve server cooperation given the incomplete information. To overcome this difficulty, we propose a multiagent reinforcement learning based task allocation algorithm that can guide the edge servers towards cooperation. We validate our approach by using real data and the results show the effectiveness of server cooperation among multiple areas even under partial observations.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Associate Professor Donghui Lin. This PhD thesis cannot be accomplished without his remarkable guidance, valuable advice and continuous encouragement. During the past three years, his philosophy of doing those researches that are fundamental in both research communities and application, influenced me a lot. This philosophy usually inspires me to think whether one idea is worth to be researched especially when I find some ideas that have not been studied in the existing work and could lead to some publications. I believe his philosophy will influence me throughout my academic career. Also, I have learned a lot from him about critical thinking, presenting research results, applying research funding and technical writing.

I also gratefully appreciate my advisory committee members, Professor Masatoshi Yoshikawa and Professor Takayuki Kanda. Without their constructive suggestions and valuable comments, the three research topics of my PhD thesis cannot go smoothly during the past three years. Moreover, their research philosophies always keep me to think how important and impactful my research is in both research and application fields.

I also gratefully appreciate Professor Takayuki Ito. His supportive advice and insightful comments improved the quality of my research a lot. His philosophy of thinking the future of State-of-the-Art rather than just following the State-of-the-Art awakened me a lot. In addition, I would like to thank Assistant Professor Rafik Hadfi, Assistant Professor Ryuta Arisaka and Assistant Professor Shun Okuhara for their fruitful discussion, and other members in Ito laboratory for their kind help.

I am also grateful to Dr. Hideki Aoyama at Panasonic Holdings Corporation, my mentor during my internship at Panasonic from September 2020 to March 2021, for his amazing technical instructions and giving the chance of conducting a joint research to continue the project until now.

Above all, I would like to thank to Professor Toshimitsu Ushio, Graduate School of Engineering Science, Osaka University, my master's degree supervisor, who provided me the valuable opportunity to study in Japan. His rigorous guidance laid the theoretical foundation of my PhD research.

Finally, I want to express my gratitude to my parents, Jihong Ding and Lingling Wang for their unconditional love and encouragement. The family is my most powerful backing. I am blessed to have you.

Contents

Abstract i

Acknowledgements v

1 Introduction 1

1.1 Overview . . . 1

1.2 Objectives . . . 3

1.3 Issues and Approaches . . . 4

1.4 Thesis Outline . . . 6

2 Background 8

2.1 Edge Cloud Computing . . . 9

2.2 Cooperative Edge Cloud Computing . . . 15

2.3 Task Allocation Problem . . . 16

2.4 Related Work . . . 23

3 Coalition Formation for High-workload Task Allocation 29

3.1 Introduction . . . 29

3.2 An Illustrative Example of High-workload Task Allocation . 31

3.3 Coalitional Markov Decision Process (CMDP) 33
3.3.1 Coalition Structure Generation 33
3.3.2 Markov Decision Process 34
3.3.3 CMDP Model 35
3.4 Dynamic Coalition Formation Algorithms 42
3.4.1 Coalitional Q-learning 42
3.4.2 Deep Coalitional Q-learning 44
3.5 Evaluation 46
3.6 Summary 54

4 Graph Convolutional Reinforcement Learning for Dependent Task Allocation 55
4.1 Introduction 55
4.2 An Illustrative Example of Dependent Task Allocation . . . 58
4.3 Dependent Task Allocation Problem 59
4.4 Graph Convolutional Reinforcement Learning Algorithms . 62
4.4.1 Case 1: Task Allocation with Single Job 62
4.4.2 Case 2: Task Allocation with Multiple Jobs 67
4.5 Evaluation 73
4.6 Summary 81

5 Multiagent Reinforcement Learning (MARL) for Distributed Task Allocation 83
5.1 Introduction 83
5.2 An Illustrative Example of Distributed Task Allocation . . . 85
5.3 Distributed Task Allocation Problem 87
5.4 Problem Formulation 99

5.5 MARL based Task Allocation Algorithms 102
5.6 Evaluation 108
5.7 Summary 126

6 Conclusion **128**
6.1 Contributions 128
6.2 Discussion 130
6.3 Future Directions 133

Publications **135**

Bibliography **138**

List of Tables

2.1 Task Definitions in Different Fields 21

List of Figures

2.1 The end-cloud architecture of video analytic. 12
2.2 The end-edge-cloud architecture of video analytic. 13

3.1 Dynamic coalition formation in an edge computing system. . 31
3.2 The dynamic transition process in CMDP. 37
3.3 The definition of optimal weighted state-action value function. 43
3.4 Deep coalitional Q-learning algorithm. 45
3.5 Comparing the performances of deep coalitional Q-learning with coalitional Q-learning. 52
3.6 Comparing the final learning results of deep coalitional Q-learning with coalitional Q-learning. 53

4.1 Dependent task allocation in edge computing. 59
4.2 Graph convolutional reinforcement learning algorithm. . . . 63
4.3 Multi-graph convolutional reinforcement learning algorithm. 69
4.4 Comparing performances of GCRL, QL and DQN algorithms in minimizing the sum of energy and delay costs. . . 74
4.5 The impacts of task workload, task dependency number and task RAM. 75

4.6 Comparing the performances in minimizing delay cost via MGCRL, DTO, TDQ and TQL algorithms. 79
4.7 Parameter analysis of task number and server number. 80

5.1 A distributed edge cloud computing system in a smart hospital. 86
5.2 Task offloading in a distributed edge cloud computing system. 88
5.3 Decentralized federated learning process. 98
5.4 Value decomposition network based task allocation algorithm in distributed edge cloud computing. 103
5.5 The detail of GRU module 107
5.6 The detail of Q^i neural network structure. 112
5.7 Comparing performances of VDN-TO with IDQL-TO and random policy in latency-sensitive case. 113
5.8 Comparing performances of VDN-TO with IDQL-TO and random policy in energy-sensitive case. 115
5.9 Comparing performances of VDN-TO with IDQL-TO and random policy in balance case. 116
5.10 Comparing performances of VDN-DFL on MNIST dataset in (a) reward, (b) processing time, (c) accuracy and (d) communication cost. 119
5.11 Comparing performances of VDN-DFL on CIFAR-10 dataset in (a) reward, (b) processing time, (c) accuracy and (d) communication cost. 120
5.12 Comparing performances of VDN-DFL on FashionMNIST dataset in (a) reward, (b) processing time, (c) accuracy and (d) communication cost. 121

5.13 Comparing performances of VDN-DFL with baselines on MNIST dataset in (a) reward, (b) accuracy , (c) processing time and (d)communication cost. 123
5.14 Comparing performances of VDN-DFL with baselines on CIFAR-10 dataset in (a) reward, (b) accuracy , (c) processing time and (d)communication cost. 124
5.15 Comparing performances of VDN-DFL with baselines on FashionMNIST dataset in (a) reward, (b) accuracy, (c) processing time and (d) communication cost. 125

Chapter 1

Introduction

1.1 Overview

Since Internet of Things (IoT) devices such as smart watches, smart phones are exponentially increasing, an enormous volume of data will be generated that must be processed promptly [Pan and McElhannon, 2017]. However, IoT devices usually have scant computation resources, which creates reliance on external computing resources [Chang et al., 2019]. Cloud computing is a classical solution that can provide abundant customizable computation resources, its effectiveness has been verified and many cloud computing providers such as AWS, Google and Azure now offer various cloud computing services. Edge computing, as a supplement to cloud computing, can offer computing services with lower latency and lower energy than cloud computing as its servers are closer to the users. However, the computation resources of edge servers are not as rich as those of cloud servers. Therefore, edge cloud computing, which combines the advantages of edge

and cloud computing is seen as the desirable computing platform for IoT [Chang et al., 2014].

A fundamental problem with edge cloud computing is how to allocate tasks to the various servers so as to minimize various costs while satisfying the task requirements. This is called the task allocation problem in edge cloud computing. Although many studies have tackled this problem, they most often assume a self-interested edge cloud computing environment, where each edge/cloud server tries to maximize its own interests. Accordingly, they usually formulate the problem as a zero-sum game and aim to solve the Nash equilibrium strategy where each server develops its best response given the strategies of the other server.

With the strong development of IoT devices and services, more and more companies are starting to provide smart communities like Aliyun's city brain. In those scenarios, the servers are usually owned by an organization rather than a single user [Donovan et al., 2017][Nishi, 2018]. The goal is to optimize the overall performance of the edge cloud computing systems rather than each server's own interests. Thus, how to make edge and cloud servers cooperate with each other to perform tasks well requires detailed studies. In this thesis, we consider cooperative edge cloud computing, a new edge cloud computing framework where each edge/cloud server tries to maximize team rewards like total delay. Thus, this study addresses a new task allocation problem in cooperative edge cloud computing. We analyze three major classical cases of cooperative edge cloud computing and propose several novel methods to cope with the issues raised. Evaluations based on several real datasets show the effectiveness of our proposed methods.

1.2 Objectives

As stated in the above section, cooperative edge cloud computing is required to support various IoT environments and its fundamental problem is task allocation. To elucidate this problem, we indicate three major cases requiring cooperation of servers for task allocation.

High-workload Task Allocation

First, **high-workload** tasks usually cannot be performed well by just one server; adequate performance is assured only if the task can be cooperatively performed by several servers.

Dependent Task Allocation

Second, the tasks are inter-dependent [Tang et al., 2020] where some tasks can only be performed after the other tasks have been completed; it corresponds to a **dependence** feature. Thus, one server needs to know the status of its own tasks' dependent tasks allocated to other servers.

Distributed Task Allocation

Third, the edge servers are usually **distributed** across various areas and share one cloud server cluster [Jošilo and Dán, 2018]. Thus they must cooperate with each other, since one server's performance does not depend on just its own actions, but can be influenced by the other servers' actions.

Thus, the objective of this thesis is to realize a series of efficient task allocation methods for cooperative edge cloud computing, with particular emphasis on the above three cases.

1.3 Issues and Approaches

In order to achieve the above objective, the issues of the above three cases and corresponding approaches are stated as follows.

1. In edge cloud computing, allocating the tasks with high-workload while satisfying task requirements (e.g., response time and required memory space) is an important problem. Since the tasks with high workload usually cannot be performed well by only one single server, performance would be better if the task can be performed on several servers. The existing researches are usually based on human-design rules like MapReduce, which is a programming model, and associated implementation by a parallel, distributed algorithm for processing and generating big data sets on a cluster [Dean and Ghemawat, 2008]. However, they consider just the information of tasks and so ignore the current status of servers. This may yield poor performance in edge cloud computing since the cost of performing the tasks strongly depends on server status, which is dynamically altered by the popping/pushing of tasks. In this topic, we study a dynamic task allocation problem in edge cloud computing where both server status and arriving tasks change over time; the goal is to identify the task allocation policy that can minimize user cost. Specifically, we consider a parallel processing case where a task's workload can be infinitely divided among the servers available; this causes a huge solution space which makes the problem intractable. Our solution is to consider an approximate method from the perspective of server coalitions rather than tasks, and propose a dynamic coalition formation algorithm to guide several edge servers into forming a coalition dynamically. Eval-

uations verify that our algorithm can significantly reduce user cost compared with some other existing algorithms.

2. Beside the feature of high-workload tasks mentioned in above, in the IoT environment, there usually exists a dependency relationship among the tasks: some tasks can be performed only after accomplishing some other specific tasks. Then, one server needs to know the situation of its own tasks' dependent tasks allocated on other servers, which corresponds to cooperation among servers. This problem poses two challenges: how to cope with dependency information for decision-making of task allocation and how to cope with the dynamics whereby server status and arriving tasks change dynamically. To solve these challenges, we propose a novel algorithm based on graph convolutional reinforcement learning for dependent task allocation: it can deal with the dependency and dynamic issues of the problem effectively. Specifically, we represent the dependent tasks as directed acyclic graphs and employ a graph convolutional network to embed the dependency information of the tasks. Then, we formulate the task allocation problem as a Markov decision process and use deep reinforcement learning to cope with the dynamics. Experiments verify that our algorithm offers significantly better performance than the existing algorithms examined.

3. In the above two parts, we focus on a centralized edge cloud computing system. However, edge servers can be allocated to various areas and each edge server can offload its tasks to the remote cloud servers, which is called distributed task allocation in edge cloud computing. Most of the existing work on distributed task allocation as-

sume that each self-interested user owns one edge server and chooses whether to execute the tasks locally or offload them to cloud servers. The goal of each edge server is to maximize its own metric of interest like response speeds, which corresponds to a non-cooperative setting. However, in this topic, we consider this problem in a cooperative setting and formulate it as a decentralized partially observable Markov decision process (Dec-POMDP) as it can well model the dynamic features. Then, we apply a multiagent reinforcement learning algorithm called value decomposition network (VDN) and propose a VDN based task allocation algorithm to solve the Dec-POMDP. Finally, we choose part of a real dataset to evaluate our algorithm and show its effectiveness in a comparison with some other methods.

1.4 Thesis Outline

This thesis is organized into six chapters including Chapter 1. The contents of the remaining chapters are summarized as follows. Chapter 2 presents the background of this thesis, an illustrative example of task allocation in edge computing and existing methods for solving edge cloud computing. Two major approaches will be discussed: one-step objective type and multiple-steps objective type. Chapter 3 studies the high-workload task allocation problem in cooperative edge cloud computing with consideration of the problem's dynamic features. The goal is to identify an optimal policy that can guide the edge servers in forming coalitions to perform high-workload tasks cooperatively. In Chapter 4, we study the dependent task allocation problem and propose a novel algorithm based on graph convolutional reinforcement learning (GCRL) for dependent task allocation: it can deal with

the dependency and dynamic issues of the problem effectively. In Chapter 5, we consider the distributed task allocation problem and formulate it as a decentralized partially observable Markov decision process (Dec-POMDP) which is a classical model for discrete-time decision problems characterized by partial observations. Although traditional RL algorithms like deep Q-network (DQN) can also be applied to solve Dec-POMDP by making each agent maintain a DQN, it is difficult to achieve cooperation. Thus, based on the cooperative multiagent reinforcement learning (MARL) algorithm called the value decomposition network, we propose a cooperative task allocation algorithm. Finally, Chapter 6 concludes this thesis by discussing the summary of contributions and future work.

Chapter 2

Background

In this chapter, we will discuss the background of edge cloud computing and introduce one of its fundamental problems: task allocation. In order to implement various IoT services in edge cloud computing, the essence is to perform their corresponding tasks. For instance, in order to achieve a body-monitoring IoT service, the tasks of collecting body data, analyzing body data and visualizing the results must be performed in the order given. Moreover, these tasks must be performed while optimizing some objectives such as minimizing delay cost and energy cost. In this chapter, we first introduce edge cloud computing, then task allocation problem and existing methods. Finally, we introduce our object in this thesis: task allocation in cooperative edge cloud computing.

2.1 Edge Cloud Computing

Cloud Computing

Before the emergence of cloud computing, organizations with high computation resource requirements had to construct and maintain their own server clusters. However, the rising cost and complexity of server clusters have become a significant burden for those organizations. It is also difficult to expand/upgrade the current server clusters to match increases in computation resource requirements. To solve those problems, Amazon Corporation first started to commercialize cloud computing in 2006 with the release of elastic computing (EC2). It allows the users of cloud computing to choose their preferred level of on-demand computation resources without considering any physical machine. Its several major advantages are summarized below [Pan and McElhannon, 2017].

- **Low Usage Cost** Cloud computing eliminates the various costs of purchasing hardware and software, as well as setting up and operating an on-site server cluster, including the supply of uninterrupted power for running and cooling. The user needs only rent the cloud servers, no other costs need be considered [Chang et al., 2019].

- **Elastic Extension** The amount of resources used can be changed elastically to suit the user's latest requirements, which yields excellent system flexibility. The computation resources are well supported by a wide range of functions and resources, making it easy to create enhanced functionality.

- **High Security** The cloud providers provide the security services es-

sential to protecting the user's data, applications and infrastructure from potential threats by offering a broad set of policies, technologies and controls that strengthen the user's overall security posture.

Thus, cloud computing offers high reliability for users given its abundant computation resources, and provides scalable and efficient computing services.

Edge Computing

Although cloud computing has become a fundamental way of providing computing services in various fields, cloud access can incur huge bandwidth consumption and long delays, and so may not satisfy IoT applications that need very low latency. Edge computing supplements cloud computing by providing cloud-like services closer to the IoT devices while offering several advantages that cloud computing cannot offer [Bonomi et al., 2012][Pan and McElhannon, 2017]. Its several major advantages are summarized as follows.

- **Low Delay** Uploading and downloading the data to/from cloud servers will incur delays. Such delays will become large given the increase in data size and may fail to satisfy the requirements of delay-sensitive IoT applications. However, edge servers can process the data locally without uploading to or downloading from cloud servers, which corresponds to low delay.
- **Low Bandwidth Consumption** The data volume from devices will exponentially increase with the number of devices. That means the bandwidth resources may not be sufficient if many devices upload the

data into the cloud at the same time. However, in edge computing, the end devices and edge servers occupy local area networks. Since the edge servers process most data locally and only upload little data to the cloud, thus bandwidth resources requirements are low.

- **Data Security Assurance** The rapid penetration of IoT devices has also raised a number of privacy concerns. Many counties do not allow data such as face data, body data and bank transaction data to be uploaded to cloud servers. Edge servers obviate the privacy concerns raised as they store and process the data locally.
- **Continuous Operation** If all IoT services fully rely on the cloud servers, there will be concerns with service continuity if the access network fails. The edge servers well support those services whose continuity cannot be interrupted, as they offer higher guarantees of continuous operation.

Although several advantages of edge computing have been illustrated, the edge servers are usually not as rich in capacities such as bandwidth, processing speed and memory size as cloud servers. Therefore, edge cloud computing has been proposed as a general solution and is attracting a lot of interest as it balances the advantages of edge and cloud computing [Chang et al., 2014].

Application: Real-time Video Analysis

As stated in the above section, edge cloud computing has become a fundamental component in IoT environment. In this section, we will introduce several classical applications of edge cloud computing.

Real-time video analysis, as one of classical IoT applications, has become a core component in many fields, such as smart parking, smart hospital and virtual reality (VR). Along with the rapid development of deep learning (DL), many DL models now offer much more accurate recognition than humans can achieve. Since the devices used to collect video data like cameras usually have limited resources like RAM and disk spaces, the DL models face barriers to deployment. One traditional solution is to deploy the recognition DL models on cloud servers, and the video data collected from the devices can be uploaded to the cloud server. This is called end-cloud architecture and has been used in many scenarios, as shown in Figure 2.1. Moreover, the collected data can be used to enhance the DL models for improved accuracy.

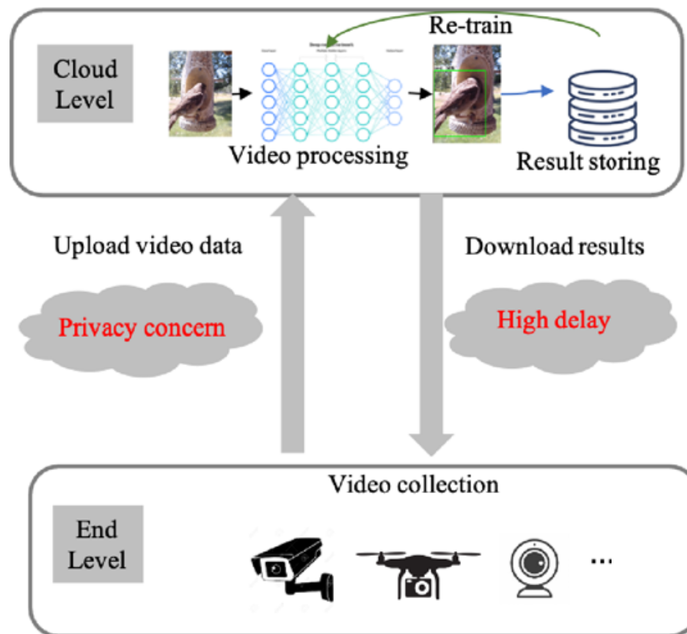


Figure 2.1: The end-cloud architecture of video analytic.

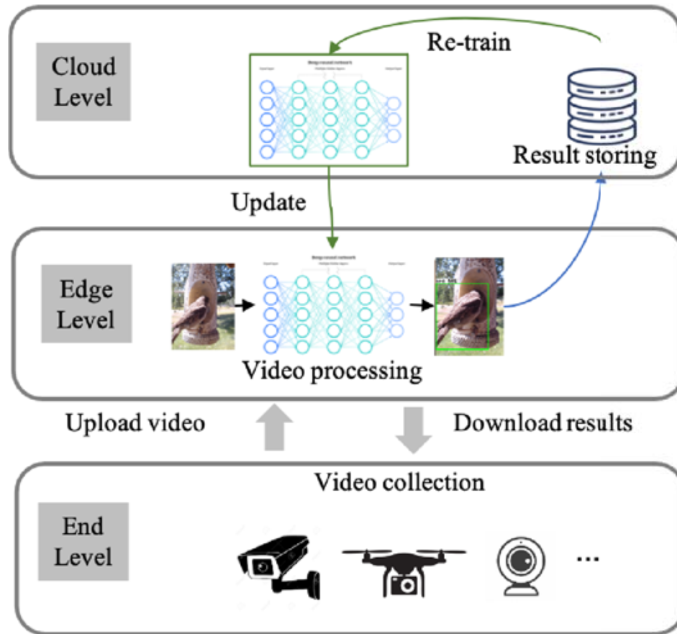


Figure 2.2: The end-edge-cloud architecture of video analytic.

However, this end-cloud architecture has shown two main issues as follows. The first is the privacy issue as videos that might show the human face cannot be uploaded and can only be processed locally. The second is the delay issue as many scenarios require real-time response. Uploading video data usually incurs unacceptably high delay.

To solve the above issues, the architecture of end-edge-cloud is proposed, as shown in Figure 2.2. At the end-level, the video data are collected from devices and then are processed through edge computing. At the edge level, the DL models are downloaded from cloud servers and used to process the input videos. The process results can be stored in the cloud servers without raising privacy concerns. Since it is not efficient to train models at the edge level, DL model re-training will be performed in the cloud servers. Then,

the latest DL model will be downloaded to edge level from the cloud. Several classical applications of real-time video analysis will be illustrated as follows.

One classical application of real-time video analysis is intelligent manufacturing (IM). As illustrated in [Li et al., 2018], uploading and processing all the video data will degrade the computing efficiency and create a barrier to the implementation of real-time inspection systems for smart industries. Such systems should also resolve the issues of response latency, risk, and privacy protection. These requirements can be satisfied by the attributes of edge computing illustrated above and many related studies have been made. For instance, DeepIns [Li et al., 2018] is a real-time inspection system enabled by edge computing that can detect defects in products. Each factory usually has multiple production lines which will generate significant volumes of data. Moreover, this data must be processed in real-time, since the production lines often run at high speed and even a latency of just a few micro-seconds might hold up production. Fortunately, allocating edge servers at the detection devices yields real-time data processing. Moreover, the detection model can be updated from the cloud servers.

Another classical application is smart forest fire detection. Unmanned aerial vehicles (UAVs) are being applied in many fields such as gas facilities for security, surveillance, emergency response and infrastructure inspection. One of the classical applications, smart forest fire detection, can substantially reduce the risk of damage caused by fires [Khan et al., 2020]. Since the UAVs have limited computation resources, they require some additional computing resources to process the data of images or video streams captured by the UAV cameras. One way is to upload the video data to remote cloud

servers for video analysis. However, it will suffer a high latency which cannot satisfy the requirement of real-time analysis, especially since the communication channels in the forest usually have low gain due to the scant communication infrastructures available. UAV-enabled edge computing offers a solution as it can rapidly process the video data to detect forest fires, without the help of cloud servers [Narang et al., 2017][Avgeris et al., 2019].

Each edge server in the above applications can independently perform its tasks and does not need to consider other edge servers. That is because the tasks have no dependent relationships and the task workloads are small enough as each edge server has enough computation resources to perform its own tasks in an independent and static way. However, given the continuous development of various task requirements and the increase in task number, one server will be insufficient to accomplish the task by itself which requires the cooperation of other servers. Thus, tasks must be performed through cooperation.

2.2 Cooperative Edge Cloud Computing

Many studies have examined the problems of edge cloud computing in depth. However, most of them considered the problem in the non-cooperative setting, where each edge server is assumed to be self-interested, and simply learns its own policy independently. As illustrated in Chapter 1, this approach fails to support the setting wherein all edge servers belong to one organization and the relationship among edge servers is cooperation. Thus, in this section, we study the new problem of cooperative edge cloud computing. We define cooperative edge cloud computing as follows. Given

an edge cloud computing system with several edge and cloud servers and a certain period such as one day or one month, the goal is to maximize the overall team interest (the whole system) in that period, by optimizing some metrics such as the total delay/energy cost of all servers.

2.3 Task Allocation Problem

Task allocation is a classical problem in computer science and has been studied for many years. Before introducing task allocation in edge cloud computing, we will introduce several traditional task allocation problems, to better position this thesis.

Task Allocation in Distributed Computing

One classical scenario of task allocation in computing science is distributed computing. In [Salman et al., 2002], they define the task allocation problem as that of assigning the tasks of a program among different processors in a distributed computer system with the goal to reduce the program turnaround time and to increase system throughput. They formulated the problem of a distributed program as a task interaction graph (TIG) where the nodes are processors in the system and the edges represent communication links between the processors. An edge weight represents the length of the shortest path between the corresponding processors. The task allocation problem is cast as identifying a function that maps the set of tasks to the set of processors. The total execution time of all tasks assigned to all processors can be calculated using a summation approach. They proposed a novel population-based heuristic method called particle swarm optimization (PSO) that draws

its models from the social behavior of bird flocks and fish schools. In order to balance exploration and exploitation, they combine local search methods (through self-experience) with global search methods (through neighboring experience).

In [Ma et al., 1982], the authors study the problem of allocating tasks among processors with considering more optimal goals in distributed computing systems. The goal is to find a task allocation that satisfies : 1) minimum inter-processor communication cost, 2) balanced utilization of each processor, and 3) all engineering application requirements. They derived an algorithm from the branch and bound (BB) method where the allocation problem is represented as a search tree.

In [Hong and Prasanna, 2004], they further consider task allocation in distributed heterogeneous computing systems. First, they formulate the computing system as a directed graph where the nodes are assumed to be connected via an arbitrary topology. Each node represents a server and the weight of a node represents its processing power. Each edge represents a network link between nodes and its weight represents the communication bandwidth of the link. The goal is to maximize the system throughput which is defined as the number of tasks computed by the system per unit of time under a steady state condition. They proposed a decentralized adaptive algorithm that yields a simple decentralized protocol for coordinating the resources of the system.

The task allocation problem in distributed computing usually considers simultaneous task allocation where the allocation is completed once all the tasks are allocated to the server at one step. This feature does not satisfy the dynamic feature of edge cloud computing where task arrival is intermittent

and irregular.

Task Allocation in Multi-Robot

Task allocation in the field of robotics is also an important topic. In [Gerkey and Matarić, 2004], they study the task allocation problem of multi-robot coordination which is called multi-robot task allocation (MRTA). The fundamental question of MRTA is determining which task should be executed by which robot in order to cooperatively achieve the global goal. The task of MRTA is defined as a subgoal to achieve a global goal of the system, and can be performed independently of other subgoals. For instance, a task can be defined as delivering a package to one designated place.

For instance, [Donald et al., 1997] discusses several fundamental measures for mobile robot cooperation to achieve complex tasks such as the internal state the robot should retain, the number of cooperating robots required, and the information that the sensors should provide. They then model the information requirements of a coordination algorithm and design a mechanism to reduce the computation overhead of the algorithm.

Moreover, MRTA can be divided into three major classes [Gerkey and Matarić, 2004].

- Task number to be executed: there are two sub-classes of single-task robots and multi-task robots. In the case of single-task robots, each robot can only perform one task at a time. In the case of multiple-tasks robots, the robots can execute multiple tasks simultaneously.
- Required robot number for task execution: there are two sub-classes of single-task robots and multi-task robots. In the case of single-task

robot, each task requires only one robot to execute. In the case of multiple-robot tasks, one task requires multiple robots to perform.

