國立臺灣大學電機資訊學院電信工程學研究所

碩士論文

Graduate Institute of Communication Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

以深度強化學習進行公平考量之行動邊緣運算資源分配最佳化

Fairness-Aware Deep Reinforcement Learning for Task Offloading and Resource Allocation in Mobile Edge Computing

張安浩

An-How Chang

指導教授: 蔡志宏 博士

Advisor: Zsehong Tsai Ph.D.

中華民國 112 年 8 月

August, 2023

doi:10.6342/NTU202302089

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

以深度強化學習進行公平考量之行動邊緣運算資源分配最佳化

Fairness-Aware Deep Reinforcement Learning for Task Offloading and Resource Allocation in Mobile Edge Computing

本論文係張安浩 R10942151 在國立臺灣大學電信工程學研究所完成之碩士學位論文，於民國 112 年 7 月 26 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Institute of Communication Engineering on 26 July 2023 have examined a Master's thesis entitled above presented by An-How Chang R10942151 candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

蔡志宏		
（指導教授 Advisor）		
林守德	馬席彬	鍾耀梁

系主任/所長 Director: 葛煥昭

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

Acknowledgements

在碩論完成之際，回首兩年來有許多感恩之處，沒有一路上的眾人相助，我想我也無法獨自完成碩論。首先要感謝蔡志宏老師一路以來的指導，很慶幸當初能夠進到老師的門下學習，不但給我學術學習上精闢的指教，更有扣回實務業界的角度思考，讓我所做的研究更有實際應用的價值。此外，每次的實驗室會議老師信手捻來的最新時事分享都很有收穫。感謝老師在我形塑碩論題目時提供許多業界在意觀點，在架構模型時總是夠提供一針見血得評價指導，為我解惑許多。

感謝同屆的實驗室夥伴家頡和翊銘，兩年來一起修課寫碩論，在相似的碩論架構背景下互相討論幫助我很多，各種大小代辦事項都很罩。如今我們都要一起畢業了，這段時間承蒙照顧了！

感謝爸爸媽媽姊姊和 Joy，在碩士辛苦低潮的時候都持續陪伴我前進，肯定我相信我會度過難關。很慶幸有離學校很近的家和家人，使我碩士的日子有堅強依靠。

感謝教會的朋友們在我碩士時依然是我喜樂的來源，在我懷疑自己時總是堅定的相信鼓勵我。在碩論念書之餘，有和你們的歡笑點滴回憶填滿空隙是這段路途上美麗的祝福。感謝神一路帶領，在大小事上都有安排。

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

摘要

隨著日益進步的使用者產品和使用者需求，手機、平板、手錶等行動裝置使用者 (mobile user) 逐漸要求越來越大量的運算需求已完成更高品質的成果，產品本身的硬體運算資源便可能不足以應付低容忍度完成時間需求。為了突破邊緣的硬體限制，行動邊緣運算 (Mobile Edge Computing) 逐漸獲得重視，藉著鄰近的邊緣運算的分擔邊緣裝置 (Edge Device) 達成運算需求。本研究建構一個系統包含多個 MEC 伺服器並模擬多個使用者在系統中提出計算需求的環境，並提出以深度強化學習 (Deep Reinforcement Learning) 為系統架構的中央決策分配模型。藉由優化最大數量的工作數量的目標，並輔以公平分配指標檢視分配抉擇，以達到公平分配的最佳優化。此研究更進一步擴展實驗，藉由測試存在多種資料大小迥異的工作環境下，測試出模型期依然保持良好且公平分配決策，進一步驗證決策模型的可性度。此外，我們測試數種加權獎勵函數 (Weighted Reward Function) 並觀察模型不同設定下的表現。總結來說，實驗顯示提出之深度強化學習模型能夠在資料大小迥異的環境下提供公平的資源分配。