- Duration of task assignment: there are two sub-classes of instantaneous assignment and time-extended assignment. In the case of instantaneous assignment, task information is only that known when the assignment and future allocation is not considered. In the case of time-extended assignment, more information is available during task allocation such as tasks that will require to be assigned in the future.

Based on the above classification, there are six combinations in MRTA. For instance, [Gale, 1989] studies the allocation problem of single-task robots, single-robot tasks, and instantaneous assignment. It formulates this as an optimal assignment problem (OAP) whose goal is to maximize the overall system utility, which was originally studied in game theory. In [Karger et al., 1999], the study item is task allocation of single-task robots, single robot tasks, and time-extended assignment where tasks can be executed in parallel by multiple robots. However, the property of parallelism exponentially enlarges the solution space. They propose an approximation method based on a greedy algorithm to solve it.

The robot task allocation problems focus more on a macro-semantic level of tasks where the issue is how to divide a large abstract task into several sub-tasks that can be independently executed by each robot.

Task Allocation in Routing Planning

Another field where task allocation is a fundamental problem is route planning. One classical example is the vehicle routing problem (VRP) [Braek-

ers et al., 2016]. The predecessor problem of VRP, the truck dispatching problem, was first proposed in [Dantzig and Ramser, 1959] where several distributed gas stations require a certain number of identical trucks to transfer a certain amount of oil from a central hub; the goal is to identify a route with minimum traveled distance for all trucks. Then, VRP is generalized to a linear optimization problem in [Clarke and Wright, 1964]. In VRP, multiple goods are required to be assigned to multiple vehicles and each goods item has a specific destination. The objective is to achieve the most cost-effective route (e.g. minimizing moving costs of all vehicles) for all delivery destinations, given a set of moving vehicles with limited capacity [Pellazar, 1994]. The allocation problem is to decide which goods should be allocated to which vehicle. Since each vehicle usually has different capacity, different task allocation policies would yield different total costs.

VRP has been studied well and many variants have been proposed. For instance, in open VRP (OVRP) the vehicles are not forcibly returned to the central depot after visiting the last customer [Fleszar et al., 2009]. Thus, the goal of OVRP is to minimize the number of vehicles used, as well as attempting to minimize total distance. Dynamic VRP (DVRP) considers the VRP problem in a real-time/online way where user requests dynamically change [Pradenas et al., 2013]. Thus, it requires re-calculation of the optimal routes to respond to new customer requests, where the difficulty is that the information of new requests is usually unknown in advance. Another variant is time dependent VRP (TDVPR) where the travel times between depots and customers change dynamically [Ichoua et al., 2003]. This setting corresponds to the realistic scenario where vehicle moving times are altered due to congestion.

Although there are many variants of VRP, they are usually modeled as a linear optimization problem. Only the objective function and constraints change according to the specific conditions.

Task Allocation in Edge Cloud Computing

Although the tasks described above have different definitions in different fields, as shown in Table 2.1, the objects to execute the tasks usually have abundant resources and only static task allocation is considered.

Table 2.1: Task Definitions in Different Fields

Fields	Task Definitions
Distributed Computing	Assigning the tasks of a program among different processors in a distributed computer system.
Multi-Robot Cooperation	A subgoal to achieve a global goal of the system, and can be performed independently of other subgoals.
Routing Planning	Deciding which goods should be allocated to which vehicle.

However, the servers in edge cloud computing usually have limited computation resources and highly dynamic features. Thus, such kind of features should be emphasized and considered while studying the task allocation problem in edge cloud computing. Then, we illustrate the realistic meaning of task allocation in edge cloud computing as follows. Although several applications of real-time video analysis have been illustrated, it is a waste to upload all the video data all the time. Thus, event-trigger video analysis is proposed where the devices will collect or upload the data only when some events happen. For instance, in the smart parking scenario, the cameras will

upload the video data only when a car is detected.

Moreover, let us consider a scenario where multiple cameras exist and each camera is to be responsible for all camera in one area. Then, if some objects are detected, it will trigger the task of object detection. Since there are multiple cameras in one environment, several tasks can be invoked at the same time. This can be formulated as a task queue which needs to be allocated to the edge cloud computing system. The edge servers usually have limited resources and they can be influenced by tasks that have high performing costs. Thus, performance attributes like delay and energy consumption will be degraded if the tasks are not allocated well. This is taken as a task allocation problem that has been addressed in many studies.

In order to satisfy various service requirements, many metrics are needed to quantitatively evaluate system characteristics. We introduce some classical task allocation metrics in edge cloud computing as follows.

One classical metric is energy cost. In edge cloud computing, most energy is consumed by two functions: executing tasks and allocating tasks. As for executing tasks, the server's computation unit such as CPU and GPU will consume energy in performing each program execution. In particular, executing some tasks by using the DL models with a lot of parameters usually incurs very high energy cost. On the other hand, allocating a task also has a certain energy costs which is related to some parameter values such as transferring data size and signal gain.

Besides energy cost, the time taken in performing or allocating the tasks is called delay cost. There are two major components: computation delay and allocation delay. The computation delay usually has an inverse relationship

with computation speed. The allocation delay is the time needed to allocate the tasks which usually depends on network speed. In edge cloud computing, allocating tasks to the edge servers usually costs much less time than allocation to cloud servers, since the edge servers are closer to users than the cloud servers.

Since the edge and cloud servers are usually rented from cloud computing providers, usage fees must be considered. For instance, renting an on-demand cloud server on AWS will be charged according to the usage time. The cost of renting a cloud server becomes cheaper, yet it is still a significant burden in many cases. Although there are many types of metrics, this thesis focuses on energy costs and delay costs. Since some other metrics do not impose any essential changes, they can be easily added to our models. Therefore, the goal of task allocation is how to allocate the tasks with optimizing the above single/multiple metrics.

2.4 Related Work

The previous section has illustrated the task allocation problems in many fields, especially in edge cloud computing. In this section, we will introduce the existing methods for task allocation. They can be divided into two major types according to the objective of task allocation, as stated below.

One-step Objective Type

Most studies on the task allocation problem assume a single-step objective. They usually define a unity function based on one or multiple metrics as illustrated in the above section. They then formulate the task allocation

problem as a constraint optimal problem (COP) while trying to satisfy some constraints. For instance, Dinh et al. [Dinh et al., 2017] studied a task allocation problem assuming a single mobile device (MD) and multiple edge devices. The goal is to minimize both delay cost and energy consumed by MD. They considered the two cases of fixed CPU frequency and elastic CPU frequency which significantly impacts the above costs. Then, they formulated the problem as a COP and proved that the problems are NP-hard. Finally, they proposed a linear programming relaxation approach to solve this problem. Tran et al. [Tran and Pompili, 2018] focused on the communication cost in mobile edge computing (MEC). Specifically, they considered a scenario wherein each base station (BS) is equipped with one edge server. The BS employs a multi-cell wireless network to process communication traffic and many users would use this network to offload their tasks. The goal is to maximize the users' task allocation gains which are defined as the weighted sum of delay cost and energy cost incurred in performing the tasks. They formulated the problem as a mixed integer nonlinear program (MINLP) and proposed a low-complexity solution. In [Tran et al., 2017], they focused on the task caching problem where some tasks like video streaming, which usually generate large volumes of data needed to be cached. They proposed a collaborative caching method in MEC environment. Tao et al. [Tao et al., 2017] formulated an energy cost minimization problem with the constraint of resource capacity; the cost factor they focused on is the energy consumed by allocating tasks from mobile devices to edge servers. Chen et al. [Chen et al., 2018a] studied mobile edge cloud computing and adopted a game theoretic approach for minimizing allocation cost in a distributed manner. Gu et al. [Gu et al., 2015] considered a mobile edge cloud computing case and attempted to minimize the joint energy cost including uploading cost,

deployment cost and inter-base station communication cost. They formulated the problem as a mixed integer nonlinear program and linearized it into a mixed integer linear programming form. Zhang et al. [Zhang et al., 2018] focused on minimizing the energy cost consumed by the system itself and proposed cost-efficient scheduling for delay-sensitive tasks in edge computing.

Multi-steps Objective Type

Besides solving the task allocation problem with one-step objective, many studies have considered the dynamic features of edge cloud computing in optimizing an objective over multiple steps. The proposals are usually formulated as a Markov decision process (MDP) and reinforcement learning (RL), a classical dynamic programming method, is usually applied.

For instance, Li and Huang [Li and Huang, 2017] formulated a dynamic process for task allocation using the MDP approach to balance the tradeoff between energy costs and Quality of Service (QoS) requirements; the energy costs they focus on are mainly associated with sensor hub and data. To avoid immense computation when using value iteration or policy iteration RL methods, they propose an ordinal optimization approximate method. Guo et al. [Guo et al., 2016] considered a dynamic continuous time process in edge cloud computing and proposed a task allocation policy for achieving optimal power-delay tradeoff in the system. They also formulate the problem as a MDP. Then, in order to reduce the computation complexity, they studied the associated dual problem and proposed an approximate method. Chen et al. [Chen et al., 2018b] considered a task allocation problem in mobile-edge computing where it is required to decide whether to execute a

computation task on the mobile device or to offload it for MEC server execution. They proposed the deep RL-based task allocation algorithm to learn the optimal policy of task allocation.

On the other hand, the centralized single-agent RL method suffers from the dimensionality curse problem along with the scaling issues. Then, the decentralized methods like fully decentralized multiagent reinforcement learning are often applied. Multiagent reinforcement learning (MARL) can be usually divided into three categories: fully centralized, fully decentralized, and mixed centralized & decentralized. In the fully centralized setting, the multiagent problems are regarded as a single agent problem. Specifically, it takes a joint action that includes all agents' individual actions, and a global state that covers all agents' observations/local states. This method is easy to deploy in many cases and offers stable learning, since the global information is known and can be used well. However, it against the curse of dimensionality problem as the joint action space and global state space grow with the number of agents.

Another approach is the fully decentralized control method, where each agent has its own local observation and maintains a policy to independently control and execute actions. This method avoids the dimensionality curse problem, however its learning is unstable. This is because each agent only considers its own information without considering other agents, which means the actions of the other agents are regarded as a part of the environment.

In order to avoid the above problem, the mixed method called centralized training and decentralized execution (CTDE) was proposed. Specifically, each agent holds an individual policy to execute the actions locally. In or-

der to improve the stability of learning, it employees a centralized training approach where it evaluates each policy under the global state and global reward.

As for the type of multiagent system, there are two major types: cooperative setting and self-interested setting. In the cooperative setting, all agents will try to optimize one common team reward function. In the self-interested setting, each agent is assumed to be self-interested and so desires to optimize its own reward.

Most of the studies on the task allocation problem considered the problem in a self-interested setting. Chen et al. [Chen et al., 2015] considered a multi-user computation allocation problem for mobile-edge cloud computing. Each mobile device chooses a channel for offloading their computation tasks. However, the uplink data rate is slow, which degrades performance if many devices choose the same channel. They assume that the mobile devices are self-interested and compete for the limited channel resources. They formulated the problem as a theoretic game model and proposed a distributed computation allocation algorithm. Liu et al. [Liu et al., 2020b] considered an edge cloud computing network with a three-layer hierarchical architecture consisting of a cloud platform, multiple gateways, and a lot of IoT users. The gateway allows a limited number of devices to be accessed at the same time, thus each device has a non-cooperative relationship with other devices. They utilize a centralized user clustering method to group the IoT users into different clusters according to user priorities, and allocation proceeds in accordance with user priorities. Chen et al. [Chen et al., 2018b] also considered a non-cooperative environment where each end user observes its local environment to learn optimal decisions for either

local computing or edge computing with the goal of minimizing long-term system cost. They formulated it as a stochastic game and proposed a fully-decentralized learning method where each agent independently learns its own policy.

The above studies do not consider the cooperative setting, so each server is assumed to be self-interested, and simply learns its own policy in isolation. This approach can not satisfy the setting wherein all servers belong to one organization and the relationship of servers is cooperative. This thesis rectifies this omission by studying the new problem of cooperative task allocation in edge cloud computing.

Chapter 3

Coalition Formation for High-workload Task Allocation

In this chapter, we focus on the first case illustrated in Chapter 1 which is high-workload task allocation in cooperative edge cloud computing. Our solution is to propose a dynamic coalition formation method for server co-operation.

3.1 Introduction

Along with the rapid development of advanced IoT services, task workloads are becoming so large that they cannot be performed well by just one server. It triggers the high-workload task allocation problem. Although this is a classical problem in some traditional researches like distributed computing, the studies to date have focused more on the task itself and ignore server status. This is because the servers in distributed computing usually have

abundant computation resources and server statuses have only limited influence on task performance. However, the servers in edge cloud computing usually have limited computation resources where the server statuses will deterministically influence the task performance. Moreover, they consider only one-step decision and do not consider how the current task allocation will influence future server status. However, in many realistic IoT scenarios edge cloud computing should focus more on long term performance, which corresponds to a long-term goal. That means that the optimal decision-making in the current step may not be optimal for the long-term goal.

Those existing studies usually employ rule-based methods like Hadcoop where a module called MapReduce which is a programming model for processing and generating big data sets in a parallel way exists [Dhirani et al., 2017][Chen et al., 2018a][Zhang et al., 2018]. However, they consider just the information of tasks and so ignore the current status of servers. This may degrade performance in edge cloud computing since the cost of performing the tasks strongly depends on server status, which is dynamically altered by popping/pushing tasks. Moreover, most existing works rarely consider the viewpoint of user cost such as cloud service fee and edge server electricity fee. User cost has become an issue of more importance, since the cloud services offered by some cloud vendors such as Amazon, Aliyun and Azure, usually come with high user costs. Thus, tackling the tasks with high workload in edge cloud computing while considering the dynamic server statuses and starting from the viewpoint of minimizing user costs requires further studies.

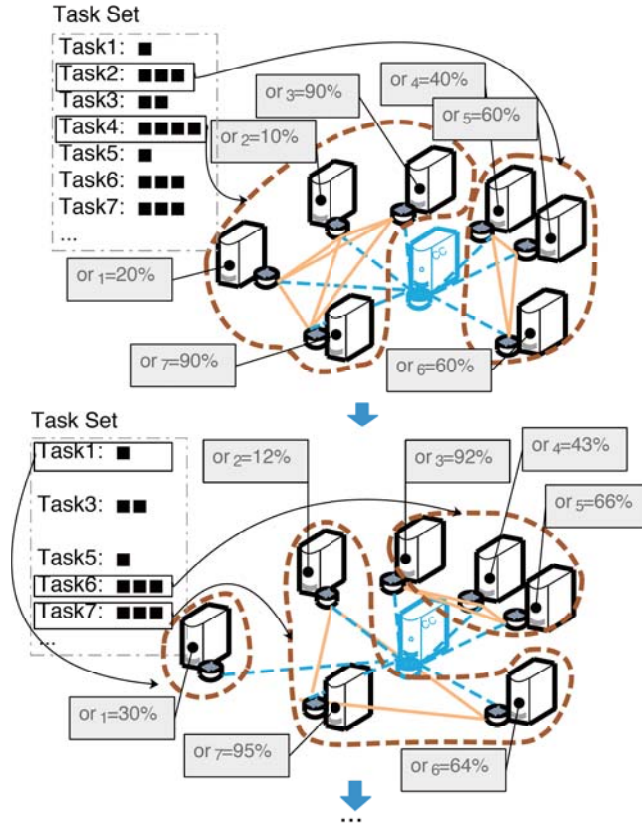


Figure 3.1: Dynamic coalition formation in an edge computing system.

3.2 An Illustrative Example of High-workload Task Allocation

As the real-time video analytic illustrated in Chapter 2, if a video analytic of a certain period is regarded as a task, both the task number and task workload will increase along with the increase of cameras. However, the increase speed of the servers may not catch up the increase of the burden from tasks. That means the limited numbers of servers should cooperate to perform the tasks with high-workload. Specifically, as shown in Figure 3.1, there is

an edge computing system consisting of one centralized coordinator, seven resource-limited edge servers and a task set consisting of several tasks. Each task has a different workload and a corresponding number of servers is required to perform the tasks to satisfy some requirements such as maximum latency. We assume each box represents one unit of workload (e.g., 1000 million CPU cycles) and one edge server can perform just one unit of workload. For instance, task 2 requires at least three edge servers to form a coalition to perform cooperatively. Moreover, each edge server's status (defined as edge server i's occupancy rate $or^i \in \{1\%, 2\%, ..., 100\%\}$ which is inverse proportion to the number of tasks allocated on it) is dynamically altered by performing tasks and influences the task performances. When tasks are executed, the corresponding energy would be consumed. We assume that the energy consumed by each server is directly related to the occupancy rate or^i and so raising the occupancy rate increases the energy consumed.

In order to describe this dynamic, the statuses of all edge servers and the task set are regarded as state s. In each state, a coalition structure is determined and each coalition chooses a task to perform. Then, each or^i and the task set would be altered after performing the tasks (corresponds to a new state). Correspondingly, the coalition structure needs to be redetermined according to the new state, as is shown in Figure 3.1. Besides energy cost, changing the coalition structure of servers in edge computing at each step incurs a cost. That is because, forming a coalition of servers usually requires the participating servers to construct signal channels. Thus, formation/release of coalitions would need the construction/release signal channels which would incur some costs. Thus, the goal of the problem is to make the servers form coalitions to perform all the tasks as soon as possible, while minimizing

both the costs of energy consumed and changes of coalition structures.

This scenario well reflects the difficulties in dynamic coalition formation in edge computing: it includes both features of dynamic and coalition formation where the coalitions should change to suit the state at each step. Moreover, the state space is huge: the state size exponentially increases with the number of either edge servers or tasks. For instance, each server has 100 possible statuses (or^i: $0.01 \sim 1$ with scale interval of 0.01), thus three servers has already corresponded to 10^6 joint statuses.

3.3 Coalitional Markov Decision Process (CMDP)

In this section, we propose a dynamic coalition formation model to cope with this problem. Before introducing our proposed model, we will explain some preliminaries of coalition formation and dynamic decision process.

3.3.1 Coalition Structure Generation

The models in coalition formation theory can be divided into two types: self-interested setting and cooperative setting. In the self-interested setting, one classical model is the transferable utility game (TUG) which is defined as the pair $(\mathscr{N}, v_{cha})$ where $\mathscr{N} = \{1, ..., n\}$ is a set of agents, and $v_{cha} : 2^{\mathscr{N}} \to \mathbb{R}$ is the characteristic function used to evaluate the value of a coalition c (a coalition is formed by several agents joining which is denoted as c, i.e., $c \subseteq \mathscr{N}$) [Banerjee et al., 2001]. Since each agent can choose only one coalition to join, several disjoint coalitions can be formed where each combination of coalitions is called a coalition structure and denoted as cs, i.e., $cs = \{c_1, ..., c_{|cs|}\}$ that satisfies $(\forall i \neq j, c_i \cap c_j = \emptyset) \wedge (\bigcup_{i=1}^{|cs|} c_i = \mathscr{N})$.

The goal of TUG is to solve a core which is the set of all stable outcomes that no coalition wants to deviate from, i.e., $core = \{(cs,x) | \sum_{i \in c} x_i \geq v_{cha}(c) \quad \text{for any} \quad c \subseteq \mathscr{N}\}$ where x is the vector used to divide the rewards earned by the coalition among each of its members. $\sum_{i \in c} x_i \geq v_{cha}(c)$ means that each coalition earns at least as much as it can make on its own. Thus, the self-interested agent will not leave its current coalition if a core is identified in TUG.

In the cooperative setting, one classical model is called coalition structure generation (CSG) [Rahwan et al., 2015]. The goal of CSG is to identify optimal coalition structure $cs*$ that has the maximized sum of all coalition values, i.e., $cs^* = argmax_{cs \in \mathscr{P}^{\mathscr{N}}} \sum_{c_i \in cs} v_{cha}(c_i)$ where $\mathscr{P}^{\mathscr{N}}$ is the set of all possible coalition structures. Since we consider the cooperative setting of edge cloud computing, we use CSG to model part of server coalition formation. However, CSG terminates once the optimal coalition structure has been found, which does not support the dynamic processes found in edge cloud computing system. Thus, we need to propose a dynamic coalition formation model in this case.

3.3.2 Markov Decision Process

Markov decision process (MDP) is a traditional model for discrete time decision-making process [Littman, 1994]. In MDP, a subject that makes a decision is called an agent and the other subject that is impacted by the agent is called the environment. The agent observes state $s \in \mathscr{S}$ of the environment ($\mathscr{S}$ is the set of states can be observed), and takes action $a \in \mathscr{A}$ from the action set. Then, the environment probabilistically transfers to the next state s' based on a transition function $\mathscr{T} : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \rightarrow [0,1]$. Cor-

Correspondingly, based on the transition of (s,a,s'), the agent obtains a reward $r(s,a,s')$ determined by reward function $r : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \rightarrow \mathbb{R}$. The agent's goal is to maximize its accumulated rewards $\sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1})$ during a period by learning an optimal policy.

3.3.3 CMDP Model

Based on MDP and CSG, we propose a dynamic coalition formation model called coalitional Markov decision process (CMDP). We first introduce CMDP model and then state how to formulate the dynamic coalition formation problem in edge computing as a CMDP.

Agent and coalition: We consider a set $\mathscr{N} = \{1,...,n\}$ of agents. Each agent can form a coalition c_i with other agents and each agent can only join one coalition. Thus, a way of coalition formation is called a coalition structure denoted as cs, i.e., $cs = \{c_i | \ (\forall i \neq j, c_i \cap c_j = \emptyset) \wedge (\bigcup_{i=1}^{|cs|} c_i = \mathscr{N})\}$. Then, we use $\mathscr{P}^{\mathscr{N}} = \{cs_1,...,cs_{|\mathscr{P}^{\mathscr{N}}|}\}$ to denote the set of all possible coalition structures.

State: We use state s to denote the status of environment and define a set $\mathscr{S} = \{s_1, s_2, ..., s_{|\mathscr{S}|}\}$ including all possible states.

Action: We denote $\mathscr{A}^{c_i}$ as the action set of coalition c_i where each coalition c_i chooses an action from $\mathscr{A}^{c_i}$. We denote joint action α^{cs} from all coalition actions under a given coalition structure cs, i.e., $\alpha^{cs} = a^{c_1} \times ... \times a^{c_{|cs|}}$. Correspondingly, the set of joint action α^{cs} is denoted by $\mathscr{A}^{cs} = \mathscr{A}^{c_1} \times ... \times \mathscr{A}^{c_{|cs|}}$.

Transition function: Based on current state s, the environment probabilistically transfers to the next state s', after taking joint action α^{cs}. This proba-

bility function, called the transition function, is denoted as $\mathscr{T} : \mathscr{S} \times \mathscr{A}^{cs} \times \mathscr{S} \rightarrow [0,1]$, i.e., $\mathscr{T}(s, \alpha^{cs}, s')$=$Prob(s'|s, \alpha^{cs})$.

Reward function: $r : \mathscr{S} \times \mathscr{A}^{cs} \times \mathscr{S} \rightarrow \mathbb{R}$ is the reward function. In this case, what we want to maximize is the sum rewards from all agents rather than an individual agent's reward. The reward function is similar to the characteristic function in CSG which is used to evaluate the values of different coalitions. However, there are two differences: 1) CSG reflects a static coalition formation process, where it is solved once an optimal coalition structure is found. Thus, the characteristic function involves only coalitions. However, in CMDP, there is a dynamic state transition process during coalition formation; this means an optimal coalition structure must be identified in each state. Thus, the reward function should involve both coalition and state. 2) Moreover, in CSG there is no concept of action where the agents desire only to form coalitions. However, after forming a coalition from several agents, the coalition must choose an optimal action from an action set. Thus, the reward function is related to the state-coalition-action pair rather than a single coalition. Moreover, the characteristic function can be regarded as a special case of reward function when CMDP consists of only one state and one action.

Cost function: In the process of dynamic coalition formation, altering the coalition structure would bring corresponding cost, since coalition formation/dissolution in real-world scenarios usually corresponds to physical activities which incur some costs like energy which cannot be ignored. Thus, we define $cost : \mathscr{S} \times \mathscr{P}^{\mathscr{N}} \times \mathscr{S} \times \mathscr{P}^{\mathscr{N}} \rightarrow \mathbb{R}$ as the cost function to calculate the cost incurred by altering a coalition structure.

Policy: In CMDP, a coalition's decision-making for choosing an action pro-

ceeds in two phases: coalition formation for all agents and each coalition taking an action. Thus, it corresponds to two types of policies, separately. As the coalition formation, we define policy $\pi^{cs} : \mathscr{S} \times \mathscr{P}^{\mathscr{N}} \rightarrow [0,1]$ to denote the probability of forming coalition structure cs under state s, i.e., $\pi^{cs}(s, cs) = Prob(cs|s)$. Then, each coalition c can choose an action from its action set. We collect the actions from all coalitions which is defined as α^{cs}. Correspondingly, we define a policy $\pi^{\alpha} : \mathscr{S} \times \mathscr{P}^{\mathscr{N}} \times \mathscr{A}^{cs} \rightarrow [0,1]$ to denote the probability of choosing a joint action under state s and coalition structure cs.

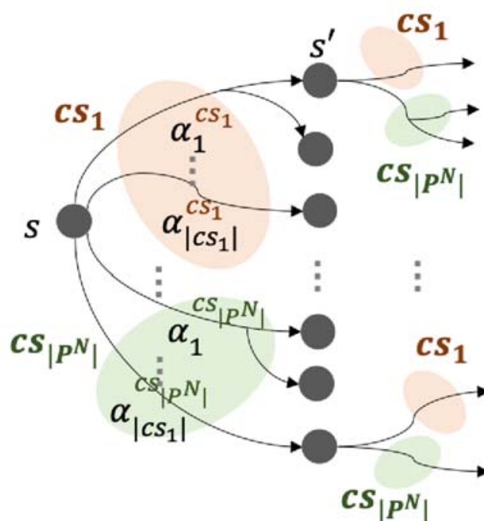


Figure 3.2: The dynamic transition process in CMDP.

Objective function: To sum up, we use the tuple $< \mathscr{N}, \mathscr{S}, \mathscr{A}^{c_i}, \mathscr{T}, r, cost^{cs} >$ to denote a CMDP. Figure 3.2 shows how CMDP formulates a dynamic coalition formation. Under current state s, coalition structure cs is determined based on policy π^{cs}. Then, based on policy π^{α}, a joint action α^{cs} is determined. By implementing α^{cs}, the environ-

ment transfers to the next state based on transition function $\mathscr{T}$ and the above process is repeated. Finally, we can obtain a trajectory $h = [s[1], cs[1], \alpha^{cs}[1], s[2], ..., s[T], cs[T], \alpha^{cs}[T], s[T+1]]$ based on that dynamic coalition formation process. Correspondingly, we can obtain the discounted sum *return* $R(h)$ of immediate rewards along trajectory h. It is defined as

$$R(h) = \sum_{t=1}^{T} \gamma^{t-1} r(s[t], \alpha^{cs}[t], s[t+1]), \tag{3.1}$$

where $\gamma \in [0, 1)$ is the discount factor to denote how important the rewards obtained in the future are. Unlike ordinary MDP, the cost of altering coalition structures also needs to be considered in CMDP which is a discounted sum $C(h)$ of immediate costs along trajectory h defined as :

$$C(h) = \sum_{t=1}^{T} \gamma^{t-1} cost^{cs}(s[t], cs[t], s[t+1], cs[t+1]). \tag{3.2}$$

The goal of CMDP is to learn optimal policies of π^{cs} and π^{α} that can maximize $R(h)$ and the cost $C(h)$; $C(h)$ is always a negative number (the bigger the cost of changing cs is, the smaller the value of $cost^{cs}$ is). Thus, we use weighted sum $J(h)$ as the objective function of CMDP which is defined as

$$J(h) = R(h) + \omega C(h), \tag{3.3}$$

where ω is the weight used to evaluate how important the cost is. Then, the goal changes to identify a couple of policies π^{cs} and π^{α} that can maximize the expectation of weighted sum $J(h)$ along trajectory h as follows.

$$\pi^{cs*}, \pi^{\alpha*} = \arg\max_{\pi^{cs}, \pi^{\alpha}} \mathbb{E}_{p^{\pi^{cs}, \pi^{\alpha}}(h)} \left[J(h) \right], \tag{3.4}$$

where $\mathbb{E}_{p^{\pi^{cs},\pi^{\alpha}}(h)}$ denotes the expectation over trajectory h drawn from $p^{\pi^{cs},\pi^{\alpha}}(h)$ which denotes the probability density of observing trajectory h under policies π^{cs} and π^{α}.

Compared with ordinary MDP, there are two major differences: 1) each coalition as an entity takes an action rather than an agent. This means that forming different coalitions corresponds to a different action set, which is more complicated than the fixed action set of ordinary MDP. 2) it does cover the concept of coalitions and altering coalition structures incurs a certain cost for pair (s, cs, s', cs') that cannot be evaluated by the reward function of naive MDP which is based on (s, α, s'). Thus, we define a cost function to evaluate the cost incurred by altering the coalition structure. Since CMDP is not a naive MDP, it requires us to propose a new algorithm that can handle its specific properties.

The Equivalent Policy

Since it is hard to do an optimization with considering both the policies π^{cs} and π^{α} meanwhile, we consider to construct an equivalent policy by using their relationship. Specifically, policy π^{α} usually makes a decision after that coalition structure cs is determined by π^{cs}, thus we can regard cs as a condition of π^{α}. We thus construct equivalent policy π^{eq} which is defined by

$$\pi^{eq} : \mathscr{S} \times \mathscr{A}^{all} \to [0, 1],$$

where $\mathscr{A}^{all} = \bigcup_{cs \in \mathscr{P}_{\mathscr{N}}} \mathscr{A}^{cs}$ includes joint actions from all possible coalition structures ($\alpha \in \mathscr{A}^{all}$). Thus, $\pi^{eq}(s, \alpha)$ is the probability of choosing joint action α from $\mathscr{A}^{all}$ under state s. Based on the relationship $\pi^{eq}(s, \alpha) = \pi^{cs}(s, cs)\pi^{\alpha}(s, cs, \alpha^{cs})$, both policies π^{cs} and π^{α} can be derived

from $\pi^{eq}(s,\alpha)$ as follows.

$$\begin{aligned}\pi^{cs}(s,cs) &= \sum_{\alpha\in\mathscr{A}^{cs}} \pi^{eq}(s,\alpha), \\ \pi^{\alpha}(s,cs,\alpha^{cs}) &= \frac{[\sum_{\alpha'\in\mathscr{A}^{cs}} \pi^{eq}(s,\alpha')]\pi^{eq}(s,\alpha)}{\sum_{\alpha\in\mathscr{A}^{all}} \pi^{eq}(s,\alpha)}.\end{aligned} \tag{3.5}$$

Thus, the optimal policy π^{eq*} is defined by

$$\pi^{eq*} = \arg\max_{\pi^{eq}} \mathbb{E}_{p^{\pi^{eq}}(h)}\big[J(h)\big]. \tag{3.6}$$

The Value Functions

In the MDP, state-action value $Q(s,a)$ is often used to evaluate the quality attained by taking action a under state s. Then, an optimal policy can be obtained based on the value $Q(s,a)$ (Q-value). Besides the reward, the cost of altering coalition structures also exists in CMDP. Thus, we need to consider two Q-value functions to evaluate the reward and cost, separately. We define the Q-value function $Q^r : \mathscr{S} \times \mathscr{A}^{all} \to \mathbb{R}$ used to evaluate the rewards which is called *state-action value function for reward*, as

$$Q^r(s,\alpha) = \mathbb{E}_{p^{\pi^{eq}}(h)}\big[R(h) \mid s[1]=s, \alpha[1]=\alpha\big], \tag{3.7}$$

where $Q^r(s,\alpha)$ is the expected rewards obtained by following policy π^{eq} under the condition of "$|s[1]=s,\alpha[1]=\alpha$". By recursion, we can rewrite

$Q^r(s,\alpha)$ as follows.

$$Q^r(s,\alpha) = \sum_{s' \in \mathscr{S}} \mathscr{T}(s,\alpha,s')\big[r(s,\alpha,s') + \gamma \sum_{\alpha' \in \mathscr{A}^{all}} \pi^{eq}(s',\alpha')Q^r(s',\alpha')\big]. \tag{3.8}$$

We define the Q-value function $Q^c : \mathscr{S} \times \mathscr{A}^{all} \to \mathbb{R}^-$ used to evaluate the cost of altering coalition structures, which is called as *state-action value function for cost*, as

$$Q^c(s,\alpha) = \mathbb{E}_{p^{\pi^{eq}}(h)}\big[C(h)|s[1] = s, \alpha[1] = \alpha\big], \tag{3.9}$$

where $Q^c(s,\alpha)$ is the expected cost obtained by following policy π^{eq} under the condition of "$|s[1] = s, \alpha[1] = \alpha$". By recursion, we can rewrite $Q^c(s,\alpha)$ as follows.

$$Q^c(s,\alpha) = \sum_{s' \in \mathscr{S}} \mathscr{T}(s,\alpha,s')\big[\sum_{cs' \in \mathscr{P}^{\mathscr{N}}} \pi^{cs}(s,cs') cost^{cs}(s,cs,s',cs') + \gamma \sum_{\alpha' \in \mathscr{A}^{all}} \pi^{eq}(s',\alpha')Q^c(s',\alpha')\big]. \tag{3.10}$$

Since the objective function $J(h)$ consists of both $R(h)$ and $C(h)$. We also consider a Q-value to evaluate $J(h)$ which is defined as *weighted state-action value function*. By calculation, it can be written in terms of Q^r and Q^c as follows.

$$Q^\omega(s,\alpha) = Q^r(s,\alpha) + \omega Q^c(s,\alpha). \tag{3.11}$$

Based on Eqs. (7) and (8), $Q^{\omega}(s,\alpha)$ can be written as

$$\begin{aligned} Q^{\omega}(s,\alpha) = \sum_{s' \in \mathscr{S}} & \mathscr{T}(s,\alpha,s')\big[r(s,\alpha,s') + \\ & \omega \sum_{cs' \in \mathscr{P}^{\mathscr{N}}} \pi^{cs}(s',cs') cost^{cs}(s,cs,s',cs') + \\ & \gamma \sum_{\alpha' \in \mathscr{A}^{all}} \pi^{eq}(s',\alpha') Q^{\omega}(s',\alpha')\big], \end{aligned} \tag{3.12}$$

where $Q^{\omega}(s,\alpha)$ includes both terms of reward $r(s,\alpha,s')$ and cost $cost^{cs}(s,cs,s',cs')$. We can solve $Q^{\omega}(s,\alpha)$ to obtain an optimal policy that maximizes $J(h)$.

3.4 Dynamic Coalition Formation Algorithms

As illustrated in the above section, CMDP cannot be solved by the classical algorithms developed for solving MDP. We first introduce a basic solution called coalitional Q-learning (CQL) as first seen in [Ding and Lin, 2020]. Then, we introduce our proposed method DCQL in this section which can handle a larger state space than CQL.

3.4.1 Coalitional Q-learning

Q-learning, is a classical algorithm that can guide agents in learning an optimal Q-value for decision-making [Watkins and Dayan, 1992]. We refer to the framework of Q-learning and consider the features of CMDP to propose an algorithm called coalitional Q-learning (CQL) to solve CMDP in [Ding and Lin, 2020]. First, based on Eq.(3.12), we further introduce an *optimal weighted state-action value function* $Q^{\omega *}$ by applying a maximization

operator which is defined as follows

$$Q^{\omega*}(s,\alpha) = \sum_{s' \in \mathscr{S}} \mathscr{T}(s,\alpha,s')\big[r(s,\alpha,s') + \max_{\alpha'}\big\{\omega cost^{cs}(s,cs,s',cs^{-1}(\alpha')) + \gamma Q^{\omega*}(s',\alpha')\big\}\big], \tag{3.13}$$

where $cs^{-1} : \mathscr{A}^{all} \to \mathscr{P}^{\mathscr{N}}$ is a function to obtain the cs given as α^{cs}. In Q-learning algorithm, there also exists a maximization operator to choose the maximized Q-value of next state to update the Q-value of current state. However, in the Eq. (3.13), the maximization operator needs also to consider the corresponding cost of changing coalition structures rather than only Q-value, as shown in Figure 3.3.

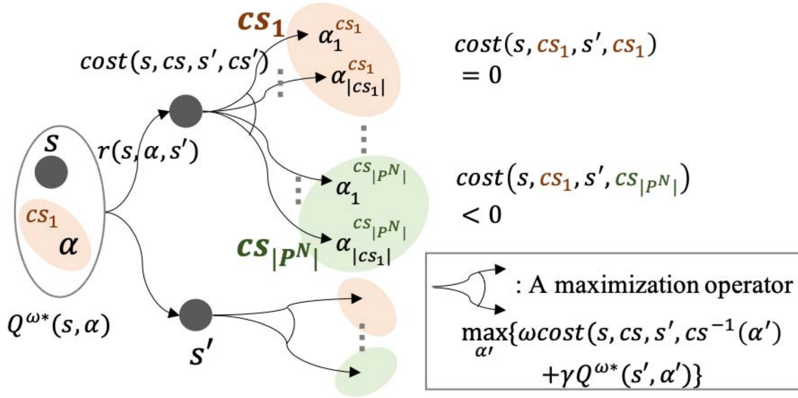


Figure 3.3: The definition of optimal weighted state-action value function.

In the CQL algorithm, the value of $Q^{\omega*}$ is updated as follows.

$$Q^{\omega*}(s[t],\alpha[t]) \leftarrow Q^{\omega*}(s[t],\alpha[t]) + \sigma\big[r[t+1] + \max_{\alpha'}\big\{\omega cost^{cs}(s[t],cs[t],s[t+1],cs^{-1}(\alpha')) + \gamma Q^{\omega*}(s[t+1],\alpha')\big\} - Q^{\omega*}(s[t],\alpha[t])\big], \tag{3.14}$$

where σ is the learning rate. Unlike Q-learning where the maximization operator only considers Q-value, the maximization operator of CQL needs also to consider the corresponding cost of changing coalition structures rather than only Q-value. As shown in the right part, the cost will be zero if the coalition structure does not change, the cost corresponds to a negative value otherwise.

3.4.2 Deep Coalitional Q-learning

Coalitional Q-learning is based on a tabular RL method which demands maintenance of a Q-table. Although tabular RL methods have good performance in many RL tasks, it becomes impractical for RL tasks with large state spaces since it is hard to maintain a huge table. For instance, in our motivating scenario, the state would increase with the number of tasks in the task set. To tackle this problem, we refer to deep RL (DRL) algorithms which combine deep neural networks with reinforcement learning. They have been verified as capable of dealing with large state spaces. Deep Q-network (DQN), is a classical DRL method proposed by Volodymyr et al. [Mnih et al., 2015]; a neural network is used to approximate the value of $Q(s,a)$. Inspired by the idea of DQN, we improve CQL and propose a novel algorithm called DCQL for CMDP with large state space. Specifically, the optimal weighted state-action value $Q^{\omega *}$ is calculated by a deep neural network. It uses the tuple of $(s_j, \alpha_j, r_j, cost_j, s_{j+1})$ obtained at each step to train the network with the goal of minimizing the following loss equation.

$$L(\theta) = \mathbb{E}_{s,a \sim \pi}\left[\left(y_j - Q^{\omega *}(s_j, \alpha_j; \theta)\right)^2\right], \tag{3.15}$$

where

$$y_j = \begin{cases} r_j + \omega cost_j^{cs} & \text{if } s_{j+1} \text{ is terminal state,} \\ r_j + \gamma \max_{\alpha'}(\omega cost_j^{cs} + Q^{\omega *}(s_{j+1}, \alpha'; \theta)) & \text{otherwise.} \end{cases}$$

where θ represents the parameters of the neural network.

Further, we use the motivating scenario in section 3.2 to illustrate DCQL, see in Figure 3.4. The statuses of all servers and task set are regarded as a state which allows DCQL to output each action's $Q^{\omega *}$ value. Based on the $Q^{\omega *}$ value, an action is chosen and the state transfers to the next state. Then, we can obtain the reward and cost to train this network through minimizing the loss function defined by Eq. (3.15).

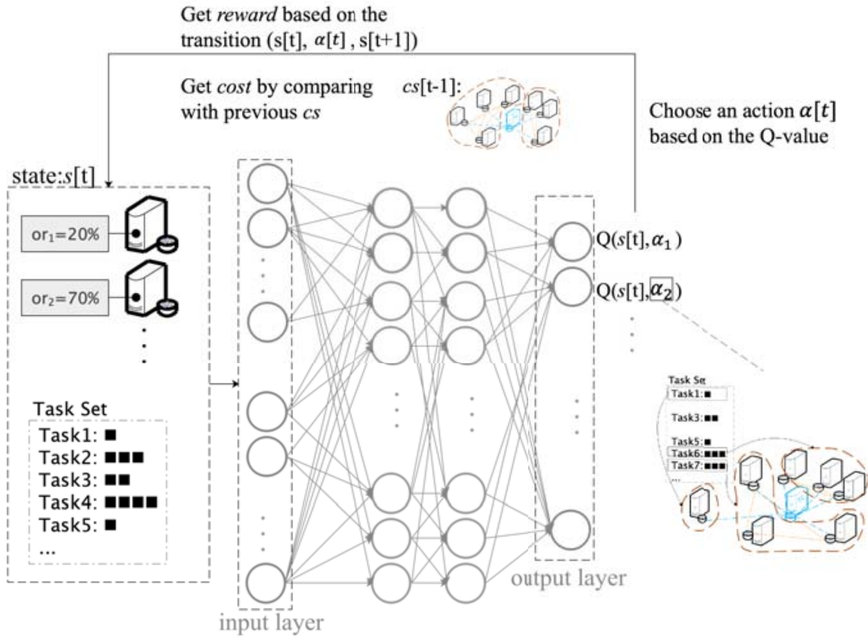


Figure 3.4: Deep coalitional Q-learning algorithm.

Then, we analyze the computation time of DCQL algorithm with the CQL algorithm. Both algorithms consist of two phases: learning phase and executing phase.

- 1) Learning phase: DCQL algorithm trains a deep neural network by back propagation based on Eq. (3.15). CQL algorithm maintains a Q-table and updates the Q-values for each state-action pair based on Eq. (3.14).

- 2) Executing phase: DCQL algorithm inputs the current state and outputs the corresponding Q-values for each action by performing forward propagation. Then it chooses an action based on the Q-values. CQL algorithm searches for the Q-values corresponding to the current state in the Q-table and chooses an action based on the Q-values.

Although DCQL algorithm incurs a longer calculation time than the CQL algorithm in the learning phase, they take similar time to choose an action in the executing phase. In actual experiments, we care only about the executing phase once it has been trained well. Specifically, DCQL is given by Algorithm 1.

3.5 Evaluation

In this section, we run the experiments of dynamic coalition formation problem in edge computing to verify the effectiveness of our proposed DCQL algorithm.

ALGORITHM 1: Deep Coalitional Q-learning (DCQL) Algorithm

1 Initialize replay memory D
2 Initialize weighted action-value function $Q^{\omega*}$ with random weights θ
3 **for** *episode m=1, M* **do**
4 Generate initial state $s[1]$
5 **for** *step t =1, T* **do**
6 Randomly select an action $\alpha[t]$ from $\mathscr{A}^{all}(s[t])$ based on a ε-greedy policy
7 otherwise select $\alpha[t] = \arg\max_{\alpha'} \{\omega cost^{cs}(s[t-1], cs[t-1], s[t], cs^{-1}(\alpha'))+$
8 $\gamma Q^{\omega*}(s[t], \alpha'; \theta)\}$
9 Execute action $\alpha[t]$ and observe reward $r[t] = r(s[t], \alpha[t], s[t+1])$ and cost $cost^{cs}[t] = cost^{cs}(s[t-1], cs[t-1], s[t], cs[t])$,
10 and then transfer to the next state $s[t+1]$
11 Store transition $(s[t], \alpha[t], r[t], cost^{cs}[t], s[t+1])$ in D
12 Set $s[t] = s[t+1]$
13 Sample random mini-batch of transitions $(s_j, a_j, r_j, cost_j^{cs}, s_{j+1})$ from D
14 Calculate y_j based on the Eq. (3.15) and perform a gradient descent step to update θ
15 **end**
16 **end**

Evaluation Setting

We refer to related sections [Chen et al., 2015][Liu et al., 2020b][Wen et al., 2012][Armenta-Cano et al., 2015][Aydin et al., 2004] to define the parameters of the following edge computing systems. And also, we make some simplifications which do not degrade the validity of our model. The specific setting is stated as follows.

Server: DCQL aims to solve a large state space problem in this section.

Since each server has 100 statuses of CPU occupancy rate, it corresponds to 10^6 joint statuses even if only 3 edge servers are considered. Thus, we take an edge computing system composed of 3 edge servers. We define the set of servers as $SV = \{sv^1, sv^2, sv^3\}$ where each server's parameter is denoted by the vector

$$sv^i = [or^i, f^i, ec^i], \tag{3.16}$$

where or^i is the CPU occupancy rate, f^i is server i's coefficient about CPU speed (e.g., workload performed per CPU cycle), ec^i is server i's coefficient about energy cost (e.g., energy consumed per CPU cycle) which can be calculated by the measurement method stated in [Wen et al., 2012]. Although the parameter values can be set based on some physical servers like Raspberry Pi , it would not influence the effectiveness of our proposed method. Thus, we set each server as $f^i = 1$ and $ec^i = 1$ for simplicity.

Task: As for setting the task set, we refer to [Chen et al., 2015] [Liu et al., 2020b] to denote CPU cycles as the task workload. Although some other task parameters like RAM and Disk can also be considered, we ignore them for simplicity and focus on workload in this section. That is because high workload has already well reflected the necessity of dynamic coalition formation and ignoring the other parameters does not degrade the validity of our model. We consider several task sets with different number of tasks and three types of tasks with workload: 1000, 2000 and 3000 million CPU cycles, separately. We assume 1000 million CPU cycles as a unit workload and it can be allocated to only one server at a time. For instance, a task with 2000 million CPU cycles requires two servers. Then, the server can execute the tasks in parallel by applying a time-slice mechanism. Without loss of generality, the number of each type task is randomly generated in

each task set. Each episode is concluded when all tasks in the task set have been completed.

Then, we formulate the problem as a CMDP and the corresponding elements of CMDP in this problem are given as follows.

State: The dynamic of edge computing is represented as state transition. Since the dynamic is driven by the factors of server status and task set status, we construct the state $s[t]$ at step t by including $or^i[t]$ from all servers at step t and the current task set status at step t, i.e., $s[t] = \big[[or^1[t], or^2[t], or^3[t]], [n^{ty_1}[t], n^{ty_2}[t], n^{ty_3}[t]]\big]$, where ty_j is j type tasks whose workload includes j units and n^{ty_j} represents the number of type ty_j tasks left in the task set.

Action: In CMDP, at each step the entity choosing the action is the coalition, thus we denote $\mathscr{A}^c$ as the action set of each coalition c. In this case, $\mathscr{A}^c$ is the set of tasks that edge coalition c is capable to execute which also includes the action to choose no tasks, i.e., $a^c = 0$. Thus, the coalition action set is defined by

$$\mathscr{A}^c = \{ty_j \mid j = |c|\} \cup \{0\}. \tag{3.17}$$

Reward Function: The goal is to perform all the tasks as soon as possible while minimizing the cost of energy consumption and the cost incurred by coalition structure alteration. Thus, when all tasks have been performed, a large positive reward is given. Specifically, the reward function is defined as follows.

$$r(s, \alpha, s') = \begin{cases} -energy(s, s') + 100, & \text{if } s' \text{ is absorb state}, \\ -energy(s, s'), & \text{otherwise}. \end{cases} \tag{3.18}$$

where 100 is the reward of absorb state in which all tasks have been accomplished, $energy(s,s')$ is the server energy consumed in performing the tasks; we refer [Armenta-Cano et al., 2015] to define it as

$$energy(s,s') = \sum_i ec^i * or^i. \tag{3.19}$$

Then we refer [Aydin et al., 2004] to approximate server i's occupancy rate or^i by the task number ta_n allocated on server i which is given by $or^i = \frac{ta_n}{ta_{max}}$ where ta_{max} is the maximum task number can be allocated on it (we assume $ta_{max} = 100$ in this section).

Cost Function: As for calculating the cost of changing coalition structure, we assume it is determined by the number of changes in agent relationships. Specifically, we use an undirected graph to describe a coalition, thus a coalition structure can be represented as several undirected graphs. In an undirected graph, each agent is represented as a node and any two agents can share one edge. Then, altering the coalition structure means adding/deleting edges of the undirected graphs. Thus, the cost of altering a coalition structure can be calculated by the number of edge changes. First, we define set N_G^{cs} to describe each agent's neighborhood (the members in the same coalition) which is defined by

$$N_G^{cs} = \left\{ N_G^{cs}(i) = \{k | k, i \in c_j \in cs \wedge k \neq i\} \;\middle|\; \forall i \in \mathscr{N} \right\}. \tag{3.20}$$

Then we consider set $N_G^{\mathscr{P}^{\mathscr{N}}}$ which includes all possible N_G^{cs}, i.e. $N_G^{\mathscr{P}^{\mathscr{N}}} = \left\{ N_G^{cs_1}, N_G^{cs_2}, \ldots, N_G^{cs_{|\mathscr{P}^{\mathscr{N}}|}} \right\}$. Furthermore, we introduce function $D : N_G^{\mathscr{P}^{\mathscr{N}}} \times N_G^{\mathscr{P}^{\mathscr{N}}} \to \mathbb{R}$ to quantify the difference between two coalition structures, as

defined by

$$D(N_G^{cs},\ N_G^{cs'}) = \frac{1}{2}\sum_i \mid N_G^{cs}(i) \triangle\ N_G^{cs'}(i) \mid, \tag{3.21}$$

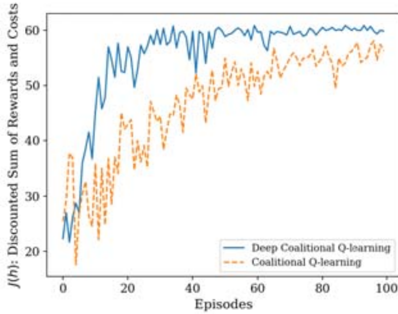
where $\triangle$ is symmetric difference calculation used to evaluate the number of edges altered. Thus, the cost of altering coalition structures can be represented by the value of $D(N_G^{cs},\ N_G^{cs'})$. We define cost function *cost* as the inverse number of $D(N_G^{cs},\ N_G^{cs'})$ as follows

$$cost^{cs}(s, cs, s', cs') = -D(N_G^{cs},\ N_G^{cs'}). \tag{3.22}$$

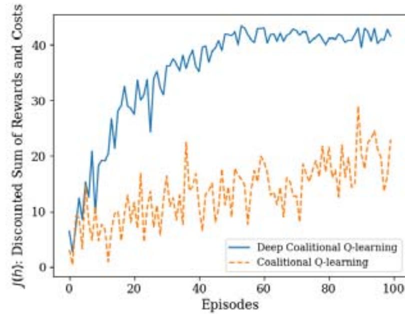
Evaluation Results

We check the performances of our DCQL algorithm by comparing it with the CQL algorithm. We set the same hyper-parameters for all algorithms with α=0.01 and γ=0.9. We consider task sets consisting of 5 tasks, 10 tasks, 15 tasks and 20 tasks. For each task set setting, we ran the following simulations for 100 episodes and each episode's maximum step is 100 (the episode is terminated even though some tasks in the task set are not executed); the results are shown in Figure 3.5. As shown in Figure 3.5 (a), since the task set has 5 tasks which corresponds to a small state space, CQL can learn effectively and matches the performance of DCQL. However, CQL cannot learn effectively when the task number increases which corresponds to a large state space, as shown in Figure 3.5 (b)-(d). DCQL has better performances than CQL since it can cope with large state spaces well.

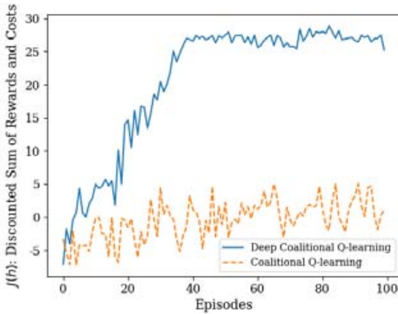
In order to compare these two algorithms visually, we focus on the final learning results (the average of the last 10 episodes) for all four task sets. With the increase of task number, it would naturally cost more steps to get



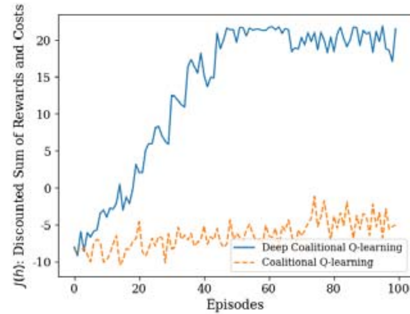
(a) The number of tasks: 5



(b) The number of tasks: 10



(c) The number of tasks: 15



(d) The number of tasks: 20

Figure 3.5: Comparing the performances of deep coalitional Q-learning with coalitional Q-learning.

the absorb state (all tasks have been executed completely). Correspondingly, the optimal value of $J(h)$ would decrease since taking more steps to get the absorb state corresponds to a smaller discount value of γ^{t-1} based on Eqs. (3.1)-(3.3). Thus, the $J(h)$ value of DCQL naturally decreases with the increase of task number, even though it can learn an optimal solution in each task set. Then, DCQL's superiority over CQL majorly depends on the CQL's performance. As shown in Figure 3.6, the percentage improvements attained by our method over CQL are 9.1%, 101.3%, 25.1 times and 26.1

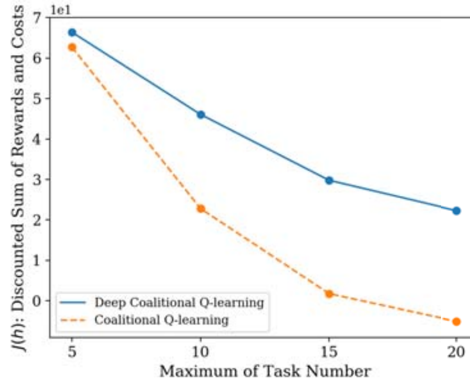


Figure 3.6: Comparing the final learning results of deep coalitional Q-learning with coalitional Q-learning.

times for the task sets with task numbers of 5, 10, 15 and 20. When the task number increases from 5 to 15, CQL's performance decreases quickly since a large state space problem is incurred. Thus, DCQL's superiority over CQL increases quickly. However, there exists a lower-bound of $J(h)$ value which corresponds to a worst case: the algorithm almost learns nothing and so the tasks are rarely executed. We terminate the episode once the maximum step is reached, thus $J(h)$ would not go to negative infinity with the increase of task number even though in the worst case. In task sets with 15 and 20 tasks, we can see CQL has the similar performances that can almost learn nothing (confirmed by Figure 3.5 (c)(d)), which are close to the lower-bound of $J(h)$ value. Thus, DCQL's superiority over CQL increases slowly when the task number increases from 15 to 20.

3.6 Summary

In this chapter, we considered the problem of high-workload task allocation in cooperative edge cloud computing. Compared with traditional high-workload task allocation methods, this thesis considers the dynamic features of edge cloud computing and has introduced two major novelties: 1) In addition to task information, the server status's dynamic change in status is also considered; 2) a long-term objective is considered rather than one step objective. We addressed these issues and formulated a theoretic model called CMDP. We propose a basic solution, CQL, which is a dynamic coalition formation algorithm that can ensure edge server cooperation. Moreover, we extended it by using a deep neural network called DCQL which can well handle the large state spaces expected. Therefore, this thesis has solved the high-workload task allocation problem in edge cloud computing. Although we employ Q-learning and DQN as described here, many other RL methods can also be applied to support different situations in edge cloud computing. This means that this work provides a new framework for server cooperation in performing high-workload tasks. These advances have been published in [Ding and Lin, 2022a].