關鍵字：計算分載最佳化、雲端資源分配、決策演算法、強化式學習、使用者平等

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

Abstract

As the requirement for computation grows exponentially in quantity and quality, mobile-edge computing (MEC) has perceived more attention to become a computational mechanism for near mobile users. With the additional computing power from nearby edge servers, edge users can potentially handle tasks that need to meet massive and real-time computation requirements. In this thesis, we consider a multi-MEC systems with multiple users, each users generate computational tasks follow by Poisson process. The thesis propose a deep reinforcement learning-based allocation model to allocate tasks gathering from all users in a centralized decision algorithm. We formulate the optimization goal as maximizing the number tasks finished within deadline while task fairness awareness is taken into considerations. In order to handle continuous task flows, we formulate a multi-round allocation scheme and simulate its operation in the real world complex system. For validation purposes, this thesis made several simulations on huge task size difference network to test model's performance and comparing to random allocation. In addition, we

doi:10.6342/NTU202302089

tested a various type of award functions, including a basic reward function and a set of weighted reward function, to observe their performance under different setups. In conclusion, the simulation results show that DRL model is effective in providing for task allocations for various task-size distributions.

Keywords: Computation Offloading, Resource Allocation, Reinforcement Learning, Fairness

doi:10.6342/NTU202302089

Contents

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

List of Figures

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

List of Tables

doi:10.6342/NTU202302089

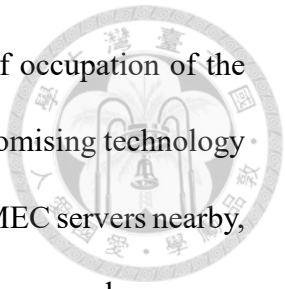
doi:10.6342/NTU202302089

Chapter 1 Introduction

1.1 Motivation

As the prevalent of 5G internet, the capability of the Internet of Things (IoT) has grown rapidly. Various kinds of devices, like cellphones, tablets and watches, can now connect to wireless networks, process information, and handle computation-intensive applications. However, with limited computational resources and battery restrictions on the devices, processing intensive computation tasks might cause a problem since the limited processing power cannot finish processing the task under delay constrain. Also, running massive computational requirements require huge energy whereas energy is scarce. In order to alleviate these edge devices process computation-intensive tasks, one of the common solutions is to offload the task to the cloud server that is located at a distance data center that have massive computational resources and sufficient energy supply. However, this can have several concerns. First, lots of the computation-intensive tasks are real-time applications, meaning that the delay tolerance is low, hence sending these tasks to the cloud might cause more time in total including transmission time. Second, suppose all edge devices rely on a distant cloud server to help process computationally intensive tasks. In that case, the wireless network speed will become the bottleneck of the computational time, as every edge device is counting on the cloud server.

doi:10.6342/NTU202302089



To tackle this problem and alleviate the massive transmission of occupation of the internet to the cloud, mobile edge computing (MEC) is viewed as a promising technology that enables mobile devices to offload computation-intensive tasks to MEC servers nearby, where these servers have plenty computational power and stable power supply comparing to the mobile devices. The MEC can increase addition computation resources near the mobile user and provide high-performance servers to mobile users [1]. By distributing the MEC servers close to the users, through computation offloading the mobile users can leverage and successfully run computational-intensive apps, live-stream, gaming and more. Therefore, due to the potential advantages of increasing computation offloading destinations and quicker computational resource allocation solutions, in [2] this has been viewed as a important part in MEC system.

Over the past decades, Deep Learning (DL) has been unprecedented popular as the capability and flexibility a DL model improves model performance among various domains [3]. Deep learning methods have some advantages: 1) Given a history dataset, an DL model can extract and learn the patterns of the underlying structure, and then make decisions based on the observations. By feeding an DL model large chunk of real time data, the mode can improve and adjust its decisions to fit domain preferences and tendencies, and no need to manually interfere or comes up heuristic guidelines. 2) For a well-trained DL model, it has potential to make quick and good decisions, which perfectly meets the desired characteristic an offloading decision model needs. As the mobile users' requests dynamically changes rapidly, local computation capability in mobile phone might not be able to handle within strict deadlines, hence fast results get more important than never before. 3) The process of optimizing the wireless network is a difficult but crucial problem to solve. Since the quantity and quality of mobile users shifts in a pattern difficult

doi:10.6342/NTU202302089

to describe and simulate. Throughout various methods proposed before, such as heuristic methods, mixed-integer linear programming, all struggle to handle this constantly changing network structure. 4) As mentioned in [3], It is complicated managing the joint 4C optimization in 5G. Diverse requests including individuals QoS needs. In such case, ML has the potential to solve respective needs more efficiently.