Chapter 4

Graph Convolutional Reinforcement Learning for Dependent Task Allocation

In this chapter, we focus on the second case illustrated in Chapter 1 which is dependent task allocation in cooperative edge cloud computing. To solve the problem a graph convolutional reinforcement learning method is proposed.

4.1 Introduction

In the IoT environment, there is usually a dependency relationship among the tasks: some tasks can be performed only after accomplishing some other specific tasks. Thus, how to make optimal task allocation decisions while considering the dependency relationship of tasks, is an import problem. Although some studies have tackled this problem, they usually assume that

server status remains unchanged [Abdel-Jabbar et al., 2014][Sundar and Liang, 2016]. This assumption is usually invalid in edge computing since edge server status attributes like RAM resource change dynamically with the pushing/popping of tasks. Thus, this section tackles the dynamic dependent task allocation decision-making problem for resource-limited edge servers.

The problem can be summarized into two issues as follows. First, the status of server computation resources such as CPU, RAM and Disk are dynamically altered with task execution. Second, some tasks can only be executed after accomplishing some other specific tasks. As for the first issue, many works apply deep reinforcement learning (DRL), a classical method to train an intelligent agent to make dynamic optimal decisions, to task allocation in edge computing [Ma et al., 2020][Chen et al., 2018b]. Specially, a task-allocation agent is trained by the trial-and-error approach: an agent observes the current status of the edge servers and task arrival, and makes a task allocation decision (action). Its action is then evaluated by a reward function based on an objective (e.g. minimizing delay cost). After several epochs of training it with the target of maximizing the accumulated rewards, we can deploy the trained agent to the edge computing system to control task allocation in edge servers.

However, the second issue is not well handled by DRL algorithms. That is because DRL algorithms usually suit just non-graphical data such as a one-dimension vector [Wu et al., 2020]. In the dependent task allocation problem, the dependency information can be regarded as a kind of graphical information as represented by a directed acyclic graph (DAG): each task is a node and dependency is represented by directed edges. Moreover, failure to

well handle this kind of graphical information of dependency may degrade the task performance, since the different dependency relationships greatly influence the cost. Thus, DRL might not well handle the dependent task allocation problem.

To tackle these issues, we propose a graph convolutional reinforcement learning (GCRL)-based task-allocation agent that consists of an encoding part and a decision-making part. In the encoding part, we apply a graph convolutional network (GCN), one of classical graph neural networks (GNN), to capture the dependency information by modeling a dependency feature vector as a DAG. The embedding result output by GCN is used as part of the input for decision-making in task allocation. In the decision-making part, we formulate the problem as a Markov decision process (MDP), a classical model for discrete-time decision-making, where the status of a feature vector is taken to be a state, and a task-allocation scheme is taken to be an action. We then employ a DRL method called Deep Q-network (DQN) to solve the decision-making problem. To show the effectiveness of our proposed GCRL-based task-allocation agent, we run dependent task allocation experiments and compare it with existing methods.

The main contributions of this chapter are as follows:

- We consider the dependent task allocation problem in the dynamic environment of resource-limited edge computing and analyze the issues of this problem from the viewpoint of agent decision-making.
- We propose a novel graph convolutional reinforcement learning based task-allocation agent for dependent task allocation that can minimize energy cost and delay cost.

- We run experiments that show our proposed method outperforms existing algorithms to confirm its effectiveness.

4.2 An Illustrative Example of Dependent Task Allocation

Dependent tasks in edge computing can appear in many IoT use cases. As shown in the left part of Figure 4.1, we consider the scenario of environment monitoring in a hospital, where wireless temperature sensors and wireless humidity sensors collect the raw data of temperature and humidity. Based on the comfort coefficient calculated by the data, some other tasks such as emergency alerts can be invoked. A set of multiple dependent tasks (collectively called a job) can be represented as a DAG: the circles (nodes) represent tasks and directed edges represent the dependency relationships. Each task's workload (CPU cycles) is represented by the size of the circle and the primary colors represent the different computation resources required CPU, RAM and Disk. Thus, a task's feature vector can be represented by a circle with different color and size. Although we take the sensors as illustrative examples, it can be extended to other IoT devices like the cameras illustrated in Chapter 2.

As for the dependent task allocation process shown in Figure 4.1, there exists a task-allocation agent and several heterogeneous edge servers. In each step, a job consisting of several dependent tasks arrives to the task-allocation agent. It observes the current status of all servers and allocates the job to one of the servers. A different task allocation usually corresponds to a different cost such as delay cost and energy cost. Task allocation also

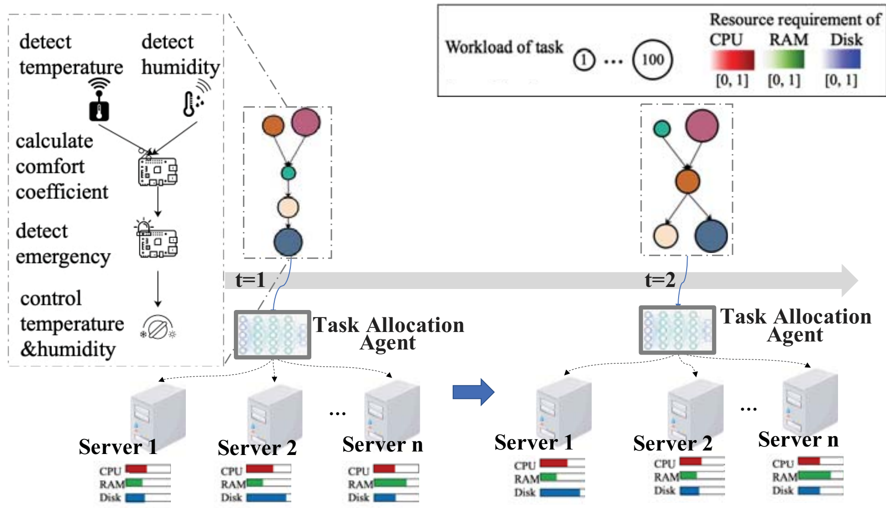


Figure 4.1: Dependent task allocation in edge computing.

changes the computation utilization of the edge servers which would influence subsequent decision-making. Moreover, the costs are sensitive to task dependency. For instance, the tasks of the job at t=1 and t=2 have the same parameter values, however, the dependencies are different; thus they would have different costs even if allocated to the same server. The dependency in a job greatly influences the cost which makes coping with task dependency difficult. Thus, given a certain period, the goal is to train an optimal task-allocation agent that can minimize the total cost during the period while satisfying the resource constraints imposed by the servers.

4.3 Dependent Task Allocation Problem

In this section, we define the above dependent task allocation in detail.

We defined task ta in Chapter 3 in a simple way, this chapter extends it and

defines it in a general way as follows.

Job and **Task** : At each step, a job consisting of several tasks arrives at the task-allocation agent; it is written as job=$(\mathbf{ta},\mathbf{dep})$ where **ta** is the set of included tasks

$$\mathbf{ta} = \{ta_1,...,ta_j,...,ta_{|\mathbf{ta}|}\}, \tag{4.1}$$

task ta_j can be represented by

$$ta_j = [wok_j, Req_j^{CPU}, Req_j^{RAM}, Req_j^{Disk}], \tag{4.2}$$

where wok_j is task j's workload required to be performed, Req_j^{CPU} is task j's CPU resources required, Req_j^{RAM} is task j's RAM resources required, and Req_j^{Disk} is task j's $Disk$ resources required.

dep = $\{(ta_j,ta_k)|ta_j \succ ta_k, ta_{j,k} \in \mathbf{ta}\}$ represents the dependency of tasks where "$ta_j \succ ta_k$" means that task ta_j can only be performed after accomplishing task ta_k.

Server: Although we have defined server in Chapter 3, it only considers three servers. In this chapter, we define it in a more general way. The edge computing system consists of several edge servers, and is denoted by

$$SV = \{sv^1,...,sv^i,...,sv^{|SV|}\}. \tag{4.3}$$

The characteristic parameters of each server $sv^i \in SV$ are denoted by the following vector

$$sv^i = [f^i, ec^i, CPU_{max}^i, RAM_{max}^i, Disk_{max}^i], \tag{4.4}$$

where f^i is server i's *computation rate* denoting the executed CPU cycles per second, ec^i is server i's *unit energy cost* denoting the energy consumed per CPU cycle, its unit of measure is J/cyc (J is Joule and cyc is CPU cycle), and it can be calculated via the measurement method stated in [Wen et al., 2012], CPU^i_{max} is the maximum *CPU capacity* of server i, RAM^i_{max} is the maximum *RAM capacity* of server i, and $Disk^i_{max}$ is the maximum *Disk capacity* of server i. These values are assumed to be constants.

Furthermore, the status of available server resources, which are altered by pushing/poping tasks, is called *server status*, and is defined as the following vector.

$$\mathbf{sv}^i = [CPU^i, RAM^i, Disk^i], \tag{4.5}$$

where CPU^i, RAM^i and $Disk^i$ are the currently available resources of CPU, RAM and $Disk$, respectively.

In this section, we consider two types of cost, one is energy cost which depends on the server energy cost, denoted as en_t=$-en^i$ where server i is allocated the job arriving at step t, and the other is delay cost, which is computation time to complete the task allocated to server i; it is defined as

$$delay_t = -\max_j \Big(\frac{wok_j}{f^i} + \max_{j' \in N_j}\big(\frac{wok_{j'}}{f^i}\big)\Big), \tag{4.6}$$

where $\frac{wok_j}{f^i}$ is the computation time when task ta_j is allocated to server i, the bigger the value of f^i is, the faster the task would be accomplished, N_j is the set of all dependent tasks of task ta_j, and ta_j must wait until all its dependent tasks have been accomplished with the wait time of $\max_{j' \in N_j}(\frac{wok'_j}{f^i})$. Finally, we take the maximum computation time of all tasks in the job as the delay

cost.

The goal is to minimize the sum of the immediate costs within period h with T steps, which is given by

$$Cost(h) = \sum_{t=1}^{T} \gamma^{t-1} cost_t, \tag{4.7}$$

where γ is the discount factor, $cost_t$ is the cost obtained at step t and is defined in a weighted summation form as follows.

$$cost_t = \omega_1 en_t^{nor} + \omega_2 delay_t^{nor}, \tag{4.8}$$

where $en_t^{nor} \in [-1, 0]$ and $delay_t^{nor} \in [-1, 0]$ are the corresponding normalized values of en_t and $delay_t$, and $\omega_1, \omega_2 \in [0, 1]$ denote the weighting parameters of energy cost and delay cost for decision-making, respectively.

4.4 Graph Convolutional Reinforcement Learning Algorithms

4.4.1 Case 1: Task Allocation with Single Job

Some DRL algorithms have been applied to task allocation, however they rarely consider task dependency. Although DRL algorithms like DQN can also be utilized for dependent task allocation by reforming graphical dependency information as a one-dimension vector, they may not have good performance. That is because a job consisting of dependent tasks is essentially graphical data and DRL algorithms cannot capture the kind of graph-

ical information necessary for decision-making. Thus, in this section we introduce our graph convolutional reinforcement learning (GCRL) based task-allocation agent. It consists of the encoding part and decision-making part as shown in Figure 4.2.

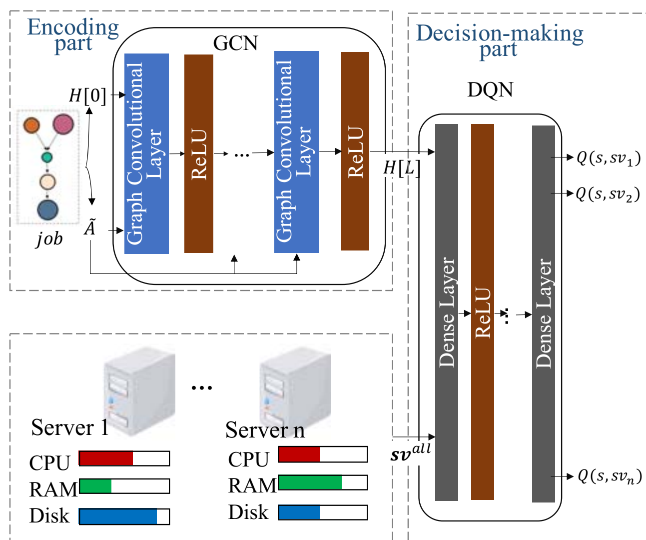


Figure 4.2: Graph convolutional reinforcement learning algorithm.

Encoding Part: First, we formulate the dependent tasks of a job as a DAG where each task of a job is a node and each dependency is represented as a directed edge. Node feature is represented as a feature vector that includes the workload and resource requirements of CPU, RAM and Disk, and each directed edge represents the dependency relationship in task execution.

To describe task features, we construct feature matrix $X \in \mathbb{R}^{|\mathbf{ta}| \times |\mathbf{sv}^i|}$ where $|\mathbf{ta}|$ is the number of nodes and $|\mathbf{sv}^i|$ is node feature. Thus, X includes all of the node's information where each row represents the features of a task. We use adjacency matrix A to describe the dependency relationship whose

element a_{ij} at intersection of the i-th row and j-th column is one, if task j's execution depends on the completion of task i (the element is 0, otherwise). Then, we use degree matrix D to denote the importance of each dependency where the more nodes it connects with, the less import it is; its element d_{ii} represents the number of node i's neighbors. Let us consider the job at $t = 1$ in Figure 4.1, which corresponds to 5 dependent tasks and each task feature has 4 dimensions.

Feature matrix X, adjacency matrix A and degree matrix D are given by

$$X = \begin{bmatrix} wok_1 & Req_1^{CPU} & Req_1^{RAM} & Req_1^{Disk} \\ wok_2 & Req_2^{CPU} & Req_2^{RAM} & Req_2^{Disk} \\ wok_3 & Req_3^{CPU} & Req_3^{RAM} & Req_3^{Disk} \\ wok_4 & Req_4^{CPU} & Req_4^{RAM} & Req_4^{Disk} \\ wok_5 & Req_5^{CPU} & Req_5^{RAM} & Req_5^{Disk} \end{bmatrix}.$$

$$A = \begin{bmatrix} a_{11} & . & . & . & a_{15} \\ . & . & & & . \\ . & & . & & . \\ . & & & . & . \\ a_{51} & . & . & . & a_{55} \end{bmatrix} D = \begin{bmatrix} d_{11} & 0 & 0 & 0 & 0 \\ 0 & d_{22} & 0 & 0 & 0 \\ 0 & 0 & d_{33} & 0 & 0 \\ 0 & 0 & 0 & d_{44} & 0 \\ 0 & 0 & 0 & 0 & d_{55} \end{bmatrix}.$$

Further, we construct $\widetilde{A}$ which includes dependency information between tasks with different importance.

$$\widetilde{A} = D^{\frac{1}{2}}(D - A)D^{\frac{1}{2}}. \tag{4.9}$$

The specific calculation process in one convolutional layer is as follows.

We take H^0=X as the first GCN layer which includes each node's (task) own feature vector, and the latent variable of H^{l+1} is updated by the latent variable H^l of the previous layer. In each layer, after calculating $\widetilde{A}H^l$, each node updates its feature vector based on its own and its dependent tasks' features. Then, $\widetilde{A}H^l$ is multiplied by parameter matrix W, which would be learned to suit the specific learning object. Moreover, in order to obtain a non-linear representation of node features, an activation layer is added; the ReLU function is used as the activation function in this section. Finally, we get latent variable H^{l+1} and use it to update the next layer. This GCN encoding process can be represented by the following equation.

$$H^{l+1} = \delta(\widetilde{A}H^l W^l), \tag{4.10}$$

where $H^l \in \mathbb{R}^{|\mathbf{ta}| \times d_l}$ is the latent variable in the l-th layer with d_l dimensions.

Although other GNN networks can be utilized for embedding dependent information, it is not the core of this section. What we focus on in this section is to propose a general framework for combining GNN and DRL algorithms to determine how to dynamically allocate dependent tasks in edge computing.

Decision-making Part: In this part, we first formulate the dynamic decision problem as a MDP. The dynamic characteristics of the problem arise from two parts: status of server's computation resources and jobs arriving at each step. Thus, the state is defined

$$s_t = \left[\mathbf{sv}_t^{all}, job_t\right] \tag{4.11}$$

where the first part $\mathbf{sv}_t^{all} = \left[\mathbf{sv}_t^1, ..., \mathbf{sv}_t^{|SV|}\right]$ includes the current resource sta-

tuses of all servers at step t; the second part job_t is the job arriving at step t.

As for state s_t, the task allocation strategy means to choose a server in SV to which job_t is allocated. The task allocation scheme at step t is regarded as an action in MDP, denoted as a_t, and all possible tasks allocation strategies correspond to the action set, $\mathscr{A} = \{sv_1, ..., sv_{|SV|}\}$. Moreover, action a must satisfy the servers' resource requirements. Specifically, job job_j must be accomplished by ensuring that the chosen server has enough computation resources. Obviously, the total computation resource requirement of all tasks scheduled to server sv_i cannot exceed the maximum computation resource of server sv_i. That is,

$$\begin{aligned} &\sum_j Req_j^{CPU} + CPU_t^i \leq CPU_{i.max}, \\ &\sum_j Req_j^{RAM} + RAM_t^i \leq RAM_{i.max}, \\ &\sum_j Req_j^{Disk} + Disk_t^i \leq Diski.max. \end{aligned} \tag{4.12}$$

Thus, we define a constrained action set $\mathscr{A}(s)$ for state s by $\mathscr{A}(s) = \{a_i | a_i \in \mathscr{A}$ satisfies Eq.(4.12)$\}$

DQN, which is proposed in [Mnih et al., 2015] where a neural network is used to approximate the value of $Q(s,a)$, is used as a decision-making framework. In DQN, the Q value is calculated by a deep neural network and the goal is to minimize the following loss equation.

$$L(\theta) = \mathbb{E}_{s,a\sim\pi}\left[\left(y_j - Q(s,a;\theta)\right)^2\right], \tag{4.13}$$

where

$$y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{j+1} \end{cases}$$

In this problem, the input of DQN consists of two parts: server status $\mathbf{sv}^{all}$ and embedding result H^L obtained from the above GCN embedding part which includes the high dimension information of the job. DQN calculates the state-action value $Q(s,a)$ for each action and chooses a server for job allocation. Based on the execution of the action, the server status would be altered and a new job would arrive, which corresponds to a state transition. We can obtain a reward of the transition to train the neural network with the aim of maximizing accumulated rewards based on Eq.(4.7) where reward is $cost_t$ at step t. GCRL is stated in Algorithm 2 in detail.

4.4.2 Case 2: Task Allocation with Multiple Jobs

The above proposed method has two strong assumptions: 1) only one job arrives at each step and 2) each job can be only allocated on one of the servers. Then, we extends it to loose these two assumptions and proposes a multi-graph convolutional reinforcement learning (MGCRL) for dependent task allocation with multiple jobs. Compared with GCRL for single job, we change the encoding part as Multi-GCN encoding part: it encodes multiple jobs for decision-making of task allocation, as shown in Figure 4.3

Although there is a similar work that combines GCN with DQN for dependent task allocation, they just use GCN to encode the jobs and take the encoding results as the input of DQN, where GCN parameters are not up-

ALGORITHM 2: Graph Convolutional Reinforcement Learning (GCRL) Algorithm

1 Initialize replay memory D_{rep}
2 Initialize action-value function Q with random weights
3 **for** *episode m=1, M* **do**
4   Generate initial state s_1
5   **for** *step t =1, T* **do**
6   Transform arriving job into a DAG and calculate its corresponding feature matrix X and $\widetilde{A}$ based on adjacency matrix A and degree matrix D.
7   Take X as H^0 and calculate H^L by GCN.
8   Randomly select an action a_t (server) from $\mathscr{A}(s_t)$ based on a ε-greedy policy
9   otherwise select $a_t = \max_a Q(s_t, a; \theta)$
10   Execute action a_t in edge computing system and observe cost $cost_t$ and transfer to the next state s_{t+1}
11   Set $s_{t+1} = s_t$, and store transition $(s_t, a_t, cost_t, s_{t+1})$ in D_{rep}
12   Sample random mini-batch of transitions $(s_j, a_j, cost_j, s_{j+1})$ from D_{rep}
13   Take $cost_j$ as r_j and calculate y_j based on the Eq. (4.13)
14   Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ to update θ
15   **end**
16 **end**

dated. Moreover, they deal with each job independently without considering the relationship of jobs. As a result, they use individual reward of allocating one task to train the neural network, which may be difficult to contribute a maximization of team reward. Different from the existing GCN with DQN framework, MGCRL has three originalities: 1) it accepts multiple graph information as inputs; 2) both the parameters of GCN and DQN can be updated in an end-to-end manner; 3) an additional reward is proposed to

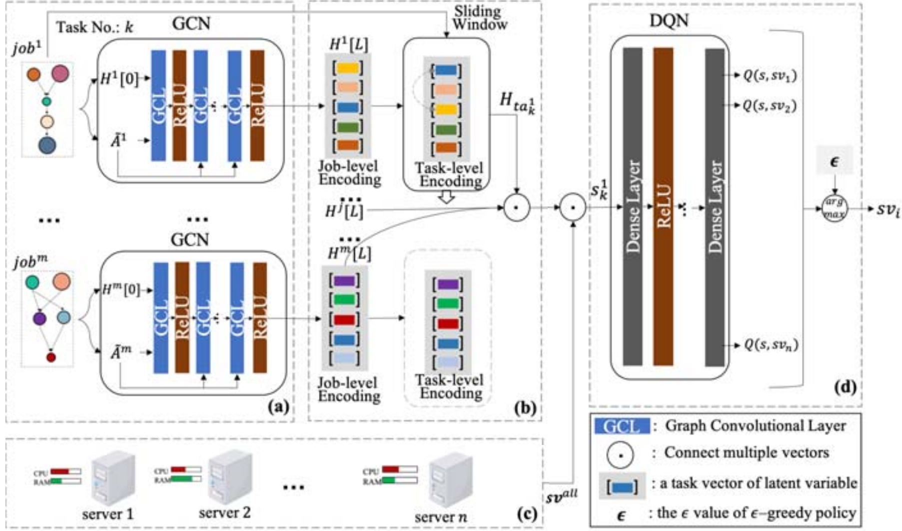


Figure 4.3: Multi-graph convolutional reinforcement learning algorithm.

evaluate each task allocation's influence to the other tasks.

Multi-GCN Encoding Part: In this part, we explain how Multi-GCN is used to encode the multiple jobs in detail. Each task feature vector in a job includes just its own information which is not enough for decision-making since the different dependency relationships greatly influence the delay cost. Thus, updating each task's feature vector requires its neighbors' (its dependent tasks) feature vector. Since GCN [Sperduti and Starita, 1997], a powerful tool to cope with graphic data, can deal with graphic information well, we use GCN to extract the task dependency relationships presented in a DAG as a latent variable with high dimensional information. Unlike the only one static graph considered in most GCN researches, in this thesis there are multiple graphs that dynamically change over time. Thus, we propose Multi-GCN to cope with multiple jobs.

First, as shown in the upper left part of Figure 4.3, we formulate each job as a DAG where each task of a job is a node and the dependencies are represented as directed edges. Then, we can construct two matrices to represent all the information of DAG. One is called as feature matrix $X^j \in \mathbb{R}^{|\mathbf{ta}^j| \times f}$ to represent task features in job^j where $|\mathbf{ta}^j|$ is the number of tasks in job^j and f is the number of task feature. Thus, each row of X^j represents the features of a task such as workload, resource requirements of CPU and RAM in job^j and X^j includes the information of all the task features. We take X^j as the input of GCN, i.e., $H^j[0]$=X^j. The other is adjacency matrix A^j is used to describe the dependency relationship whose element at intersection of the k-th row l-th column is one if task ta^j_l execution depends on the completion of task ta^j_k (the element is 0, otherwise).

Thus, Multi-GCN yields the embedding result $H^j[L]$ for each job^j based on GCN embedding which includes the high dimension information of the job^j (L is the layer number of GCN).

DQN Decision-making Part: Based on the above part, we can obtain each job^j's GCN encoding results $H^j[L]$. Then, we use a sliding window to choose each job. We take task No. k inside a job j as input of the sliding window and obtain each task embedding result. For each task ta^j_k in job^j, we change the position of $H^j[L]$'s first element $H^j[L]_0$ with $H^j[L]$'s k-th element $H^j[L]_k$ and take it as task-level encoding $H_{ta^j_k}$ of task ta^j_k, which is denoted as $H_{ta^j_k}$=$[H^j[L]_k, H^j[L]_1, ..., H^j[L]_{k-1}, H^j[L]_0, H^j[L]_{k+1}, ..., H^j[L]_{|\mathbf{ta}^j|}]$. Then, for each task ta^j_k's allocation, we take the following three parts as a input s^j_k of DQN:

1) the task's encoding result $H_{ta^j_k}$ which includes own task information and

also relevant task information;

2) the encoding results of other jobs $H^{-j} = [H^0[L], ..., H^{j-1}[L], H^{j+1}[L], ..., H^{|\mathbf{job}|}[L]]$;

3) the vector of all server statuses $\mathbf{sv}^{all}$.

We denote it as $s_k^j = [H_{ta_k^j}, H^{-j}, \mathbf{sv}^{all}]$. Then, as shown in part (d) of Figure 4.3, DQN calculates the state-action value $Q(s_k^j, a_k^j)$ for each action (server). We choose a server to allocate task ta_k^j based on ε-greedy policy. Specifically, it chooses the action which has a maximum Q-value with the probability of $1-\varepsilon$ or randomly chooses an action with the probability of ε.

After allocating a task to a server, a new task No. k is input to sliding window to repeat the above process. The sliding window moves to the next job until all the tasks of the current job have been allocated. Finally, we can choose the severs for all the tasks in the arriving multiple jobs and take it as a joint action. Based on the execution of the joint action, the server status would be altered and several new jobs would arrive, which corresponds to a state transition. The goal of DQN is to minimize the following loss equation.

$$\begin{aligned} L(\theta) = \mathbb{E}_{s_k^j, a_k^j \sim \pi} \Big[\big(cost_k^j + \gamma max_{a_k^{j'}} Q(s_k^j[t+1], a_k^{j'}; \theta) \\ - Q(s_k^j[t], a_k^j[t]; \theta) \big)^2 \Big], \end{aligned} \tag{4.14}$$

where θ are the parameters of DQN.

As a conclusion, our proposed method can be summarized as follows. It majorly consists of two parts: Multi-GCN Encoding Part (a)(b)(c) and DQN Decision-making Part (d). (a): Multi-GCN takes multiple jobs' DAGs as an

input and outputs each job encoding result. (b): There is a sliding window used to choose a job, then takes the task No. as an input and outputs the corresponding task encoding result. Then, we connect it with other job encoding results. (c): The statuses of all servers are connected with the output of (b) as a input for DQN. (d): For each task, it outputs corresponding Q-values for allocating each server and chooses a server to allocate the task based on ε-greedy policy. After allocating a task to a server, a new Task No. is input to sliding window to repeat the above process (The sliding window moves to the next job until all the tasks of the current job have been allocated)

Then, we introduce how to calculate difference reward for each individual reward. As for the individual reward $cost_k^j = \frac{wok_k^j}{rf^i}$ after allocating a task on the server i, the $r = \frac{1}{|n^i|}$ is inverse ratio of the number of the tasks are allocated on server i. The more tasks are allocated on server i, the slower the tasks would be. Thus, it would be better for the whole system where the tasks with high workloads are allocated on the servers whose computation rates are high. Thus, we define a difference reward $diff_k^j$ for task ta_k^j allocation on server i as

$$diff_k^j = \sum_{k' \in n^i} cost_{k'}^j - \sum_{k' \in n^i \setminus k} cost_{k'}^j \qquad (4.15)$$

where $n^i \setminus k$ is the set of n^i without k; $diff_k^j$ means the cost value if task ta_k^j is not allocated on server i. Thus, we update reward $cost_k^j$ as $cost_k^j \leftarrow cost_k^j - diff_k^j$. Finally, we use the reward updated by the above equation to train the whole MGCRL neural network to achieve the optimal goal.