[4] Deep Reinforcement Learning (DRL) has gained huge success in recent year, where powerful DRL-based models thrive in gaming [5], dynamic path finding [6], optimization [7] and more. Advantages are that DRL model make quick and good decisions, and the model can continuously learning through latest real-time history data, so that it dynamically adjust throughout the time. In recent years, DRL is also introduced to solve offloading engine and resource allocation for network([8], [9]), aiming to maximize the utilization of the network resources [4].

Typically DRL models solving computation offloading and resource allocation aim to optimize the allocation decision with minimal computational delay and energy consumption [9] at all cost. However, the fairness among users is also a crucial factor that cannot be neglected to maintain. As a perspective of the computation resource supplier, providing a great service leads to higher QoS.

In the real-time application, the status of MEC servers and the cloud are dependent on previous tasks requests at the moment, those tasks occupy part of the computational resources, which further affect how much available capacity each server can provide. In practice, the network is always dynamically changing its resource status. A well-designed scheduling model should be able to allocate resource where the status of each MEC servers and the cloud are dynamically changing after each round of resource allocation. Hence,

doi:10.6342/NTU202302089

in this thesis I will design a simulation scheme that the scheduling model can make good decisions under multi-round allocation simulation.

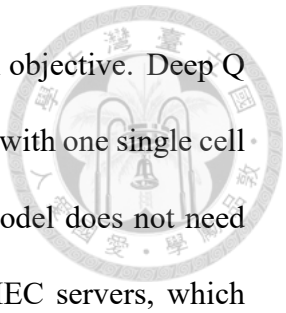
1.2 Background and Literature Review

Related literature in this field have studied various aspects of the optimization problem. For example, [10] Chang et.al proposes a bi-level optimization approach that divide the problem into upper level and lower level and solve the problem applying Lagrangian function and Hungarian method recursively. However, one potential drawback is that continuously solving Lagrange multiplier increases computation requirement and possibly extend the processing time, which is not favorable for latency-sensitive tasks.

Another research Huang et. al [11] proposed a Deep Reinforcement Learning (DRL)-based Online Offloading (DROO) that can learn the binary offloading decision to decide to execute locally or at MEC server. Their algorithm does not involve solving mixed integer programming problems, which helps avoiding spending too much time to solve that. The method includes the offloading decision and a resource allocation problems separately. The optimization goal is set to be maximize channel realization. To offload N tasks simultaneously, the action space for offloading decision along cause 2^N, which is exponentially large and therefore time-costly to explore optimal action. Since the number of task N is a pre-defined fixed scalar, the simulation in the research didn't involve a multi-round simulation where a random number of tasks comes and go, which motivates this study to further test the robust of DRL-based method in the multi-round environment.

[8] Gao et. al consider a multi-user MEC system, where multiple mobile users could perform computation offloading to an MEC server. The author formulated the sum of

doi:10.6342/NTU202302089



cost by combining delay and energy consumption as the optimization objective. Deep Q network model is implemented and simulated on a small environment with one single cell MEC server and 5 users. Due to the setup of the simulation, this model does not need to handle communications and task allocations between multiple MEC servers, which simplified a common real world scenario. Therefore, it is doubtful the performance can transforms to more complex system. In the Deep Q network setup, the state is set as the total weighted cost of the entire system and the total available computation capacity of the MEC server, denoted as (tc, ac). However, this state combine too less information to allocate if the task have distinct characteristic. For example, in the simulation the task size is sampled withing one small fixed range, each tasks have size difference less than two times difference. In this way, the RL model might learn the implicit task size setup and might not perform well when different task size appears. Therefore, in this thesis we add multiple task size range with great difference so that the model can learn through a more complex system.