4.5 Evaluation

Case 1: GCRL for Task Allocation with Single Job

This section evaluates the performance of our GCRL based task-allocation agent by comparing it with the benchmarks of Q-learning (QL) [Watkins and Dayan, 1992] and Deep Q-network (DQN) [Chen et al., 2018b] for dependent task allocation.

Evaluation Setting

We take an edge computing system composed of 10 edge servers whose parameters are based on published values of edge servers like RaspberryPi. As for the setting of tasks, the values of tasks' parameters such as number of arriving tasks $|\mathbf{ta}|$, task workload wok_j, and task dependency **dep** are all randomly generated in accordance with a probability distribution. Based on the random property, we take 100 episodes as one *round* and use the average of costs in one round to compare the performances of the different algorithms. We set the same hyper-parameters of RL for all algorithms with α=0.01 and γ=0.9.

Performance Evaluation

We first check the evaluation result of our GCRL in minimizing the sum of energy and delay costs by comparing with QL and DQN.

We ran the following simulations for 5,000 episodes, which correspond to 50 rounds, and show the results in Figure 4.4. Since QL's effectiveness is poor in a large state space, it cannot learn during training and is similar to the performance of a random policy. DQN has better performance than QL since it can cope with large state spaces. However, DQN would not

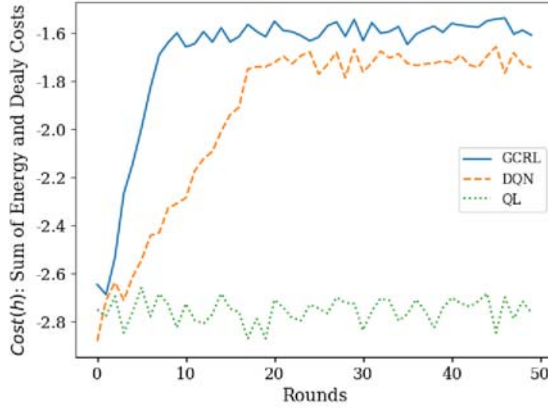


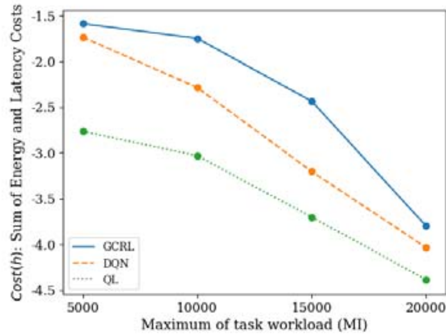
Figure 4.4: Comparing performances of GCRL, QL and DQN algorithms in minimizing the sum of energy and delay costs.

be effective since it fails to cope with the dependency information. Our proposal, the GCRL algorithm, well copes with the dependency information and so achieves the best performance in terms of minimizing energy and delay costs. A comparison of the results finds that the GCRL algorithm can decrease the total cost by around 20% relative to DQN, because it can cope with both the large state space and the dependency information of tasks.

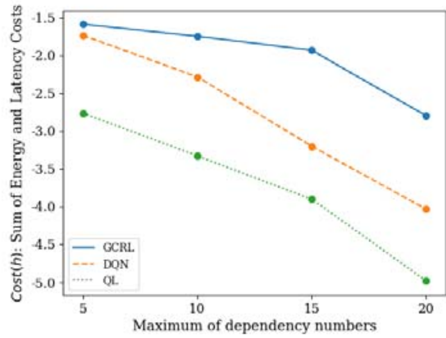
Parameter Analysis

In order to evaluate the performances of these algorithms comprehensively, we analyzed their performances in different parameter settings. The parameters included the maximum task workload, maximum number of dependent tasks in arriving job, and maximum task RAM requirements.

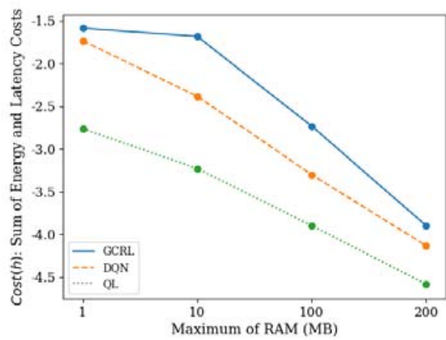
i) The Effect of Task Workload This experiment checks the performance of GCRL algorithm with different task workloads whose unit of measure is Million Instructions (MI). The maximum task number of a job, n_{max}, is fixed



(a) Impact of task workload.



(b) Impact of task dependency number.



(c) Impact of task RAM.

Figure 4.5: The impacts of task workload, task dependency number and task RAM.

at 10 and maximum wok_{max} of task workload is changed from 5000 MI to 20000MI in steps of 5000MI. The results are shown in Figure 4.5 (a); since the server computation resource is limited, a larger task workload setting requires more servers to be invoked, which corresponds to a smaller available action space. Thus, the advantage of the proposed algorithm decreases with the increase of task workload since the difference between optimal policy and other policies is not obvious in a small available action space.

ii) The Effect of Task Dependency This experiment checks the performance of our GCRL algorithm with different numbers of dependent tasks. We keep the maximum task number n_{max} at 10 and maximum wok_{max} of task workload at 5000. We changed the number of dependency relationships in **dep** from 5 to 20 in steps of 5. The results, shown in Figure 4.5 (b), confirm that our algorithm offers lower costs than the other algorithms under different dependency relationship number. When the dependent task number is small, we can see DQN has similar performance with our GCRL algorithm. This is because task dependency has little impact on the cost if the tasks are nearly independent of each other. However, the advantage of the proposed algorithm increases with the increase of dependency relationship number, since a larger dependency relationship number has a stronger impact on the cost.

iii) The Effect of Task RAM This experiment checks the performance of our GCRL algorithm with different task RAM requirements. The maximum task number n_{max} and maximum wok_{max} of task workload are fixed at 10 at 5000MI, respectively. The task RAM requirements are changed among 1MB, 10MB, 100MB, 200MB, respectively. The results, shown in Figure 4.5 (c), confirm that GCRL offers lower user cost than the other al-

gorithms under different task RAM requirements. This advantage decreases with the increase of RAM resource requirement, since tasks with huge RAM requirements would require more servers to run, which corresponds to a very limited action space.

Case 2: GCRL for Task Allocation with Multiple Jobs

Evaluation Setting

This section uses a real-world dataset to evaluate our proposed method together with other baseline algorithms. We use part of the dataset from Aliyun called Cluster-trace-v2018 which includes about 4000 servers and a periods of 8 days [Guo et al., 2019]. This dataset includes many dependent tasks from the real-world which has been used in many dependent task allocation studies for evaluation purposes [Liu et al., 2019][Tang et al., 2020]. There are two types of tasks in the dataset, one is dependent-task which corresponds to a DAG, the other is independent-task which can be regarded as a special case of DAG with only one node. Each task has a number of instances which corresponds to the workload in task definition and has resource requirements of RAM and CPU.

For the arriving time of each job, we refer to [Tang et al., 2020] to set a random number between [0, 100] where each number corresponds to one step. Thus, the jobs that are assigned the same number are deemed to arrive at the same time which corresponds to the multiple jobs allocation problem.

Performance Evaluation

In this experiment, we evaluate the performance of our MGCRL algorithm with different baseline algorithms of dependent task allocation, which are

stated as follows.

- **MGCRL** (Multi-graph convolutional reinforcement learning): It includes Multi-GCN encoding part and DQN decision-making part. Multi-GCN encoding part can deal with the information of multiple jobs and output each task corresponding embedding information successively. Then, we take each task's embedding result from Multi-GCN part and other information as inputs of DQN to choose a server for each task. Based on the delay cost obtained, both the parameters of Multi-GCN and DQN are updated together.

- **DTO** (Dependent Task offloading) [Tang et al., 2020]: It includes GCN model and DQN model where GCN model can output the task-level encoding results and take each task's encoding result as the input of DQN to output the server to be allocated without considering other arriving jobs. Moreover, the parameters of GCN are not updated, only the parameters of DQN are updated.

- **TDQ** (Traditional Deep Q-learning) [Chen et al., 2018b][Mnih et al., 2015]: It includes several fully-connectd layers which transfer the input of state to the action values. Although it can be used for dependent task allocation, it may not cope with dependency information well. That is because a job consisting of dependent tasks is essentially graphic data and TDQ usually suits only non-graphic data like one-dimension vectors

- **TQL** (Traditional Q-learning) [Watkins and Dayan, 1992]: It includes a Q-table to record the Q-values for each state-action pair and updates the Q-values based on Q-learning algorithm.

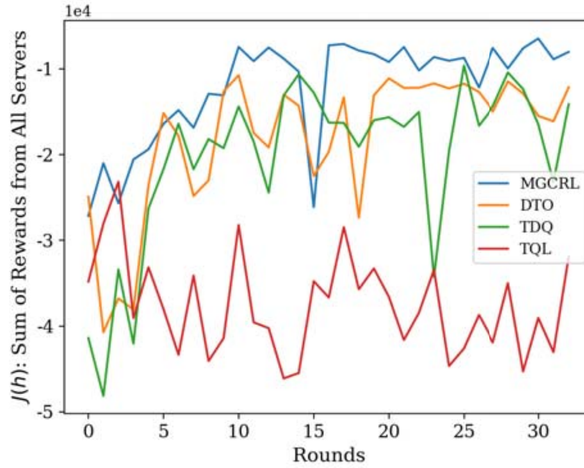


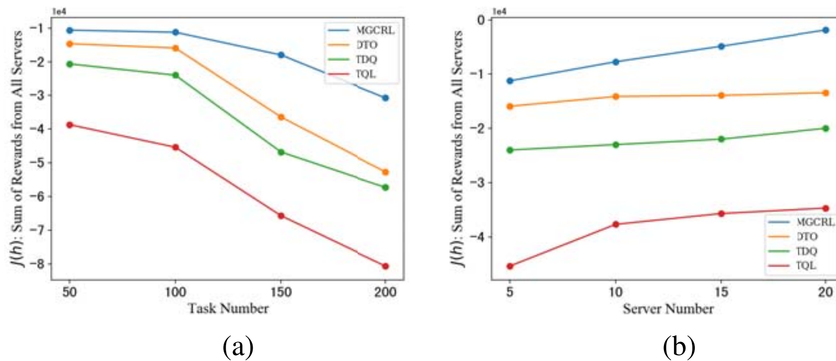
Figure 4.6: Comparing the performances in minimizing delay cost via MGCRL, DTO, TDQ and TQL algorithms.

We choose 100 tasks from the dataset and set 5 edge servers, then run 500 episodes for each method. Since the steps of the arriving jobs are assigned randomly in each episode, we take the average of 15 episodes' rewards as one round learning results which are shown in Figure 4.6. TQL can not learn during the training, since the dynamics of server statuses and arriving jobs bring a large state space which incurs it hard to maintain a Q-table. TDQ has better performance than TQL since it can cope with large state spaces. However, TDQ would not be effective since it fails to cope with the dependency information well. DTO which combines GCN and DQN has a better performance than TDQ, since it can deal with dependency information well. However, DTO can only encode one job information where the information of other jobs is ignored. Our proposal, the MGCRL algorithm, can cope with the dependency information with multiple jobs well and so achieved the best performance in terms of minimizing delay costs. Specifically, the

final sum of rewards in one episode of MGCRL corresponds to $1e4$ seconds, and the final sum of rewards in one episode of DTO corresponds to $1.5e4$ seconds which breaks down delay costs to around $0.5e4$ seconds totally; it indicates that the delay cost of executing one task can be decreased 50 seconds averagely by using our proposed method. In short, MGCRL has an overall best performance than all baselines and outperforms the baseline approach DTO around 30% on total delay cost.

Parameter Analysis

In this experiment, we evaluate the performance of our MGCRL algorithm with different number of tasks and number of servers. First, we keep the number of servers as 5 and change the number of tasks from 50 to 200 in steps of 50. The bigger of the task number is, more tasks are required to be allocated on the servers. We run 500 episodes and take the average of the last 50 episodes' rewards as the learning results. The results are shown in



(a) (b)

Figure 4.7: Parameter analysis of task number and server number.

Figure 4.7 (a), the results confirm that our algorithm offers lower delay costs than the other algorithms under different task number. When the task num-

ber is small, we can see DTO has similar performance with our MGCRL algorithm. This is because task number incurs a little influence to the cost. As the increase of task number, the advantage of our proposed algorithm increases since our proposed method can cope with the large scale of tasks well.

Then, we evaluate the performance of our MGCRL algorithm with different server numbers. We keep the maximum of task number at 100 and change server number of task workload from 5 to 20 in steps of 5. We run 500 episodes and take the average of the last 50 episodes' delay costs as the learning results. The results are shown in Figure 4.7 (b), since the bigger of server number is, the less tasks would be allocated on each server. This will decrease the overall delay cost, since the less number of tasks allocated to a server corresponds to a faster computation rate. As the increase of server number, the advantage of proposed algorithm increases since it corresponds to a bigger action space where a better task allocation can be learned. The results confirm that our algorithm offers lower delay costs than the other algorithms under different server number.

4.6 Summary

In Chapter 3, we focused on high-workloads in task allocation which can be regarded as a task's internal feature. In this chapter, we consider the relationship among tasks where the execution of one task will influence the conduct of other tasks. We call this problem dependent task allocation. Unlike existing studies which focus on task information itself and ignore the server status, we study this problem by considering both dynamic and

dependent features.

Our solution is to propose graph convolutional reinforcement learning-based task allocation methods for two cases: 1) a single job can be allocated to just one server; 2) multiple jobs can be allocated to multiple servers. Specifically, a GCN module is used to embed the dependency information of the tasks. A RL module is used to process the decision-making steps needed for task allocation. Experiments confirm the performance of our proposed methods. We consider that models trained by our proposed methods are indispensable for the development of dependent task allocation in edge cloud computing with strong dynamic features. This work has been published in [Ding et al., 2021].

Chapter 5

Multiagent Reinforcement Learning (MARL) for Distributed Task Allocation

In this chapter, we focus on the third case illustrated in Chapter 1 which is distributed task allocation in cooperative edge cloud computing. We propose a multiagent reinforcement learning based method to solve this problem.

5.1 Introduction

In distributed task allocation problem, edge servers are distributed among different areas. Each edge server observes its own status, such as CPU occupancy rate to decide how to perform the tasks such as offloading the tasks or performing the tasks locally. Based on those allocation decisions, each edge

server's status would be altered. Different task allocation decisions usually correspond to different performances such as delay cost, energy cost and accuracy of trained model. Then, how to offload the tasks with the aim of optimizing a certain object is an important research topic in distributed edge cloud computing [Chen et al., 2015][Liu et al., 2020b][Chen et al., 2018b].

Many studies have examined the distributed task allocation problem in edge cloud computing, however they usually consider the problem at a user level where each user owns an edge server and assume each edge server is self-interested, which corresponds to a non-cooperative setting. Specifically, each user only desires to maximize its own interest, which may cause a conflict between them. For instance, if many edge servers try to offload their tasks to cloud servers with limited channel resources at the same time, the offload rate will decrease due to network congestion. With the high development of smart communities such as smart factory, smart campus and smart hospital, the edge servers are usually owned by an organization like a technology corporation rather than users [Nishi, 2018][Donovan et al., 2017][Abdellatif et al., 2019]. Thus, its goal is to maximize the team reward that emphasizes overall interest of all edge cloud servers rather than each edge server's own interest, which corresponds to a cooperative setting. However, the existing studies are not suitable for this kind of cooperative scenario, making further research essential.

In this chapter, we study a new problem called the distributed task allocation in cooperative edge cloud computing. This problem raises two issues: 1) each edge server's status dynamically changes and task arrival is uncertain. 2) each edge server can observe only its own status, which corresponds to a local observation, which makes it hard to achieve cooperation. To cope with

these issues, we formulate this problem as a decentralized partially observable Markov decision process (Dec-POMDP) which is a classical model that copes with dynamic decision problems with partial observations. Then, we apply a multiagent reinforcement learning (MARL) algorithm called value decomposition network (VDN), and propose a VDN-based task allocation algorithm to solve it. In our proposed method, a team value function is used to evaluate team interest and it is then divided into individual value functions for each edge server (regarded as an agent). Each agent updates its individual value function in the direction that can maximize the team reward. Then, we consider two classical cases of the distributed task allocation problems: case 1) there exist multiple tasks which can be performed by one server; case 2) there exists only one single task which cannot be performed by one server. As for case 1), we choose part of a real Google datacenter dataset to conduct evaluations to verify the effectiveness of our proposed algorithm in a comparison with other existing algorithms. As for case 2), we consider a decentralized federated learning as an example where multiple edge servers jointly learn one model. We evaluate our proposed method by training a image classification model on three classical image datasets.

5.2 An Illustrative Example of Distributed Task Allocation

In this section, we use the motivating scenario of the smart hospital to illustrate the problem of distributed task allocation in cooperative edge cloud computing. As shown in Figure 5.1, there are several inpatient wards in different areas and each ward is equipped with an edge server. Each edge

server takes charge of performing the IoT tasks demanded by its inpatient ward. For instance, analyzing patient's body data to monitor his/her health condition can be regarded as a health condition monitoring task. If five patients are in one ward at a moment, a task queue consisting of the five health condition monitoring tasks will arrive at the corresponding edge server. The edge server can decide which tasks are executed locally and which tasks are offloaded to the cloud servers.

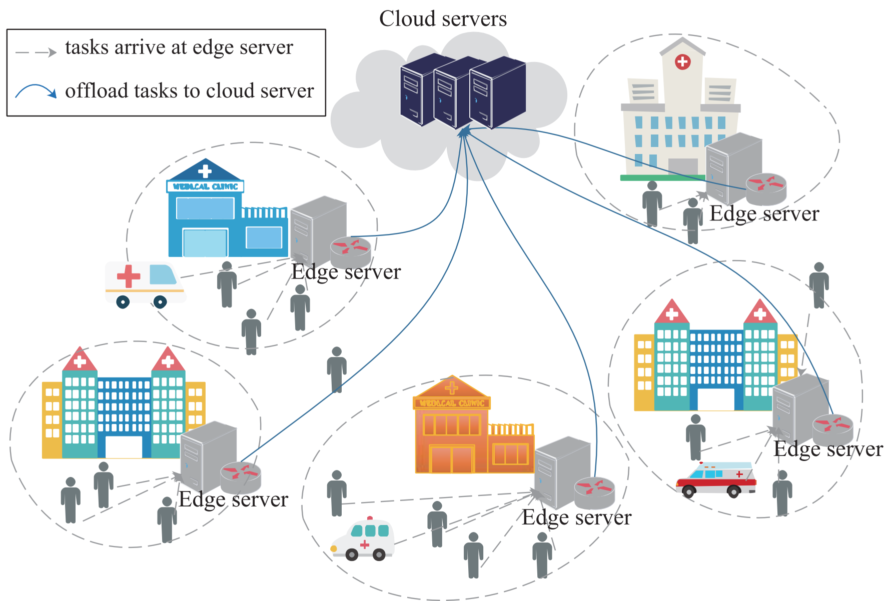


Figure 5.1: A distributed edge cloud computing system in a smart hospital.

Since cloud servers usually have more powerful computation capacity than edge servers, it will cost less time if the tasks are performed by cloud servers. However, allocation time would become significant if many edge servers choose to offload their tasks at the same time. This is because the allocation rate usually has an inverse relationship with the number of offloaded tasks. Since all edge servers are owned by one organization, all the

edge servers are in cooperative relationships and try to optimize the team interest like minimizing the sum of all IoT task delay costs rather than its own interest. If such cooperation is not achieved well, delay in performing tasks may cause a bad user experience for patients. However, it is difficult to maximize the team interest due to the issues stated in Section 5.1, which requires further to study.

5.3 Distributed Task Allocation Problem

In this section, we illustrate the distributed task allocation in cooperative edge cloud computing of both case 1) and case 2) stated in section 5.1.

Case 1: Independently Performing Multiple Tasks

As for the case 1 where each task can be performed by only one server, one classical example is task offloading. We consider the distributed edge cloud computing system shown in Figure 5.2, which includes several edge servers distributed among various areas and a cloud server cluster. We regard multiple cloud servers as one single server entity since this section does not focus on how the tasks are performed in the cloud servers.

At each step, each edge server would be accessed by a task queue consisting of several tasks and the edge server decides which tasks to execute locally and which tasks to offload to the cloud servers. Server status like CPU occupancy rate would be altered by allocating the tasks and also influences task performance attributes such as latency and energy consumption. We use cost to evaluate the performance numerically, and consider two types of costs: delay cost and energy cost in this section. The delay cost consists of

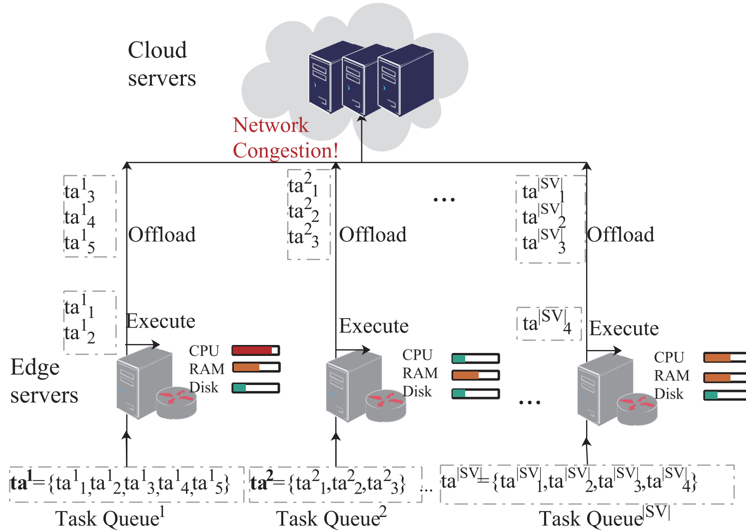


Figure 5.2: Task offloading in a distributed edge cloud computing system.

computation delay and transmission delay, and the energy cost is incurred by performing/uploading the tasks. We consider the goal of how to minimize team cost in order to ensure all tasks in all edge servers have good performance. This means that all edge servers should cooperate in deciding task allocation rather than considering just their own interests.

While it seems that each edge can easily acquire status information of the other servers by sending Hello packets, we assume each edge server has only a partial observation in which each edge server cannot observe other server status information. The reasons for this setting are stated as follows: 1) Since sending a packet takes a certain time, collecting the status information of all edge servers will cost more time that exponentially increases with the number of edge servers, which incurs an additional delay cost. This delay cost would become significant, especially if network congestion happens [Chen and Wang, 2020]. 2) As each edge server must acquire global

information for all edge servers to do decision-making, the size of observation also exponentially increases with the number of edge servers. This would make learning problematic when the number of edge servers is large [Chen and Wang, 2020]. 3) Moreover, system robustness is degraded since each edge server cannot make a decision until acquiring all edge server information. If one server fails to send its own information, all other edge servers are unable to make their decisions, which paralyzes the whole edge cloud computing system. Thus, we consider the problem in a decentralized way with partial observations, which is defined as follows.

Server: We use the definition of server set in Chapter 4. In order to distinguish the edge server and cloud server, we change the indexes of servers to $SV = \{sv^0, sv^1, sv^2, ..., sv^{|SV|}\}$ where we regard many cloud servers as a single entity with infinite computational resources denoted as sv^0, and $sv^i (i > 0)$ represents the i-th edge server. Each server's parameters are denoted by the vector defined in Eq.(4.4)

Task: At each step, a task queue consisting of several tasks might arrive at edge server sv^i. Since this problem includes the task offloading process, we extend the task definition in Eq.(4.2) by adding data size b^i_j, which is denoted as $\mathbf{ta}^i$=$\{ta^i_1, ta^i_2, ..., ta^i_{|\mathbf{ta}^i|}\}$ with

$$ta^i_j = [work^i_j, b^i_j, Req^{i.CPU}_j, Req^{i.RAM}_j, Req^{i.Disk}_j], \tag{5.1}$$

where $work^i_j$ is task j's *workload* denoting the CPU cycles required to be performed, b^i_j is task j's *data size* whose unit of measure is *bit*. Although b^i_j can be independent with $work^i_j$, we assume b^i_j has a linear relationship with $work^i_j$ in this section, i.e., $b^i_j = c * work^i_j$, where c is a constant whose

unit of measure is bit/cyc. $Req_j^{i,CPU}$, $Req_j^{i,RAM}$ and $Req_j^{i,Disk}$ are task j's requirements for CPU, RAM and Disk resources.

Each task would be performed either on edge server or offloaded to the cloud servers. Thus, we refer to some related work to set the costs corresponding to the parts on which they are executed. Although we have defined the energy cost and delay cost in Chapter 4, those of edge end and cloud end should be distinguished as follows.

Computing at Edge Server: First, we consider the case when the task is performed on edge servers and refer to [Chen et al., 2015] to set the following equations. The computation time of the task ta_j^i is given by

$$t_j^i = \frac{work_j^i}{f^i}, \tag{5.2}$$

where the bigger the f^i is, the less time t_j^i is. The corresponding computation energy is given by

$$e_j^i = ec^i work_j^i, \tag{5.3}$$

where the bigger the $work_j^i$ is, the more energy it costs. Thus, the sum cost of computation time and energy at the edge servers can be calculated by

$$C_{j,edge}^i = \omega_t t_j^i + \omega_e e_j^i, \tag{5.4}$$

where $\omega_t, \omega_e \in [0,1]$ denote the weighting parameters of delay and energy ($\omega_t + \omega_e = 1$).

Computing at Cloud Server: Similarly, we can calculate the cost while the task is offloaded to the cloud servers. We consider that a base-station is used

to establish wireless communication channel between the edge servers and the cloud servers [López-Pérez et al., 2012]. Since all edge servers belong to one organization, we assume they share one wireless channel. Then, from [Chen et al., 2015][Liu et al., 2020b][Chen et al., 2018b] offload rate $ur(Up)$ is calculated by

$$\begin{aligned} ur(Up) &= w\log_2\left(1+\frac{qg}{\omega+qg\sum_{i\neq 0}|Up^i|}\right) \\ &= w\log_2\left(1+\frac{1}{\frac{\omega}{qg}+\sum_{i\neq 0}|Up^i|}\right) \\ &= w\log_2\left(1+\frac{1}{l+\sum_{i\neq 0}|Up^i|}\right), \end{aligned} \tag{5.5}$$

where w is the channel bandwidth whose unit of measure is bit/s, $\log_2\left(1+\frac{qg}{\omega+qg\sum_{i\neq 0}|Up^i|}\right)$ denotes the partition of the channel bandwidth that can be used (dimensionless) where the more tasks are uploaded, the smaller this value is, q is the edge server's transmission power which is determined by the wireless base-station according to some power control algorithms such as [Xiao et al., 2003][Chiang et al., 2008], g denotes the channel gain between the edge server and the base-station, $\sum_{i\neq 0}|Up^i|$ is the number of tasks being offloaded from all edge servers, Up^i is the set of tasks being offloaded from server i, Up is the set including all tasks offloaded, ω is the background noise power. Since ω, q and g are constants, we denote $l=\frac{\omega}{qg}$ as the channel coefficient. Thus, the unit of measure of upload rate $ur(Up)$ is bit/s. For instance, let us consider the case where $w=10bit/s$ and $l=0$. Then, at a step, if only one task is uploaded, the upload rate will be $10log_2(2)=10bit/s$. If two tasks are uploaded at the same time, the upload rate will be $10log_2(3/2)=5.8bit/s$

Correspondingly, the allocation time of task $t^i_{j,off} \in Up^i$ is given by

$$t^i_{j,off}(Up) = \frac{b^i_j}{ur(Up)}. \tag{5.6}$$

After the tasks are offloaded to cloud server sv^0, its corresponding calculation time is given by

$$t^i_{j,exe} = \frac{work^i_j}{f^0}, \tag{5.7}$$

where f^0 is the computation rate of cloud server sv^0. Since in most IoT environments task arrival might be sparse, this section assumes that the cloud servers are accessed using in on-demand manner where the user only pays when the cloud servers are in running. Then, the more task workloads (CPU workloads) are uploaded to the cloud servers, the longer usage-time and a higher cost (e.g., fee) would be. Since the energy cost increases in proportion to CPU workload, the evaluations in this section use energy cost. Specially, the computation energy of executing task t^i_j at cloud server is defined by

$$e^i_{j,exe} = ec^0 work^i_j. \tag{5.8}$$

From [Chen et al., 2015], the energy of uploading task t^i_j to the cloud servers is

$$e^i_{j,up} = \frac{qb^i_j}{ur(Up)}. \tag{5.9}$$

The sum cost of computation time and energy for performing task ta^i_j on cloud server sv^0 can be calculated by

$$C^i_{j,cloud}(Up) = \omega_t(t^i_{j,off}(Up) + t^i_{j,exe}) + \omega_e(e^i_{j,exe} + e^i_{j,up}). \tag{5.10}$$

Then, at one step the total cost of performing the tasks from all servers can be calculated and is denoted as $Cost$. In this section, the following three cases are considered respectively.