Zhou et. al [9] also investigated the joint optimization of computation offloading and resource allocation in a dynamic multi-user MEC system. The objective in [9] is to minimize the energy consumption of the entire MEC system. The author formulated the problem as a mixed-integer nonlinear programming problem, and propose a DRL-based method to determine the joint policy of computation offloading and resource allocation. The model was simulated in a single MEC system with 5 users. Since the objective function focus on the minimal energy consumption, the delay constrain is viewed as a side objective. However, the computation time is the major performance issue. Hence from the user's perspective, minimizing computation time is a crucial factor. On the other hand, the simulation environment in [9] is extremely simple, where only 5 users are implemented

doi:10.6342/NTU202302089

to show its energy consumption and delay performance are shown.

Aside from maximizing the optimal performance on latency or energy objective, [12] points out the importance of consider the fairness among task, and propose a joint optimization problem that taking into account the system efficiency and fairness. The author defined the task fairness ratio as the time consumption divided by the task deadline requirement. A proposition is then proof that the task fairness is minimized only if each task's task fairness are equal. To integrate this concept, the author proposed a two-level algorithm, where the first part find the superior offloading scheme through evolutionary strategies, and the second part generate resource allocation scheme through a fairness process. Inspired by this research, this thesis will apply the fairness ratio defined in [12] and integrate with Deep RL structure that hasn't been done before.

1.3 Research Objective

Combining several observations we've seen through papers above, The research objective can be summarized as:

1. **Fairness-Aware DRL Allocation Model**

   Throughout previous researches applying deep reinforcement learning-based algorithm to solve resource allocation problems, no research had considered the fairness between tasks. The main focus was more about minimizing time consumption, reducing overall system energy consumption, maximize number of tasks executed but valuing the task fairness. However, since we view all users are equally important and not being left to starvation, hence whether the allocation protect task fairness becomes one of the research objective in this thesis.

doi:10.6342/NTU202302089

2. **Performance Validation for Multi-Round Operations**

Consider designing a robust resource allocation algorithm to perform well in real time network, the model should be able to thrive and adjust in state-dependent multi rounds situation. Although a deep reinforcement learning model automatically needs to interact and improve in a state-to-state environment, duo to the difficulty and complexity of simulating multi round environments, previous works often train in a scheme where the initial state does not comes after the environment inputs an action from model. Instead, the initial state are randomly initialized or starting at the same empty state.

3. **Class State based on Types of Task Sizes**

In the setup of reinforcement learning, the environment is described as a state vector, and the RL model can make action based on the state. Therefore, what information a state provides affect the performance of the model. Although it seems that the state should be as thorough as possible, including detailed background parameters, huge dimension of the space spanned by state possible values might make the problem difficult to solve. This is due to the fact that, RL model needs to probe through possible states as much as possible to learn through experience, large state space took longer training time and more computational resource.

Therefore, in our model we simplified the state description about task's size. Instead of sending the actual tasks size to the state, each task is categorized by three types of task sizes: Small, Medium, and Large. The reason of not writing the actual task size into state is that, in network flow where massive flows come from places to places, measuring the actual task size directly is impractical and impossible [13]. Besides, if the model were to always know the actual task size, the model can directly estimate

doi:10.6342/NTU202302089

its requirement for computational power, then indirect allocation methods are no need to apply. By denoting the tasks as its task size type, one can simplify the dimension of the data space of the state because the number of tasks size type is an integer constant, whereas the possible task size type falls only within the positive number space.

doi:10.6342/NTU202302089

Chapter 2 System Architecture

2.1 Environment

Consider a MEC network that consists N_M MEC Servers, N_B Base Stations, and N_U Users in the network. The number of MEC server is no greater than the number of the base station, i.e. $N_M \leq N_B$, since we assume MEC Servers are scarce resources, these MEC servers will be shared by all nearby base stations. As each user send tasks to base stations, the controller of the base station has to decide how should the task been allocate. Figure 2.1 shows the network environment.