Latency-sensitive Case: only delay cost is desired to be minimized. Then, the total delay cost of both edge and cloud servers can be calculated by

$$C_{time} = \sum_{i=1}^{|SV|} \Big[\sum_{j \notin Up^i} t_j^i + \sum_{j \in Up^i} (t_{j,off}^i(Up) + t_{j,exe}^i) \Big]. \tag{5.11}$$

Thus, we have $Cost = C_{time}$ in latency-sensitive case.

Energy-sensitive Case: only energy cost is desired to be minimized. Then, the total energy cost of both edge and cloud servers can be calculated by

$$C_{energy} = \sum_{i=1}^{|SV|} \Big[\sum_{j \notin Up^i} e_j^i + \sum_{j \in Up^i} (e_{j,exe}^i + e_{j,up}^i) \Big]. \tag{5.12}$$

Thus, we have $Cost = C_{energy}$ in energy-sensitive case.

Balance Case: the sum of delay and energy cost is desired to be minimized, where $Cost$ is defined based on Eqs. (5.11) and (5.12).

$$Cost = \omega_t \frac{C_{time} - C_{time}^{min}}{C_{time}^{max} - C_{time}^{min}} + \omega_e \frac{C_{energy} - C_{energy}^{min}}{C_{energy}^{max} - C_{energy}^{min}}, \tag{5.13}$$

where $C_{time}^{max}, C_{time}^{min}, C_{energy}^{max}$ and C_{energy}^{min} represent the maximum/minimum values of time and energy, separately.

Moreover, the tasks must be accomplished by providing enough computation resources to the edge server. Obviously the total computation resource requirements of all tasks executed at edge server sv^i cannot exceed the max-

imum computation resource of server sv^i. That is,

$$\begin{aligned}
&\sum_{j \notin Up^i} Req_j^{i,CPU} + CPU^i \leq CPU_{max}^i, \\
&\sum_{j \notin Up^i} Req_j^{i,RAM} + RAM^i \leq RAM_{max}^i, \\
&\sum_{j \notin Up^i} Req_j^{i,Disk} + Disk^i \leq Disk_{max}^i.
\end{aligned} \tag{5.14}$$

Thus, the goal is to minimize the discounted cost summation under a period with certain length T, i.e., $\sum_{t=1}^{T} \gamma^t Cost[t]$, where γ is a discount factor denoting the importance of future cost. The goal is to minimize the discounted cost summation of all servers in an episode with satisfying the constraints of computation resources, i.e.,

$$\begin{aligned}
&\min \sum_{t=1}^{T} \gamma^t Cost[t] \\
&\text{s.t. } Eq.(5.14)
\end{aligned} \tag{5.15}$$

Case 2: Jointly Performing One Task

As for the case 2 where each task cannot be performed by only one server, one classical example is federated learning (FL) where multiple edge servers jointly learn one model.

FL is initially proposed in the works [McMahan et al., 2017][Konečnỳ et al., 2016] where one shared model can be learned jointly by distributed mobile devices by aggregating locally-computed update results [Zhang et al., 2020]. It has been studied well which can be classified into various types with regarding to different dimensions such as data partitioning methods,

communication architectures, and scales of federation where many survey papers have discussed it [Li et al., 2021] [Li et al., 2020]. In this section, we classify it according to whether a centralized manager server exists to discuss.

Centralized Federated Learning (CFL): One fundamental problem in FL is how to improve the convergence speed of model learning by optimally selecting client. Nishio et al.[Nishio and Yonetani, 2019] propose a client selection problem in a mobile edge computing with limited computational resources. Although more clients join to each round to learn would benefit to learning efficiency, some client with poor resources might slow down the overall learning speed. Thus, they formulate the client selection as a constraint optimal problem with maximizing the number of selected clients while satisfying some constraints such as deadline time. Wang et al.[Wang et al., 2020] address the non-independent and identically distributed problem where each client has a non-IID dataset. They propose a deep reinforcement learning based client selection method, FAVOR, to tackle the dynamically to select a subset of devices in each communication round to maximize validation accuracy. Zhang et al. [Qian Zhang et al., 2022] employee a MARL-based FL framework to process client selection with jointly optimizing model accuracy, processing latency and communication efficiency.

Decentralized Federated Learning (DFL): The above studies assume a centralized FL where a centralized parameter server exists to control all the client servers. However, in some cases like medical scenarios, it is difficult to search a central trusted server where every client may want to coordinate with it directly [Roy et al., 2019]. Further, depending on a centralized parameter might decrease robustness such like a fault occurs at the server will

break down the whole federated learning system. To solve this problem, [Roy et al., 2019] considers a random parameter server selection and once the selected parameter server is determined. The other servers will randomly take part in the current round learning. Lalitha et al. [Lalitha et al., 2018] consider a decentralized federated learning with the constraints where each client server can only communicate with its neighbor client servers. Hu et al.[Hu et al., 2019] address the problem that the network capacities between client servers are usually limited which hinders the transferring of learning model with large parameters. They propose a segmented gossip approach to split a model into a subset of segmentation, then those segmentation are transferred among nodes by a peer-to-peer manner.

Unlike the client selection problem has been studied well in CFL, the client selection has not been addressed well in DFL. Moreover, the parameter server selection needs also to be considered well. In this thesis, we consider to tackle this new problem and propose a novel algorithm to solve it.

Problem Definition

Based on the type of parameter server (manager), FL are usually divided into two types: cross-device setting and cross-silo setting. In cross-device setting, the parameter server is usually a server with powerful computation capacity like a cloud server. While the client servers are usually some mobile devices with limited computation resources like smartphones. In cross-silo setting, the client is usually assumed as a organization with powerful computation resources. In this thesis we consider a cross-silo setting where edge servers are assumed with powerful computations. Then, we define the client and manager selection problem in DFL as follows.

Server: In the cross-silo setting of DFL, a set $SV = \{sv^1, sv^2, ..., sv^{|SV|}\}$ of edge servers is given and we denote manager as $sv^m (m \in \{1, ..., |SV|\})$. Each server has the following parameters which is defined by the vector,

$$sv^i = [p^i, fa^i] \tag{5.16}$$

where p^i is all the parameters of the model, fa^i is the probability to failure on each round, which cannot be known previously.

Aggregating Delay: Once an edge server decides to join the round learning at round t, the aggregating delay $delay_ag^i[t]$ will be generated. As shown in the upper part of Figure 5.3, it consists of two parts: its own processing time $pt^i[t]$ for calculating local gradient. Once one client edge server is selected as a manager, then all the other servers are called client and all clients would communicate with the manager. Since all the edge servers are usually allocated to the distributed areas, their communication costs are usually different which are represented by the matrix of C whose elements c_{ij} represent the communication cost of edge server i and j. We denote communication cost from server sv^i to manager sv^m as $cc^i_m[t]$. Thus, given manager server sv^m, we have the aggregating delay for edge server i

$$delay_ag^i_m[t] = pt^i[t] + cc^i_m[t] \tag{5.17}$$

Moreover, the aggregating delay of the round depends on the latest finishing processing edge server, which is defined as

$$delay_ag[t] = max_{i \in SV}(delay_ag^i_m[t] * a^i[t]) \tag{5.18}$$

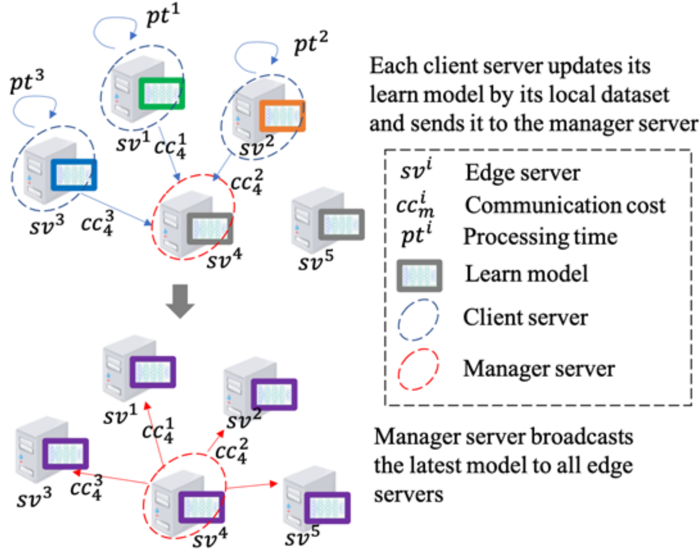


Figure 5.3: Decentralized federated learning process.

where $a^i[t] \in \{0,1\}$ denotes whether the server chooses to join in the current round of learning.

Broadcast Delay: Thus, given the manager sv^m the sum of communication costs is calculated as follows

$$delay_bro[t] = \begin{cases} max_{i \in SV}\{\sum_t c_{im}[t]\} & \text{if success} \\ d & \text{otherwise.} \end{cases} \tag{5.19}$$

where d is the deadline delay that can be waited for all client servers. It means that the next round automatically starts if all the client servers do not receive the latest model from manager.

Accuracy: After each round of learning, the change of parameters usually brings an alteration of accuracy $accu(p^i)$. We desire to obtain a high accuracy after finishing the learning.

Objective Function: Thus, the goal is to maximize the final accuracy $accu(p^i)$ while minimizing the delay costs.

$$\max \quad accu(p^i) - \sum_{t=1}^{T} \Big(delay_agg[t] + delay_bro[t] \Big) \tag{5.20}$$

In order to solve an optimal solution, there are two major issues needed to be solved. The first is how to make each agent learn to participant current round learning. For example, some edge servers have high communication cost with other edge servers. Thus, in initial steps, it should not join into the current learning round since it will slow down the whole all learning process. However, it might have some data that other servers cannot own. The model accuracy cannot improve further if that server does not take part in. Thus, it should take part in the final learning phase. The another major issue is that how to select the a manager once the group of servers is determined. Since two edge servers may have different communication costs, choosing the selected servers will influence the total communication costs.

5.4 Problem Formulation

Although the above two cases have differences such as objective functions, both of them can be formulated as Dec-POMDP, which is a classical model for formulating discrete time decision processes with partial observations. This is because they have two distinctive features: partial observations and dynamic changes.

In Dec-POMDP, each agent has a local observation and obtains a joint immediate reward depending on the results from all agents. Dec-POMDP can

be described as the tuple $< \mathcal{N}, \mathcal{S}, \mathcal{A}^i, \mathcal{O}^i, \mathcal{T}, r^i >$, where $\mathcal{N}$ is the set of agents, $\mathcal{S}$ is the set of states, $\mathcal{A}^i$ is the set of agent i's actions and collecting each agent action $a^i \in \mathcal{A}^i$ can form a joint action $a = a^1 \times ... \times a^{|\mathcal{N}|} \in \mathcal{A} = \mathcal{A}^1 \times ... \times \mathcal{A}^{|\mathcal{N}|}$, $\mathcal{O}^i$ is the set of agent i's observation, $\mathcal{T}$ is the state transition function and $\mathcal{T}(s,a,s')$ is the probability of the environment transitioning to state s' after taking joint action a under state s, i.e., $\mathcal{T}(s,a,s')$=$Prob(s'|s,a)$, r^i is the reward function for agent i and $r^i(s,a,s')$ is the reward obtained by agent i after taking joint action a under state s. How to formulate the cooperative task allocation problem as Dec-POMDP is stated as follows.

Observation: We regard each edge server as an agent, each agent's partial observation at step t consists from two parts: 1) server information $\mathbf{sv}^i[t]$ status, 2) task information $\mathbf{ta}^i[t]$. Thus, agent i's observation is defined as

$$o^i[t] = \Big[\mathbf{sv}^i[t], \mathbf{ta}^i[t] \Big], \tag{5.21}$$

where $\mathbf{sv}^i[t]$ is server i's status at step t. As for $\mathbf{ta}^i[t]$, in case 1 of task offloading, it is edge server i's task queue arriving at step t; in case 2 of decentralized federated learning, it is the status of trained model at step t.

State: By collecting each edge server's observation, we can form a state defined as

$$s[t] = \Big[o^1[t], ..., o^{|SV|}[t] \Big]. \tag{5.22}$$

Action: In case 1 of task offloading, each edge server would decide which tasks to execute locally or to offload to the cloud servers. Each task allocation set Up^i can be regarded as action $a^i \in \{0,1\}^{|\mathbf{ta}^i[t]|}$ where 0 represents

local computing and 1 represents allocation. For instance, as for task queue $\mathbf{ta}^i$=$\{ta^i_1, ta^i_2, ta^i_3, ta^i_4\}$, $a^i = [0,0,1,0]$ means that task ta^i_3 is offloaded to the cloud servers and the other three tasks are performed locally.

In case 2 of decentralized federated learning, each edge server decides whether to take part in the current round learning. The action $a^i \in \{0,1\}$ where 0 represents to not take part in current state and 1 represents to take part in current learning. Then, we define a joint action at step t which includes the actions from all edge servers as follows.

$$\mathbf{a}[t] = (a^1[t], ..., a^{|SV|}[t]). \tag{5.23}$$

Policy: We define policy function $\pi^i : \mathscr{O}^i \times \mathscr{A}^i \rightarrow [0,1]$ for each agent, which means each agent takes an action a^i under its observation o^i based on policy π^i probabilistically. Specially, the ε-greedy policy is used where the agent chooses a current optimal action or randomly chooses action with a certain probability.

Transition Function: The environment transfers to the next state $s[t+1]$ upon completion of joint action $\mathbf{a}[t]$ based on transition function $\mathscr{T} : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \rightarrow [0,1]$. This means that the environment probabilistically transfers to the next state s', depending on current state s and joint action $\mathbf{a}$, i.e., $\mathscr{T}(s, \mathbf{a}, s')$=$Prob(s'|s, \mathbf{a})$.

Reward Function: As for the transition $(s[t], \mathbf{a}[t], s[t+1])$, we can obtain cost $Cost[t]$ at step t. In Dec-POMDP, the objective is always to maximize the reward $r(s[t], \mathbf{a}[t], s[t+1])$, thus we take the inverse of $Cost[t]$ as the immediate reward at step t, i.e., $r(s[t], \mathbf{a}[t], s[t+1]) = -Cost[t]$. Thus, minimizing costs corresponds to maximizing the reward, though the cost

definitions of the above two cases are different.

Objective Function: Let us consider a certain period h with T steps and h can be represented as follows,

$$h = [s[1], \mathbf{a}[1], s[2], \mathbf{a}[2], ..., s[T], \mathbf{a}[T], s[T+1]]. \tag{5.24}$$

Then, the discounted sum $R(h)$ of the immediate rewards in the period is given by

$$R(h) = \sum_{t=1}^{T} \gamma^{t-1} r(s[t], \mathbf{a}[t], s[t+1]), \tag{5.25}$$

which is what we want to maximize in one period. Thus, the objective function is given as follows.

$$J(\pi^1, ..., \pi^{|SV|}) = \mathbb{E}_{\pi^1, ..., \pi^{|SV|}, \mathscr{T}} \left[R(h) \right], \tag{5.26}$$

which means we want to identify a couple of policies $(\pi^1, ..., \pi^{|SV|})$ for all agents that can maximize the expectation of team rewards during total period h.

5.5 MARL based Task Allocation Algorithms

There are usually two main types of Dec-POMDP: non-cooperative setting and cooperative setting. The non-cooperative setting can be solved by using fully decentralized MARL algorithms like IDQL [Tampuu et al., 2017]. Their effectiveness in tackling the task allocation problem in edge clouds has been confirmed in some studies [Liu et al., 2020a]. The cooperative setting considered in this section can be solved by the value decomposition

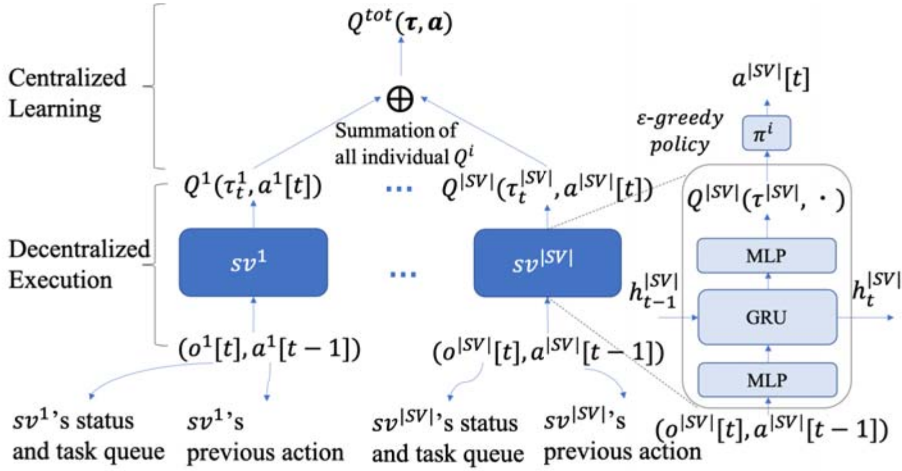


Figure 5.4: Value decomposition network based task allocation algorithm in distributed edge cloud computing.

network (VDN) [Sunehag et al., 2017] and QMIX [Rashid et al., 2018], classical cooperative MARL algorithms. In this section, we propose a novel VDN-based distributed task allocation algorithm for cooperative edge cloud computing. Unlike the traditional VDN which focuses on the general case, our proposed method is intended to solve the server cooperation problem in edge cloud computing. Specifically, the proposed method can be divided into two parts: 1) centralized learning and 2) decentralized execution, as shown in Figure 5.4.

As for 1) centralized learning, although each agent can independently learn its individual policy by its own individual reward like in [Liu et al., 2020a], it cannot achieve a cooperative behavior since the team reward is not considered. Thus, we consider to train the agents in a centralized way.

As for 2) decentralized execution, the joint action space $|\mathscr{A}|^{|SV|}$ will exponentially increase with the number of agents if we directly choose a joint

action from the joint action space $\mathscr{A} = \mathscr{A}^1 \times ... \times \mathscr{A}^{|SV|}$. This is called centralized execution which makes the problem suffer the curse of dimensionality. Thus, we consider to make each agent independently choose action a^i from its own action space $\mathscr{A}^i$.

Centralized Learning

In reinforcement learning, state-action value function $Q : \mathscr{S} \times \mathscr{A} \to \mathbb{R}$ is normally used to learn the optimal policy for agents; it evaluates the quality that results from taking particular actions in certain states [Watkins and Dayan, 1992] [Mnih et al., 2015]. After learning an optimal state-action value function, an optimal policy can be obtained by taking the action with maximum state-action value (Q-value). In this section, we consider two types of Q-value: total Q-value Q^{tot} and individual Q-value Q^i. Total Q-value Q^{tot} is used to evaluate joint action $\mathbf{a}$ under the team reward. Then, individual Q-value Q^i is updated in the direction that can maximize Q^{tot} and is used to choose individual action a^i. We elucidate them separately as follows.

Total Q: In MARL, the Q-value for state and joint action is used to denote the discounted sum of rewards that can be obtained in the future after taking joint action $\mathbf{a}$ under state s. When a couple of optimal policies $(\pi^{1*}, ..., \pi^{|SV|*})$ are given, it is called optimal Q-value and is defined as

$$Q^*(s, \mathbf{a}) = \mathbb{E}_{\pi^{1*}, ..., \pi^{|SV|*}} \left[R(h) \mid s[1] = s, \mathbf{a}[1] = \mathbf{a} \right],$$

where "$|s[1] = o, \mathbf{a}[1] = \mathbf{a}$" means the initial state and joint action are fixed on state s and $\mathbf{a}$, respectively. $Q^*(s, \mathbf{a})$ is the conditional expectation of $R(h)$

given $s[1] = s, \mathbf{a}[1] = \mathbf{a}$. By recursion, it can be rewritten as follows.

$$Q^*(s,\mathbf{a}) = \sum_{s' \in \mathscr{S}} \mathscr{T}(s,\mathbf{a},s') \left[r(s,\mathbf{a},s') + \gamma \max_{\mathbf{a}' \in \mathscr{A}} Q^*(s',\mathbf{a}') \right]. \tag{5.27}$$

However, we cannot obtain the state s in Dec-POMDP as each agent has only its own partial observation o^i. But, agents can benefit from conditioning on their entire action-observation history τ_t^i until step t [Rashid et al., 2018], which is defined by

$$\tau_t^i = [o^i[1], a^i[1], o^i[2], a^i[2], ..., o^i[t-1], a^i[t-1], o^i[t]]. \tag{5.28}$$

We collect all agents' action-observation histories at step t as joint trajectory $\tau_t = [\tau_t^1, ..., \tau_t^{|SV|}]$. Then, we replace the global state s with τ_t to define total Q-value Q^{tot} as $Q^{tot} : \Gamma \times \mathscr{A} \to \mathbb{R}$ where Γ is the set of all possible joint trajectories τ. Specially, Q^{tot} is defined as the sum of the individual Q-values of all agents, i.e.,

$$Q^{tot}(\tau, \mathbf{a}; \theta) = \sum_{i=1}^{|SV|} Q^i(\tau^i, a^i; \theta).$$

Then, our goal is to learn an optimal Q^{tot*} that can maximize the sum $R(h)$ of team rewards over period h. Specially, a reply memory D is used to store the tuple of $(\tau_t, r[t])$ where the joint trajectory τ_t stores the observations and actions of all agents upto step t and $r[t]$ is the team reward obtained at step t. Then, we randomly sample the tuples from D to train $Q^{tot}(\tau, a)$ with the

target of minimizing the following loss function:

$$\mathscr{L}(\theta) = \sum_{k \in D} \left[\left(y_k - Q^{tot}(\tau_k, \mathbf{a}_k; \theta) \right) \right], \tag{5.29}$$

where k is the sample index and $y_k = r_k + \gamma \max_{\mathbf{a}'} Q(\tau_k, \mathbf{a}'; \theta)$ is the target, θ is the set of all agent parameters.

Individual Q: We note above that Q^{tot} is the sum of individual Q^i which will be updated with the aim of maximizing Q^{tot} by backpropagation gradients. Specially, the value of Q^i is based on a deep neural network called deep recurrent Q-network (DRQN) [Hausknecht and Stone, 2015]. The input of DRQN is each agent's observation, previous action and its history information based on τ^i and the output is the values $Q^i(\tau^i, a^i)$ for each action $a^i \in \mathscr{A}^i$.

In DRQN, a classical recurrent neural network called gated recurrent unit (GRU) is used to process observation-action history. Specially, τ_t^i can be handled by the GRU unit to obtain abstract information h_t^i. As shown in Figure 5.5, the current information $(o^i[t], a^i[t-1])$ and the previous abstract information h_{t-1}^i are input to GRU and then the updated abstract information h_t^i is output and used as part of the input in the next step; x_t^i is the result of encoding $(o^i[t], a^i[t-1])$ by a multilayer perceptron (MLP) layer. Specially, h_t^i can be calculated as follows.

$$\begin{aligned} r_t^i &= \delta(W_r \cdot [hç_{t-1}, x_t^i]), \\ z_t^i &= \delta(W_z \cdot [h_{t-1}^i, x_t^i]), \\ \tilde{h}_t^i &= tanh(W_h \cdot [r_t * h_{t-1}^i, x_t^i]), \\ h_t^i &= (1 - z_t^i) * h_{t-1}^i + z_t^i * \tilde{h}_t^i, \end{aligned} \tag{5.30}$$

where $[\cdot,\cdot]$ means to connect two vectors and $*$ means the product of two matrices. In this case, since each edge server has the same structure, we assume all agents share one neural network.

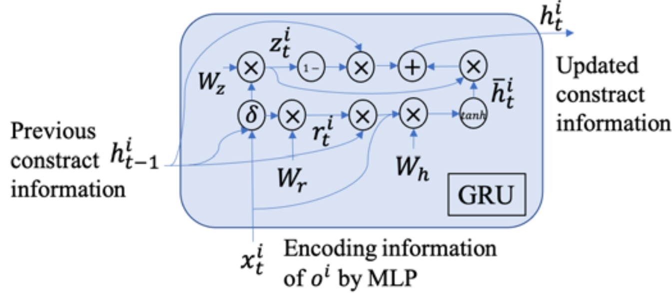


Figure 5.5: The detail of GRU module

Decentralized Execution

The above sub-section described how Q^i is trained in a centralized way. However, choosing actions in a centralized way will incur a huge joint action space, as it increases exponentially with agent number. Thus, we consider the decentralized execution approach where each agent independently chooses action a^i based on its individual Q^i, i.e.,

$$\arg\max_{\mathbf{a}} Q^{tot}(\tau,\mathbf{a}) = \begin{pmatrix} \arg\max_{a^1} Q^1(\tau^1,a^1) \\ \arg\max_{a^2} Q^2(\tau^2,a^2) \\ \cdots \\ \arg\max_{a^{|SV|}} Q^{|SV|}(\tau^{|SV|},a^{|SV|}) \end{pmatrix}.$$

This means that the result of collecting each optimal action by operation

$\arg\max_{a} Q^i(\tau^i, a^i)$ is equivalent to that of directly searching for a joint action by operation $\arg\max_{\mathbf{a}} Q^{tot}(\tau, \mathbf{a})$. To ensure that a global argmax performed on Q^{tot} yields the same result as a set of individual argmax operations performed on each Q^i, monotonicity can be enforced through a constraint on the relationship between Q^{tot} and each Q^i [Rashid et al., 2018], i.e, $\frac{\partial Q^{tot}}{\partial Q^i} \geq 0$. This means that all Q^i functions have the same monotonicity with regard to Q^{tot}. It is easy to prove that VDN-TO satisfies this equation since we have $\frac{\partial Q^{tot}}{\partial Q^i} = 1 > 0$ for each agent.

Finally, the specific process of decentralized execution is shown in the left part of Figure 5.4 where each edge server captures its observation o^i, and takes action a^i based on its own policy $\pi^i(\tau^i, a^i)$. Then, all observations o^i are collected to form state $s = (o^1, ..., o^{|SV|})$ and all actions are collected to form a joint action $\mathbf{a} = (a^1, ..., a^{|SV|})$. The environment transfers to the next state s' based on the transition function $\mathscr{T}(s, \mathbf{a}, s')$ and reward $r(s, \mathbf{a}, s')$ can be obtained.

As for the case 1 of task offloading, we propose value decomposition network based task offloading algorithm (VDN-TO) as shwon in Algorithm 3. As for the case 2 of decentralized federated learning, we propose value decomposition network based decentralized federated learning algorithm (VDN-DFL) as shwon in Algorithm 4.

5.6 Evaluation

This section uses task data from some real datasets to evaluate the performance of our proposed method in performing distributed task allocation in cooperative edge cloud computing for both case 1 and case 2.