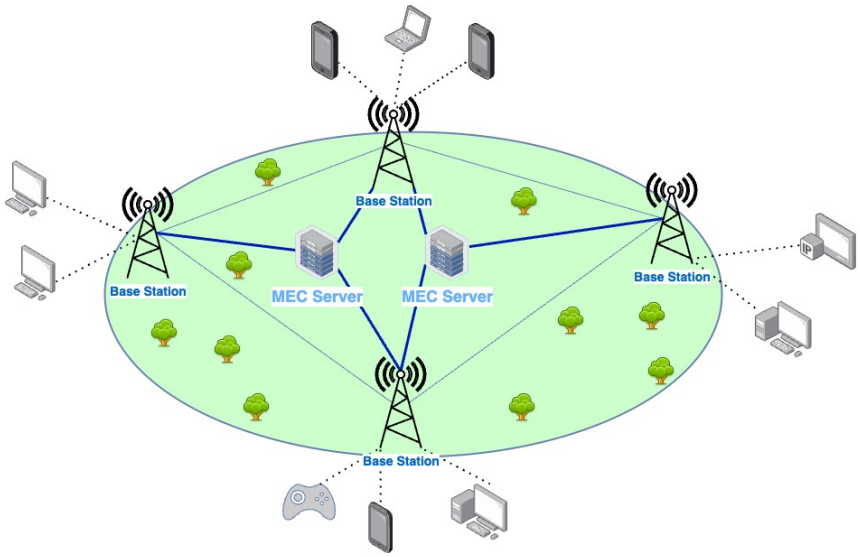


Figure 2.1: System Environment

doi:10.6342/NTU202302089

2.2 User & Task

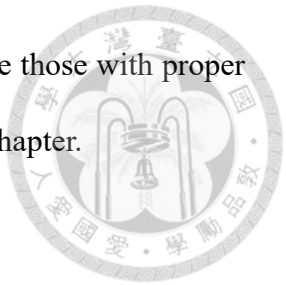
2.2.1 Users with different task size requirements

In a dynamic network, the tasks size can have 10 to 100 times size difference than the others when the user's need are different. In this situation, how to allocate tasks to MEC servers can make great difference in optimization. Therefore, in this thesis, the task size is generated from multiple type of task size, each task size type have great size difference up to 10 times. From the user perspective, generating tasks from different task size type symbolize different type of user request format. For example, enterprise users ask for huge task size for computational-intensive applications, whereas in the same time smaller and less intensive requests are also needed. To simulate this task size different, integrating different tasks size type into the simulation simultaneously creates more complex challenge to the allocation model. This setup fit closer to the real world network scenario as well.

In actual network, it is common to have both kind of the users asking for computational resources in the same time. To effectively allocate each kind of task size type to the optimal allocation, the decision is effected by what the optimization goals are. A straightforward way is to minimize overall computational time with as much tasks finished within its delay tolerance threshold as possible. This way of allocation alone, however, might favor the enterprise users and those large tasks. In general, enterprise users tend to ask for massive and strict time tolerance. In order to maximize overall performance with the least amount of tasks failed to finish in time, the model will always prioritize allocating majority of the computational resources to the enterprise user, which leads to individual user starvation. Hence in my environment, by introducing the concept of different task

doi:10.6342/NTU202302089

size types, we can see how different task type are treated and evaluate those with proper fairness index which we will go through the details in the following chapter.