ALGORITHM 3: Value Decomposition Network based Task Offloading (VDN-TO) Algorithm

```
1  Initialize replay memory D to capacity N
2  Initialize DRQL network Q^i with random weights
3  for episode m=1, M do
4      for step t =1, T do
5          For each edge server i, the task queue ta^i[t] arrives and its own
             status sv^i[t] is observed to form the observation
6          o^i[t] = [sv^i[t], ta^i[t]]
7          Add o^i[t] and a^i[t-1] to the trajectory τ_t^i for each agent i
8          Decentralized execution:
9          for edge server i =1, |SV| do
10             Based on the trajectory τ_{t-1}^i, edge server i randomly selects
                 an action a^i from A^i based on a ε-greedy policy
11             otherwise selects a^i[t] = arg max_{a^i} Q^i(τ_t^i, a^i; θ)
12         end
13         Collect the actions a^i[t] from all edge servers to form a joint
             action a[t] = [a^1[t], ..., a^{|SV|}[t]]
14         Execute joint action a[t] in distributed edge cloud, then transfer
             to the next state s[t+1]
15         Obtain reward r(s[t], a[t], s[t+1]) and set s[t+1] = s[t]
16         Store transition (τ_t, r_t) in D
17         Centralized learning: Sample random mini-batch of
             transitions from D
18         Calculate y_k and perform a gradient descent step on
             (y_k - Q^{tot}(τ_k, a_k; θ))^2 to update θ
19     end
20 end
```

Case 1: Independently Performing Multiple Tasks

Evaluation Settings

ALGORITHM 4: Value Decomposition Network based Decentralized Federated Learning (VDN-DFL) Algorithm

1 Initialize replay memory D to capacity N and all the parameters of DRQL network.
2 Start to learn a FL model
3 **for** *episode m=1, M* **do**
4   **for** *step t =1, T* **do**
5   Each edge server i observes its current parameter status of the learned model, $o^i[t]$.
6   Add $o^i[t]$ and $a^i[t-1]$ to the trajectory τ_t^i for each agent i
7   **for** *edge server i =1,* $|SV|$ **do**
8   Based on the trajectory τ_{t-1}^i, edge server i decides whether to join current learning. It randomly selects an action a^i from $\mathscr{A}^i$
9   otherwise selects $a^i[t] = \arg\max_{a^i} Q^i(\tau_t^i, a^i; \theta)$
10   **end**
11   Forming a joint action $\mathbf{a}[t] = [a^1[t], ..., a^{|SV|}[t]]$ and choose manager by manager agnet.
12   Execute joint action $\mathbf{a}[t]$ in distributed edge cloud, then transfer to the next state $s[t+1]$
13   Obtain reward $r(s[t], \mathbf{a}[t], s[t+1])$ and set $s[t+1] = s[t]$
14   Store transition $(\tau_t, r[t])$ in D
15   Sample random mini-batch of transitions from D
16   Calculate y_k and perform a gradient descent step on $(y_k - Q^{tot}(\tau_k, \mathbf{a}_k; \theta))^2$ to update θ and update fr^i.
17   **end**
18 **end**

Task Setting: The real data chosen is called google-cluster data; it represents 29 day's worth of Borg cell information from May 2011, on a cluster of about 12.5k servers [Reiss et al.,]. The task information includes the computation resources of RAM, CPU and Disk, all of which are used di-

rectly as task parameters. Since it does not include the information about task workload, we randomly generate the workload of each task following a uniform probability distribution. Specifically, in this section we use a uniform distribution among $(0, work_{max})$ where $work_{max}$ is the maximum workload of the tasks, i.e., $work \sim unif(0, work_{max})$.

Since the scenario we considered in this section is a smart hospital which usually hosts no more than ten servers, we randomly choose five servers' data from two days [Reiss et al.,]. The chosen server numbers are $\{3938719206, 351618647, 329150663, 1303745, 431052910\}$. Then, each episode is deemed to be finished when all tasks in the task set are been completed.

Neural Network Setting: The detail of neural network structure in VDN-TO is shown in Figure 5.5 (Since each agent has the same neural network framework, we only draw one of them). As for the first MLP, its input is agent i's current observation o^i and its previous action a^i, then the layer size is equal to $|o^i| + |a^i|$. Specifically, each server has five features and each action has dimension of 5 (we assume the maximum number of arriving task at each step is 5, i.e., $|a^i| = ta_{max} = 5$). Thus, the input MLP layer size is 10 and None activation function is used. Then, GRU with 64 neurons takes the results of the first MLP layer and its hidden variable h^i_{t-1} at the last step as the inputs. The activation function of the layer is tanh. Thus, it outputs hidden variable h^i_t with dimension of 64. As for the second MLP, it takes h^i_t as its input and outputs the Q-values for each action. The output layer size $2^{ta_{max}}$ is equal to $2^5 = 32$, since we assume $ta_{max} = 5$. We implemented it using Tensorflow 2.0.

Evaluation Results In this section, the following task offloading (TO) base-

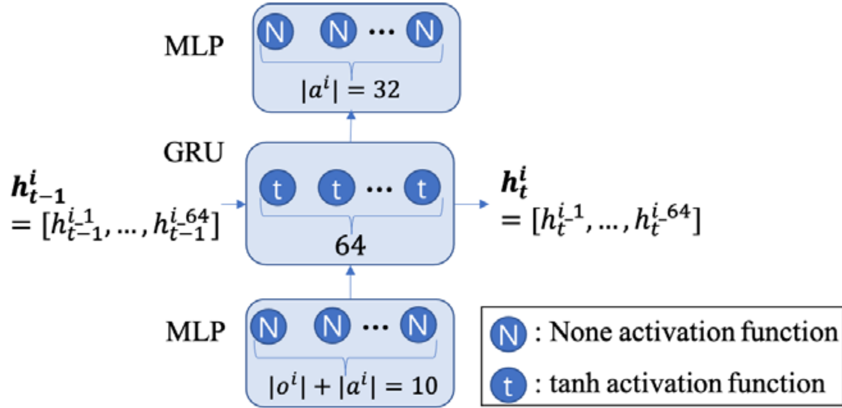


Figure 5.6: The detail of Q^i neural network structure.

line algorithms are adopted: 1) IDQL-TO [Liu et al., 2020a] which is a deep RL based TO approach; 2) QL-TO [Jiang et al., 2020] which is a tabular RL based TO approach; 3) Joint-TO [Barbarossa et al., 2013] which is a rule-based TO approach where the edge servers are divided to two groups according to the task data size that they need to offload, and only one group of servers can jointly offload tasks; and 4) random policy. In this case, we called our proposed method as VDN-TO and compare it with the baselines. Since the tasks include some random elements which make them different at each episode, we take 5 episodes as one *round* and use the average of $R(h)$ in one round to compare. Specially, each experiment consisted of 500 episodes which corresponds to 100 rounds. We used the same hyperparameters for all RL/DRL based algorithms with α=0.01 and γ=0.9.

First, we consider the latency-sensitive case whose objective cost is defined in Eq. (5.11). The results are shown in Figure 5.7. QL-TO approaches the performance of random policy. That is because QL-TO is a tabular method which prevents it from learning an optimal policy in large state space. Since

IDQL-TO can well cope with the large state space, it can attain better performance than QL-TO. However, the result of learning is unstable: the range of its oscillation is very large, and sometimes its performance is worse than that of random policy even in the final learning phase. This is because each edge server just considers its own interest and they will conflict if they choose to offload many tasks at the same time. Joint-TO can attain better performance than QL-TO and is more stable than IDQL-TO. However, its performance does not improve since it does not learn over the iterations. On the other hand, our proposed VDN-TO algorithm uses a total Q and each edge server tries to optimize it in a cooperation way, which yields good and stable performance.

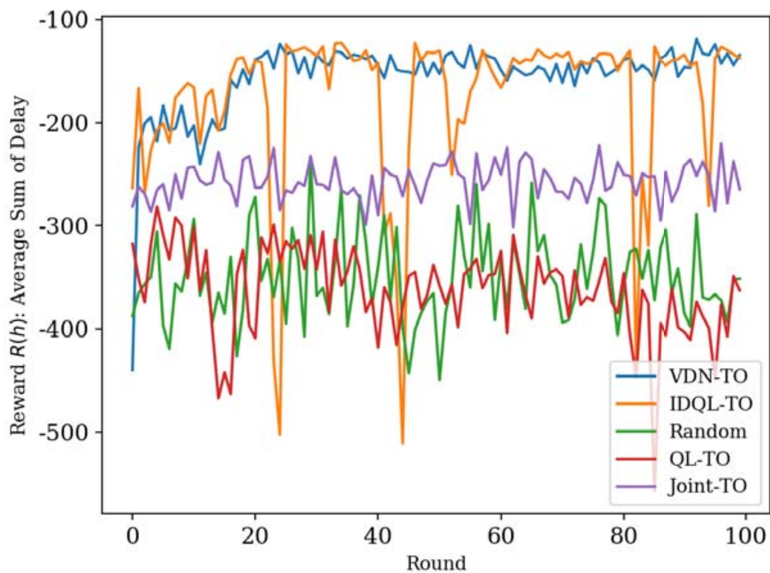


Figure 5.7: Comparing performances of VDN-TO with IDQL-TO and random policy in latency-sensitive case.

Second, we consider the energy-sensitive case whose objective cost is defined in Eq. (5.12). The results are shown in Figure 5.8. Since cloud servers have much higher unit energy cost than edge servers, the optimal policy is to execute all tasks at edge servers in this energy-sensitive case. Thus, each edge server's maximized interest can result in maximizing the team interest. Moreover, each agent's optimal policy does not influence other agents' optimal policies. Although QL-TO still suffers the large state space problem, its learning environment becomes more stable (non-interest-conflict) than in the latency-sensitive case so its performance is superior to that of random policy. Joint-TO can get the same performance as QL-TO in a stable way. However, its performance does not improve since it does not learn over the iterations. In this non-interest-conflict situation, each edge server's maximized interest is consistent with team maximized interest, thus the self-interested IDQL-TO can also achieve a performance as good as VDN-TO. Our proposed VDN-TO algorithm can still learn optimal policies in a stable manner.

Third, we consider a balance case whose objective cost is defined in Eq. (5.13). Without losing generality, we set $\omega_t = 0.5, \omega_e = 0.5$ in Eq. (5.13). The results are shown in Figure 5.9. Although QL-TO still suffers the large state space problem, its learning environment is more stable than in the latency-sensitive since the energy-sensitive part is included. Thus, its performance is better than that of random policy. Joint-TO still maintains a stable performance due to its rule-based characteristic. Although IDQL-TO has similar performance with VDN-TO, it exhibits oscillation. Since the latency part requires a cooperative setting under non-interest-conflict situation, which can be well handled by VDN-TO, the performance of VDN-TO

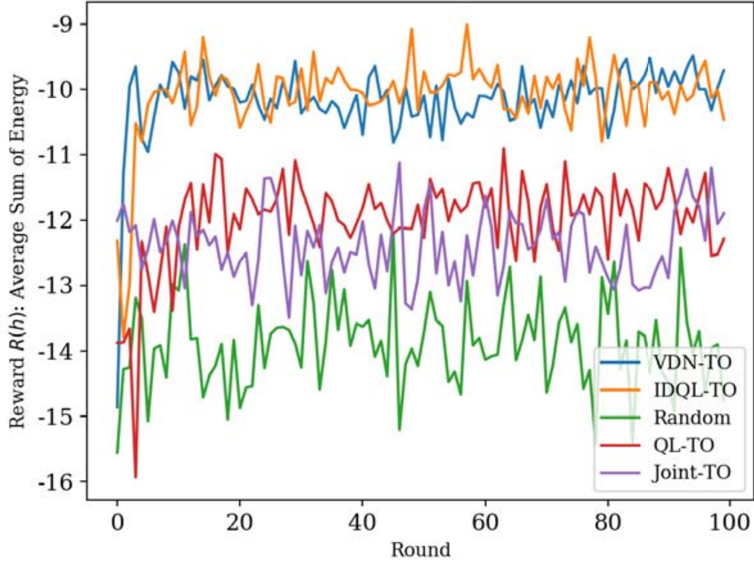


Figure 5.8: Comparing performances of VDN-TO with IDQL-TO and random policy in energy-sensitive case.

is better than that of IDQL-TO.

To summarize the above experiments, we can conclude that our proposed VDN-TO algorithm can solve the cooperative task allocation problem in distributed edge cloud computing. It exceeds the performance of other baseline algorithms in three classical settings of edge cloud computing: latency-sensitive case, energy-sensitive case, and a balance between latency and energy. Moreover, we consider a decentralized setting and the centralized manner is not considered in this section. Although the centralized manner can evaluate the results of joint action more accurately (it treats the problem as a single agent problem), it might yield better performance than cooperation under partial observations. However, this advantage might be effective

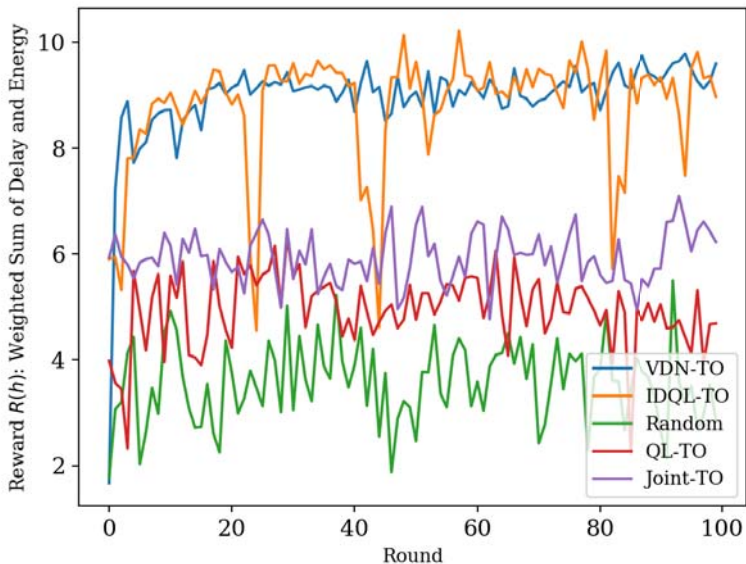


Figure 5.9: Comparing performances of VDN-TO with IDQL-TO and random policy in balance case.

only if the number of edge servers is small, which is seldom the case in real-world scenarios.

Case 2: Jointly Performing One Task

Evaluation Setting

We call our proposed method as VDN-DFL in this case. Then, we refer [Wang et al., 2020] to implement our VDN-DFL codes in Python with PyTorch, a deep learning framework. And also, using Python threading library to simulate the client severs where each client server is simulated by a thread.

Dataset: Similar with most of federated learning studies [Wang et al., 2020], we evaluate our VDN-DFL by training on CNN models on three popular benchmark datasets: MNIST, FashionMNIST and CIFAR-10. Specifically, each dataset is illustrated as follows.

- **MNIST**: MNIST is a handwritten digits dataset from number 0 to 9 that includes a training set of 60,000 examples, and a test set of 10,000 examples. The number of examples for each digit is same.
- **FashionMNIST**: FashionMNIST is a dataset of Zalando's article 28x28 grayscale images with ten classes such as T-shirt and Jeans, that includes of a training set of 60,000 examples and a test set of 10,000 examples.
- **CIFAR-10**: CIFAR-10 is a 32x32 colour images in 10 classes such as airplane, automobile and bird, that includes a training set of 60,000 examples, and a test set of 10,000 examples. The number of examples for each class is same.

The clients perform local training for E = 6 epochs in each training round. We set a 10 rounds in each episode and the goal is to obtain an accuracy as high as possible with minimizing the processing time and communication cost.

Training Process of Our VDN-DFL

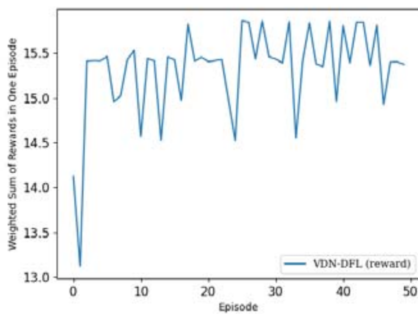
We first train our VDN-DFL with 10 client edge servers. The VDN part consists of one layer MLP network with the size of $|o^i|$ where we employee a PCA to reduce the all parameters of current CNN models to a 10 dimension vector. Then MLP connects a 64 hidden layers with the ReLU activation

functions. Then the output layer size is two since only $\{0,1\}$ actions exist. We train it on a AWS EC@ instance x.large with a 2CPU and 1GB memory.

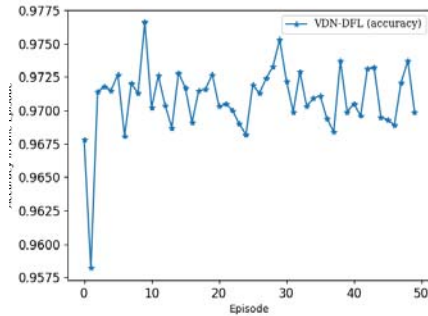
Figure 5.10-Figure 5.12 show the learning process of VDN-DFL on the MNIST, FashionMNIST and CIFAR-10 datasets. The total reward represents the sum of accuracy improvement, processing time and communication cost, which corresponds to the weights w_1, w_2 and w_3. We set $w_1 = 10$, $w_2 = 0.1$ and $w_3 = 0.1$ to represent that the accuracy improvement is the most important among these three components. Figure 5.10 (a) shows that our proposed method can improve its performance during episode training. Figure 5.10 (b)-(d) show the performance of each component during the training. Specifically, we record the final accuracy after finishing round training at each episode in Figure 5.10 (b).

We can see the accuracy can improve well during episode training. Since the MNIST is a dataset that is easy to learn, it can learn a final accuracy after 5 rounds. Since the deadline of communication cost is set a high value which means once the server is failed down, a high communication cost is incurred. From Figure 5.11(d), we can see that the communication cost decreases during the learning. Since both reward and communication cost have a dominant influence to the reward, the component of processing time improvement does not show an increment during learning. The processing time even increases during round training. That is because improving the accuracy in the final rounds during training, it usually requires more servers to take part in each round, which corresponds to a high processing time.

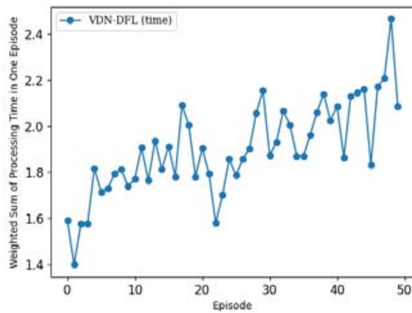
Figure 5.11 shows the results on learning CIFAR-10 whose learning performances are similar to those on MNIST dataset. It is harder to learn than MNIST, thus the final accuracy learned is lower than that of MNIST given



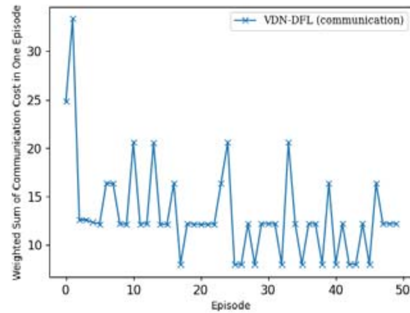
(a) reward



(b) accuracy

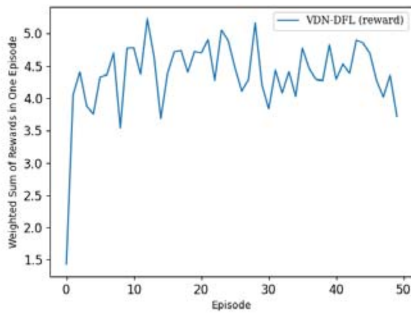


(c) processing time

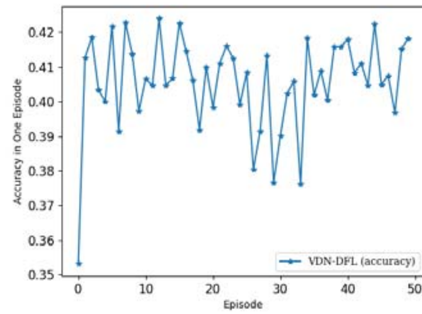


(d) communication cost

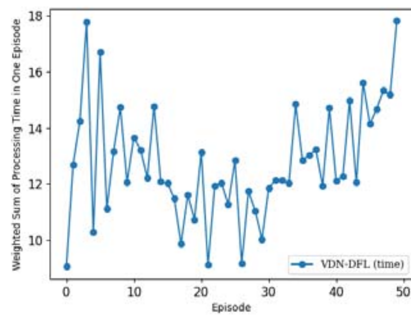
Figure 5.10: Comparing performances of VDN-DFL on MNIST dataset in (a) reward, (b) processing time, (c) accuracy and (d) communication cost.



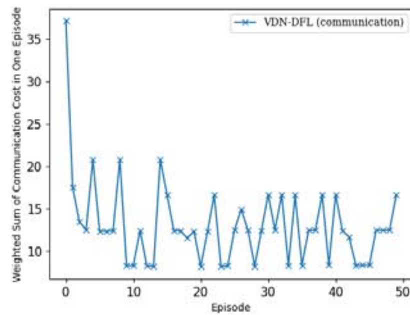
(a) reward



(b) accuracy

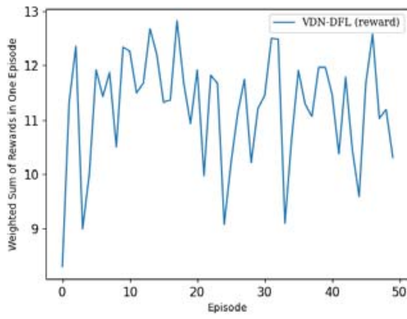


(c) processing time

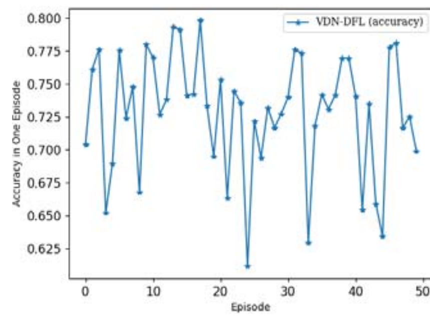


(d) communication cost

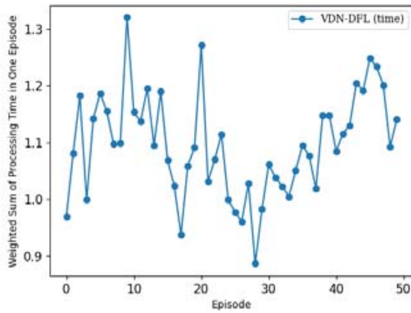
Figure 5.11: Comparing performances of VDN-DFL on CIFAR-10 dataset in (a) reward, (b) processing time, (c) accuracy and (d) communication cost.



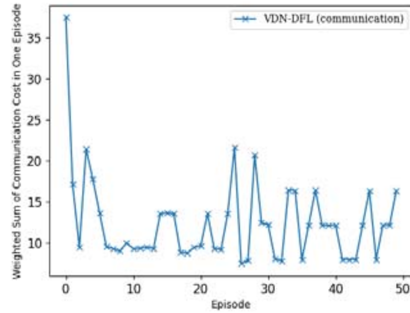
(a) reward



(b) accuracy



(c) processing time



(d) communication cost

Figure 5.12: Comparing performances of VDN-DFL on FashionMNIST dataset in (a) reward, (b) processing time, (c) accuracy and (d) communication cost.

the same round number. As for the learning on FashionMNIST dataset, although the Figure 5.12 (a) has shown a improvement during learning in reward. Compared with Figure 5.10 and Figure 5.11, we can see the final perfomance is not as stable as those on MNIST and CAFIR-10. That is because the dataset of FashionMNIST is more difficult to be learned. From Figure 5.12 (b), we can see the accuracy improvement is unstable.

Comparison of Baselines

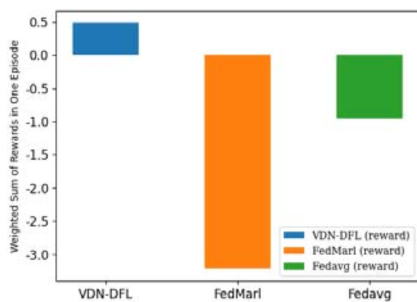
We choose two baseline algorithms to compare with our proposed method.

- **Fedavg**: Fedavg [Li et al., 2019][Nilsson et al., 2018] is a classical FL algorithm for centralized FL where *k* clients are randomly selected to take part in learning during each round. We apply it to DFL by randomly setting a parameter server then it just follows the traditional way of client selection.

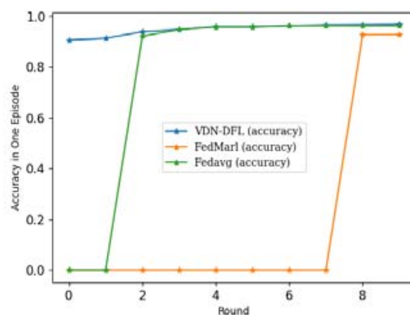
- **FedMarl**: FedMarl is a client selection algorithm for centralized FL by applying a MARL algorithm to process the client selection. We use it to DFL by randomly setting a parameter server then it just follows the traditional way of client selection [Qian Zhang et al., 2022].

Figure 5.13 shows that the performances of our proposed method and the other baselines on dataset MNIST. From Figure 5.13 (a), we can see that FedMarl and Fedavg have the similar performances which can hardly learn during the training.

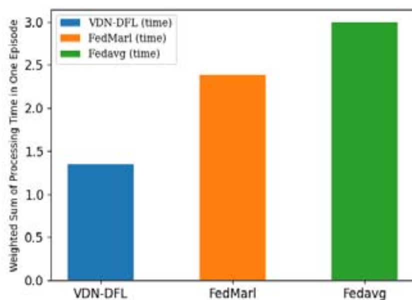
That is because failure of parameter server would incur a high communication cost. Specifically, Figure 5.13 (b) shows the accuracy during 10 round learning in the final episode.



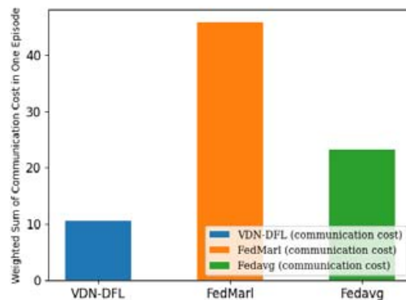
(a) reward



(b) accuracy

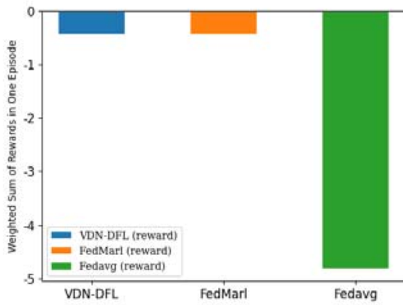


(c) processing time

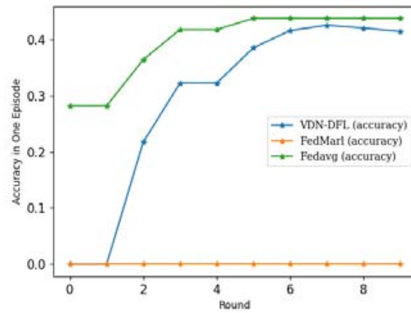


(d) communication cost

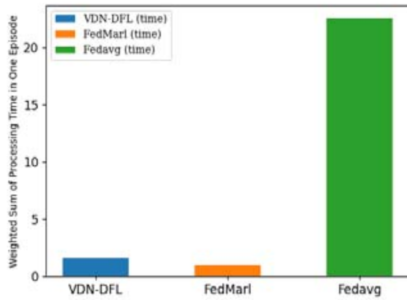
Figure 5.13: Comparing performances of VDN-DFL with baselines on MNIST dataset in (a) reward, (b) accuracy , (c) processing time and (d)communication cost.



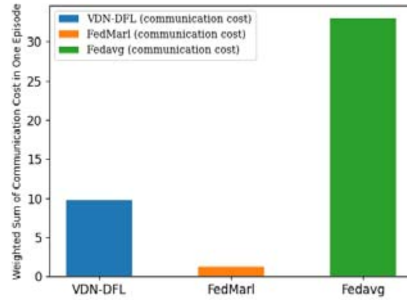
(a) reward



(b) accuracy

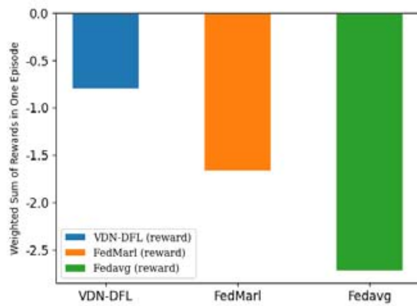


(c) processing time

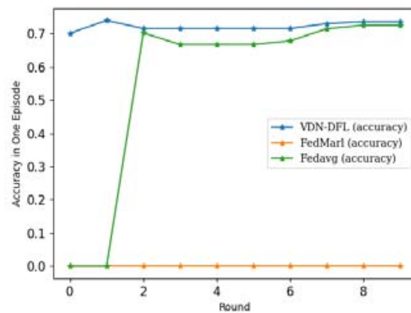


(d) communication cost

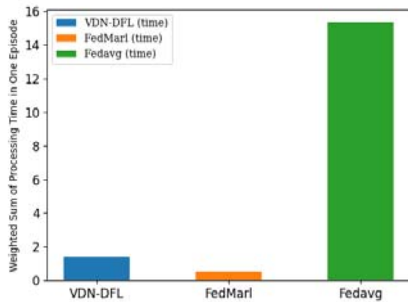
Figure 5.14: Comparing performances of VDN-DFL with baselines on CIFAR-10 dataset in (a) reward, (b) accuracy , (c) processing time and (d)communication cost.



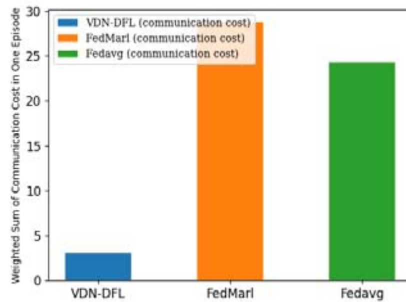
(a) reward



(b) accuracy



(c) processing time



(d) communication cost

Figure 5.15: Comparing performances of VDN-DFL with baselines on FashionMNIST dataset in (a) reward, (b) accuracy, (c) processing time and (d) communication cost.

Although the three algorithms can get a good accuracy finally, our proposed algorithm can get an accuracy over 90% at the first round learning, which corresponds to a fast learning in accuracy improvement. As for the processing time and communication cost, we can see that our proposed method can have a smaller value than FedMarl and Fedavg, as shown in Figure 5.13 (c) and (d). As for the FedMarl, although it can get a good accuracy with smaller processing time than Fedavg, it corresponds to a highest communication cost. That is because FedMarl can only learn a client selection and fails to learn a manager selection, which incurs a high communication cost. Compared with these results, it shows that our proposed method can learn a high accuracy fast while generating a smallest processing time and communication cost.

As for the learning result on CIFAR-10, our proposed method still maintains a better performance than the other baseline algorithms, as shown in Figure 5.14 (a). Although the performance of FedMarl can learn better than Fedavg in reward, its accuracy does not improve during training. Further, from Figure 5.14 (c) and (d), we can see that the processing time and communication cost become to zero. That means FedMarl learn to choose none client selection and manager selection to avoid a failure which brings negative cost. Thus, it can still show that our proposed method can learn a better performance than other baselines.

5.7 Summary

In Chapters 3 and 4, we studied two major task features which are the task's internal feature (high-workloads) and task interacted feature (dependency).

However, both they are in a centralized edge cloud computing environment. However, there are many realistic scenarios where the edge servers are distributed among multiple areas, which corresponds to a distributed task allocation problem.

Unlike most existing studies which usually tackle the problem in the self-interested setting, this chapter studies the new problem of distributed task allocation in cooperative edge cloud computing, with the target of maximizing team reward. Specifically, we consider two major cases of this problem as follows. The first is task offloading, where multiple tasks exist and each task can be well performed by one server. The second is federated learning, where all the servers jointly perform one single task. We propose a multiagent reinforcement learning based task allocation algorithm to guide servers towards cooperating with each other even under partial observations. We validate our approach by using a real dataset to compare it with baseline algorithms. The results show that our approach achieves significantly better performance than the baseline algorithms. We believe solving this distributed task allocation problem is key to achieve cooperative edge cloud computing. This work has been published in [Ding and Lin, 2022b].

Chapter 6

Conclusion

6.1 Contributions

With the rapid development of smart IoT communities, task allocation in cooperative edge cloud computing has become a fundamental problem to be solved. In a departure from most existing studies on edge cloud computing that emphasize the self-interested setting, this thesis emphasizes a cooperative setting. In this research, we considered three major cases needing cooperation of the edge cloud servers as three research topics: high-workload task allocation, dependent task allocation and distributed task allocation. To resolve the issues raised in those topics, we proposed several novel methods for task allocation in each case of cooperative edge cloud computing. In this chapter, we summarize our contributions as follows.

- As for high-workload task allocation, we proposed a dynamic coalition formation model called CMDP and its corresponding solutions

called CQL and DCQL. It can enable the servers in edge cloud computing to cooperate in performing the tasks automatically. In contrast to the existing studies that usually perform task allocation with high-workloads through the use of static human-designed rules, our proposed methods can adaptively update their models to suit dynamic features in different IoT environments without any human-design rules. Moreover, existing studies usually 1) focus on the task itself and ignore server status; 2) and also usually consider a one-step goal in a static process rather than a multi-step goal in a dynamic process. Our proposed methods allow edge servers to organize/cooperate by themselves by considering the dynamic information of both servers and tasks, which can well cope with the dynamic features of the IoT environment. Our proposed methods can be used by the developers who intend to construct a centralized edge cloud computing system that must cope with high-workload tasks.

- As for dependent task allocation, we developed graph convolutional reinforcement learning algorithms for realizing task allocation with the features of both dependency and dynamic task flow. In contrast to dependent task allocation proposed in existing studies which allocate dependent tasks in a static process, we consider their deployment in a dynamic environment where the arriving tasks and server statuses dynamically change. Specifically, the graph convolutional part can extract the task information and the relationships between multiple tasks. The RL part offers effective decision-making for long-term goals rather than just one-step goals. These two parts are developed in series and are updated at the same time to achieve the goal. That

is, these end-to-end methods do not require any manual operation by developers in edge cloud computing.

- As for distributed task allocation, we consider to make all the servers to cooperate under partial observations to optimize the team interest over the entire edge cloud computing system. Specifically, we consider two classical scenarios. The first is task offloading, where multiple tasks exist and the tasks can be well performed by one server. The second is federated learning, where all the servers jointly perform one single task. We proposed a novel multiagent reinforcement learning for distributed task allocation by regarding each edge server as an agent. Our proposed methods allow servers to cooperate even though each server cannot observe the other servers' information. It can well cope with the dynamic features and incomplete observations expected in the IoT environment, without any human designed rules. We believe this method will make it easy for the developers who want to construct their own edge cloud computing system in a distributed way.

6.2 Discussion

In this section, we discuss both the limitations and extensions of the three topics. As for the topic in Chapter 3, our proposed method offers both centralized learning and centralized execution. A centralized coordinator that can know the global information of all servers exists. This corresponds to a decision problem with complete information which makes the learning process stable. However, scale-out is limited since both the state space and

action space will exponentially increase with the agent number. This hinders the efficiency of our proposed method in which the number of edge servers is large.

As for the topic in Chapter 4, GCRL also applies a centralized approach to allocate tasks while considering task dependency at the same time. Thus, it also suffers from the curse of dimensionality. As for MGCRL, although a centralized coordinator can be set to control the entire task allocation process, it is essentially a fully decentralized technique. This is because the proposed methods take each task of the multiple jobs as the input, which linearly increases with the number of tasks. Also, since the action space is the set of servers, the action space linearly increases with the number of servers. The disadvantage of MGCRL is that it cannot evaluate each task allocation precisely. This is because the reward depends on all task allocation decisions, which cannot precisely evaluate each individual action.

As for the topic in Chapter 5, we adopt centralized learning and decentralized execution. This combination can avoid the above disadvantages such as the dimensionality curse and unstable learning process. Specifically, each edge server can independently make decisions as to task-offloading which avoids the joint action space that exponentially increases with agent number. Moreover, our use of centralized learning allows all the agents' information to be collected which makes the learning process more stable.

In this thesis, we independently consider three task allocation cases in cooperative edge cloud computing with the features of high-workload, dependency and distribution. In this section, we will consider the cases in which any two of the above features exist at the same time.

We start with the case in which both high-workloads and dependency exist at the same time. First, the servers must form coalitions to jointly perform the tasks due to the high-workload. Although our coalitional-RL methods can cope with this problem well and the dynamic information is also well handled, the dependency information must also be considered. Unlike directly inputting the task information to the coalitional RL methods to make the servers dynamically form coalitions, the embedding part of our proposed GCRL or MGCRL can be used to handle the dependency information. Then the embedding results can be input to the decision part which can use the coalitional-RL methods.

Next is the case in which both high-workloads and distribution exist at the same time. It requires that the servers form coalitions in a distributed way. As for coalition formation in topic 1, the coalition formation part is based on CSG where a centralized coordinator exists to control each agent in joining different coalitions. However, it is impossible to identify a centralized coordinator in distributed task allocation. This is because the centralized coordinator does not exist and the edge servers cannot observe each other. Thus, we consider to replace CSG by a coalition formation game where each agent can independently choose one coalition to join. In the coalition formation game, the goal is to identify a core which is a state in which each agent would not leave its current coalition. In the traditional coalition formation game, it assumes that each agent is self-interested. In our cooperative setting, we need each self-interested agent to share one common team reward function.

As for the case wherein both features of dependency and distribution exist at the same time, the task arriving at one edge server might depend on the

other task arriving at another server. It means that each edge server decides whether task offloading needs to consider such kind of dependency information. We can first use GCN to encode the dependency information and then take the embedding result as the input to VDN to make them achieve a cooperation.

6.3 Future Directions

- **Improve the generalization ability of our proposed methods** In this thesis, we considered a dynamic edge cloud computing environment which can be formulated as an MDP. Since user behavior in many scenarios follows a regular pattern like a Gaussian distribution, we correspondingly assumed that the transition probability function of MDP remains unchanged. However, in some scenarios user behavior may not follow any certain pattern, which creates the problem of an uncertain transition probability function. This suggests the need to consider a generalization problem: how to generalize one model trained for one environment so that it covers another environment with different transition probability function.
- **Extension to continuous space** In this thesis, we studied a discrete action space of task allocation where each task can be totally allocated to one of the servers or be divided into finite sub-tasks to allocate, which corresponds to a discrete action space. In some cases, it requires to consider resource allocation for each task further after allocating task on the servers, such as deciding how much CPU resources are needed for task allocation. This corresponds to a con-

tinuous action space, since the computation resources are usually defined as continuous values. Thus, extending our proposed methods to support their application to a continuous space is another important direction.

Publications

Journal

Shiyao Ding and Donghui Lin. Multi-Agent Reinforcement Learning for Cooperative Task allocation in Distributed Edge Cloud Computing. *IEICE Transactions on Information and Systems*, vol.E105-D(5), pp.936-945, 2022.

Shiyao Ding and Donghui Lin. Deep Coalitional Q-learning for Dynamic Coalition Formation in Edge Computing. *IEICE Transactions on Information and Systems*, vol.E105-D(5), pp.864-872, 2022.

Conferences

Shiyao Ding, Donghui Lin, and Xin Zhou. Graph Convolutional Reinforcement Learning for Dependent Task Allocation in Edge Computing. In *The 5th IEEE International Conference on Agents (IEEE ICA 2021)*, pp.25-30, Kyoto, Japan, December 2021. (Best Student Paper Award)

Shiyao Ding. Multi-Agent Reinforcement Learning for Task Allocation in Cooperative Edge Cloud Computing. In *The International Conference on*

Service-Oriented Computing (ICSOC 2021), PhD Symposium, November 2021.

Shiyao Ding and Donghui Lin. A Coalitional Markov Decision Process Model for Dynamic Coalition Formation among Agents. In *The 2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2020)*, pp.308-315, Melbourne, Australia, December 2020.

Shiyao Ding and Donghui Lin. Dynamic Task Allocation for Cost-Efficient Edge Cloud Computing. In *The 17th IEEE International Conference on Services Computing (IEEE SCC 2020)*, pp.218-225, Beijing, China, October 2020.

Other Publications

Journal

Shiyao Ding, Toshimitsu Ushio. Learning in Two-Player Matrix Games by Policy Gradient Lagging Anchor. *IEICE Transactions on Fundamentals of Electronics,Communications and Computer Science*, vol.E102–A-4, pp.708-711, April 2019.

Conference

Shiyao Ding, Hideki Aoyama and Donghui Lin. Combining Multiagent Reinforcement Learning and Search Method for Drone Delivery on a Non-Grid Graph. In *The 20th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2022)*, L'Aquila, Italy, July 2022.

Bowen Wei, **Shiyao Ding** and Donghui Lin. A Constraint-based Approach to Edge Resource Allocation for Complex Event Processing. In *The 2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2020)*, pp.526-531, Melbourne, Australia, December 2020.

Bibliography

[Abdel-Jabbar et al., 2014] Abdel-Jabbar, M.-A. H., Kacem, I., and Martin, S. (2014). Unrelated parallel machines with precedence constraints: application to cloud computing. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 438–442. IEEE.

[Abdellatif et al., 2019] Abdellatif, A. A., Mohamed, A., Chiasserini, C. F., Tlili, M., and Erbad, A. (2019). Edge computing for smart health: Context-aware approaches, opportunities, and challenges. *IEEE Network*, 33(3):196–203.

[Armenta-Cano et al., 2015] Armenta-Cano, F., Tchernykh, A., Cortés-Mendoza, J. M., Yahyapour, R., Drozdov, A. Y., Bouvry, P., Kliazovich, D., and Avetisyan, A. (2015). Heterogeneous job consolidation for power aware scheduling with quality of service. In , pages 687–697.

[Avgeris et al., 2019] Avgeris, M., Spatharakis, D., Dechouniotis, D., Kalatzis, N., Roussaki, I., and Papavassiliou, S. (2019). Where there is fire there is smoke: A scalable edge computing framework for early fire detection. *Sensors*, 19(3):639.

[Aydin et al., 2004] Aydin, H., Melhem, R., Mossé, D., and Mejía-Alvarez, P. (2004). Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on computers*, 53(5):584–600.

[Banerjee et al., 2001] Banerjee, S., Konishi, H., and Sönmez, T. (2001). Core in a simple coalition formation game. *Social Choice and Welfare*, 18(1):135–153.

[Barbarossa et al., 2013] Barbarossa, S., Sardellitti, S., and Di Lorenzo, P. (2013). Joint allocation of computation and communication resources in multiuser mobile cloud computing. In *2013 IEEE 14th workshop on signal processing advances in wireless communications (SPAWC)*, pages 26–30. IEEE.

[Bonomi et al., 2012] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16.

[Braekers et al., 2016] Braekers, K., Ramaekers, K., and Van Nieuwenhuyse, I. (2016). The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering*, 99:300–313.

[Chang et al., 2019] Chang, C., Srirama, S. N., and Buyya, R. (2019). Internet of things (iot) and new computing paradigms. *Fog and edge computing: principles and paradigms*, pages 1–23.

[Chang et al., 2014] Chang, H., Hari, A., Mukherjee, S., and Lakshman, T. (2014). Bringing the cloud to the edge. In *2014 IEEE Conference*

on Computer Communications Workshops (INFOCOM WKSHPS), pages 346–351. IEEE.

[Chen et al., 2015] Chen, X., Jiao, L., Li, W., and Fu, X. (2015). Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, 24(5):2795–2808.

[Chen et al., 2018a] Chen, X., Li, W., Lu, S., Zhou, Z., and Fu, X. (2018a). Efficient resource allocation for on-demand mobile-edge cloud computing. *IEEE Transactions on Vehicular Technology*, 67(9):8769–8780.

[Chen et al., 2018b] Chen, X., Zhang, H., Wu, C., Mao, S., Ji, Y., and Bennis, M. (2018b). Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018.

[Chen and Wang, 2020] Chen, Z. and Wang, X. (2020). Decentralized computation offloading for multi-user mobile edge computing: A deep reinforcement learning approach. *EURASIP Journal on Wireless Communications and Networking*, 2020(1):1–21.

[Chiang et al., 2008] Chiang, M., Hande, P., Lan, T., Tan, C. W., et al. (2008). Power control in wireless cellular networks. *Foundations and Trends® in Networking*, 2(4):381–533.

[Clarke and Wright, 1964] Clarke, G. and Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581.

[Dantzig and Ramser, 1959] Dantzig, G. B. and Ramser, J. H. (1959). The truck dispatching problem. *Management science*, 6(1):80–91.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[Dhirani et al., 2017] Dhirani, L. L., Newe, T., Lewis, E., and Nizamani, S. (2017). Cloud computing and internet of things fusion: Cost issues. In *2017 Eleventh International Conference on Sensing Technology (ICST)*, pages 1–6. IEEE.

[Ding and Lin, 2020] Ding, S. and Lin, D. (2020). A coalitional markov decision process model for dynamic coalition formation among agents. In *2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pages 308–315. IEEE.

[Ding and Lin, 2022a] Ding, S. and Lin, D. (2022a). Deep coalitional q-learning for dynamic coalition formation in edge computing. *IEICE TRANSACTIONS on Information and Systems*, 105(5):864–872.

[Ding and Lin, 2022b] Ding, S. and Lin, D. (2022b). Multi-agent reinforcement learning for cooperative task offloading in distributed edge cloud computing. *IEICE Transactions on Information and Systems*, 105(5):936–945.

[Ding et al., 2021] Ding, S., Lin, D., and Zhou, X. (2021). Graph convolutional reinforcement learning for dependent task allocation in edge computing. *2021 IEEE International Conference on Agents (ICA).*

[Dinh et al., 2017] Dinh, T. Q., Tang, J., La, Q. D., and Quek, T. Q. (2017). Offloading in mobile edge computing: Task allocation and computational

frequency scaling. *IEEE Transactions on Communications*, 65(8):3571–3584.

[Donald et al., 1997] Donald, B. R., Jennings, J., and Rus, D. (1997). Information invariants for distributed manipulation. *The International Journal of Robotics Research*, 16(5):673–702.

[Donovan et al., 2017] Donovan, S., Chung, J., Sanders, M., and Clark, R. (2017). Metrosdx: A resilient edge network for the smart community. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 575–580. IEEE.

[Fleszar et al., 2009] Fleszar, K., Osman, I. H., and Hindi, K. S. (2009). A variable neighbourhood search algorithm for the open vehicle routing problem. *European Journal of Operational Research*, 195(3):803–809.

[Gale, 1989] Gale, D. (1989). *The theory of linear economic models*. University of Chicago press.

[Gerkey and Matarić, 2004] Gerkey, B. P. and Matarić, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The International journal of robotics research*, 23(9):939–954.

[Gu et al., 2015] Gu, L., Zeng, D., Guo, S., Barnawi, A., and Xiang, Y. (2015). Cost efficient resource management in fog computing supported medical cyber-physical system. *IEEE Transactions on Emerging Topics in Computing*, 5(1):108–119.

[Guo et al., 2019] Guo, J., Chang, Z., Wang, S., Ding, H., Feng, Y., Mao, L., and Bao, Y. (2019). Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th*

International Symposium on Quality of Service (IWQoS), pages 1–10. IEEE.

[Guo et al., 2016] Guo, X., Singh, R., Zhao, T., and Niu, Z. (2016). An index based task assignment policy for achieving optimal power-delay tradeoff in edge cloud systems. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE.

[Hausknecht and Stone, 2015] Hausknecht, M. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*.

[Hong and Prasanna, 2004] Hong, B. and Prasanna, V. K. (2004). Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 52. IEEE.

[Hu et al., 2019] Hu, C., Jiang, J., and Wang, Z. (2019). Decentralized federated learning: A segmented gossip approach. *arXiv preprint arXiv:1908.07782*.

[Ichoua et al., 2003] Ichoua, S., Gendreau, M., and Potvin, J.-Y. (2003). Vehicle dispatching with time-dependent travel times. *European journal of operational research*, 144(2):379–396.

[Jiang et al., 2020] Jiang, F., Liu, W., Wang, J., and Liu, X. (2020). Q-learning based task offloading and resource allocation scheme for internet of vehicles. In *2020 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 460–465. IEEE.

[Jošilo and Dán, 2018] Jošilo, S. and Dán, G. (2018). Selfish decentralized computation offloading for mobile cloud computing in dense wireless networks. *IEEE Transactions on Mobile Computing*, 18(1):207–220.

[Karger et al., 1999] Karger, D. R., Stein, C., and Wein, J. (1999). Scheduling algorithms.

[Khan et al., 2020] Khan, L. U., Yaqoob, I., Tran, N. H., Kazmi, S. A., Dang, T. N., and Hong, C. S. (2020). Edge-computing-enabled smart cities: A comprehensive survey. *IEEE Internet of Things Journal*, 7(10):10200–10232.

[Konečnỳ et al., 2016] Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

[Lalitha et al., 2018] Lalitha, A., Shekhar, S., Javidi, T., and Koushanfar, F. (2018). Fully decentralized federated learning. In *Third workshop on Bayesian Deep Learning (NeurIPS)*.

[Li et al., 2018] Li, L., Ota, K., and Dong, M. (2018). Deep learning for smart industry: Efficient manufacture inspection system with fog computing. *IEEE Transactions on Industrial Informatics*, 14(10):4665–4673.

[Li et al., 2021] Li, Q., Wen, Z., Wu, Z., Hu, S., Wang, N., Li, Y., Liu, X., and He, B. (2021). A survey on federated learning systems: vision, hype and reality for data privacy and protection. *IEEE Transactions on Knowledge and Data Engineering*.

[Li and Huang, 2017] Li, S. and Huang, J. (2017). Energy efficient resource management and task scheduling for iot services in edge com-

puting paradigm. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 846–851. IEEE.

[Li et al., 2020] Li, T., Sahu, A. K., Talwalkar, A., and Smith, V. (2020). Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60.

[Li et al., 2019] Li, X., Huang, K., Yang, W., Wang, S., and Zhang, Z. (2019). On the convergence of fedavg on non-iid data. *arXiv preprint arXiv:1907.02189*.

[Littman, 1994] Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier.

[Liu et al., 2019] Liu, L., Tan, H., Jiang, S. H.-C., Han, Z., Li, X.-Y., and Huang, H. (2019). Dependent task placement and scheduling with function configuration in edge computing. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE.

[Liu et al., 2020a] Liu, X., Yu, J., Feng, Z., and Gao, Y. (2020a). Multi-agent reinforcement learning for resource allocation in iot networks with edge computing. *China Communications*, 17(9):220–236.

[Liu et al., 2020b] Liu, X., Yu, J., Wang, J., and Gao, Y. (2020b). Resource allocation with edge computing in iot networks via machine learning. *IEEE Internet of Things Journal*, 7(4):3415–3426.

[López-Pérez et al., 2012] López-Pérez, D., Chu, X., Vasilakos, A. V., and Claussen, H. (2012). On distributed and coordinated resource allocation for interference mitigation in self-organizing lte networks. *IEEE/ACM Transactions on Networking*, 21(4):1145–1158.

[Ma et al., 1982] Ma, P.-Y. R. et al. (1982). A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, 100(1):41–47.

[Ma et al., 2020] Ma, S., Guo, S., Wang, K., Jia, W., and Guo, M. (2020). A cyclic game for service-oriented resource allocation in edge computing. *IEEE Transactions on Services Computing*, 13(4):723–734.

[McMahan et al., 2017] McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. (2017). Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[Narang et al., 2017] Narang, M., Xiang, S., Liu, W., Gutierrez, J., Chiaraviglio, L., Sathiaseelan, A., and Merwaday, A. (2017). Uav-assisted edge infrastructure for challenged networks. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 60–65. IEEE.

[Nilsson et al., 2018] Nilsson, A., Smith, S., Ulm, G., Gustavsson, E., and

Jirstrand, M. (2018). A performance evaluation of federated learning algorithms. In *Proceedings of the second workshop on distributed infrastructures for deep learning*, pages 1–8.

[Nishi, 2018] Nishi, H. (2018). Information and communication platform for providing smart community services: System implementation and use case in saitama city. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1375–1380. IEEE.

[Nishio and Yonetani, 2019] Nishio, T. and Yonetani, R. (2019). 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.

[Pan and McElhannon, 2017] Pan, J. and McElhannon, J. (2017). Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, 5(1):439–449.

[Pellazar, 1994] Pellazar, M. B. (1994). Vehicle route planning with constraints using genetic algorithms. In *proceedings of national aerospace and electronics conference (NAECON'94)*, pages 111–118. IEEE.

[Pradenas et al., 2013] Pradenas, L., Oportus, B., and Parada, V. (2013). Mitigation of greenhouse gas emissions in vehicle routing problems with backhauling. *Expert Systems with Applications*, 40(8):2985–2991.

[Qian Zhang et al., 2022] Qian Zhang, S., Lin, J., and Zhang, Q. (2022). A multi-agent reinforcement learning approach for efficient client selection in federated learning. *arXiv e-prints*, pages arXiv–2201.

[Rahwan et al., 2015] Rahwan, T., Michalak, T. P., Wooldridge, M., and Jennings, N. R. (2015). Coalition structure generation: A survey. *Artificial Intelligence*, 229:139–174.

[Rashid et al., 2018] Rashid, T., Samvelyan, M., Schroeder, C., Farquhar, G., Foerster, J., and Whiteson, S. (2018). Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning*, pages 4295–4304. PMLR.

[Reiss et al.,] Reiss, C., Wilkes, J., and Hellerstein, J. Google cluster-usage traces: Format+ schema, technical report.

[Roy et al., 2019] Roy, A. G., Siddiqui, S., Pölsterl, S., Navab, N., and Wachinger, C. (2019). Braintorrent: A peer-to-peer environment for decentralized federated learning. *arXiv preprint arXiv:1905.06731*.

[Salman et al., 2002] Salman, A., Ahmad, I., and Al-Madani, S. (2002). Particle swarm optimization for task assignment problem. *Microprocessors and Microsystems*, 26(8):363–371.

[Sperduti and Starita, 1997] Sperduti, A. and Starita, A. (1997). Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735.

[Sundar and Liang, 2016] Sundar, S. and Liang, B. (2016). Communication augmented latest possible scheduling for cloud computing with delay constraint and task dependency. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1009–1014. IEEE.

[Sunehag et al., 2017] Sunehag, P., Lever, G., Gruslys, A., Czarnecki, W. M., Zambaldi, V., Jaderberg, M., Lanctot, M., Sonnerat, N., Leibo, J. Z., Tuyls, K., et al. (2017). Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*.

[Tampuu et al., 2017] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. (2017). Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395.

[Tang et al., 2020] Tang, Z., Lou, J., Zhang, F., and Jia, W. (2020). Dependent task offloading for multiple jobs in edge computing. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE.

[Tao et al., 2017] Tao, X., Ota, K., Dong, M., Qi, H., and Li, K. (2017). Performance guaranteed computation offloading for mobile-edge cloud computing. *IEEE Wireless Communications Letters*, 6(6):774–777.

[Tran et al., 2017] Tran, T. X., Pandey, P., Hajisami, A., and Pompili, D. (2017). Collaborative multi-bitrate video caching and processing in mobile-edge computing networks. In *2017 13th annual conference on wireless on-demand network systems and services (WONS)*, pages 165–172. IEEE.

[Tran and Pompili, 2018] Tran, T. X. and Pompili, D. (2018). Joint task offloading and resource allocation for multi-server mobile-edge computing networks. *IEEE Transactions on Vehicular Technology*, 68(1):856–868.

[Wang et al., 2020] Wang, H., Kaplan, Z., Niu, D., and Li, B. (2020). Op-

timizing federated learning on non-iid data with reinforcement learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1698–1707. IEEE.

[Watkins and Dayan, 1992] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

[Wen et al., 2012] Wen, Y., Zhang, W., and Luo, H. (2012). Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In *2012 proceedings IEEE Infocom*, pages 2716–2720. IEEE.

[Wu et al., 2020] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*.

[Xiao et al., 2003] Xiao, M., Shroff, N. B., and Chong, E. K. (2003). A utility-based power-control scheme in wireless cellular systems. *IEEE/ACM Transactions On Networking*, 11(2):210–221.

[Zhang et al., 2018] Zhang, Y., Chen, X., Chen, Y., Li, Z., and Huang, J. (2018). Cost efficient scheduling for delay-sensitive tasks in edge computing system. In *2018 IEEE International Conference on Services Computing (SCC)*, pages 73–80. IEEE.

[Zhang et al., 2020] Zhang, Y., Zhang, P., Luo, Y., and Luo, J. (2020). Efficient and privacy-preserving federated qos prediction for cloud services. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 549–553. IEEE.