2.2.2 Task Arrival Pattern

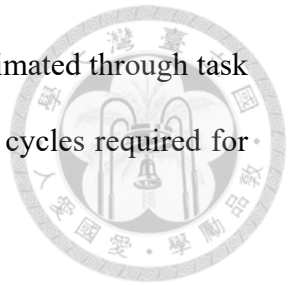
To simulate an environment interact with the network system similar to real world, each user will generate tasks following Poisson arrival process. Denote the Poisson arrival process with arrival rate λ as $Poi(\lambda)$. Different arrival rate λ will be sampled with respect to different traffic scenario. Consider three level of arrival rate: Computation-intensive duration, Workday duration, and Off-work duration. The Computation-intensive duration when a series of real-time tasks are requesting, for example training a deep neural network model, streaming a live event, multiple people online meeting. Here we will set an average arrival rate of the Computation-intensive duration as λ_c. The Workday duration refers to general workday request flows, where users require computational resource in a less intensive delay tolerance and sparse request frequency. The average arrival rate of the Workday duration is denoted as λ_w. The Off-day duration is when the users aren't working and the remaining requests come from regular maintains of the server. The average arrival rate of the Off-day duration is λ_o. In the simulation, various size of arrival rate represent different arrival pattern to evaluate the allocation model's performance.

2.2.3 Task Structure A_i

For a computation-intensive task A_i required by a user, it is described by three features: D_i denotes the computation input Data Size, X_i denotes the computation intensity in CPU cycles per bit, τ_i is the requested completion deadline in the second unit. Here

doi:10.6342/NTU202302089

we denoted a task as $A_i := (D_i, X_i, \tau_i)$. These parameters can be estimated through task profiles. Noted that by definition, $D_i X_i$ is the total number of CPU cycles required for completing the task A_i.



2.3 At Base Station

2.3.1 Local Computation v.s. Remote Computation

The tasks generated by the users will being processed in one of the three ways: locally by the user's device, offloaded to a MEC server, or offloaded to the cloud. In general, we can view offloading to either MEC server or cloud as processing the task "Remotely", comparing to processing the task "locally" at the user's device. In this environment, the user's device will use a simple criteria to decide whether a task should be offloaded to the remote or process it locally. The following paragraph will further elaborate this decision scheme.

2.3.2 Single Round to Multi-Round

In a single round $[r_i, r_{i+1}]$, the base station will collect new task requirements, and decide how to allocate the tasks and assign to the desire remote server. The task does not need to finish the computation within one single round time. In the following chapter, the large scale is called the public round.

For each public round, the system time is equally divided into consecutive time frames lengths as multiple smaller rounds, which is called the *Mini Round* $r \in \mathbb{N}$. Denote the duration of an mini round as T_r. T_r has to be long enough for a the system do all things

doi:10.6342/NTU202302089

mentioned above. Aside from that, every public rounds we'll update latest status of each MEC servers and the cloud, for $R \in \mathbb{N}$. The length of an Public Round is denoted as T_R. Given a fixed positive integer u, the length of one public round $[R_i, R_{i+1}]$ is u rounds of mini rounds, i.e.

$$T_R = |T_{R_2} - T_{R_1}| = |T_{r_u} - T_{r_1}| = u|T_{r_2} - T_{r_1}| = uT_r$$

The visual demonstration is shown as Figure 2.3:

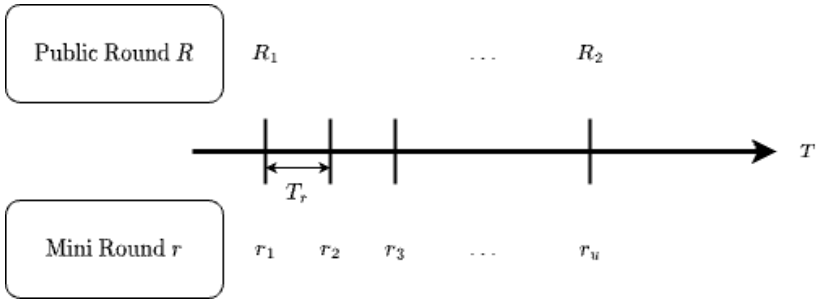


Figure 2.2: An illustration of the Public Round and the Mini Round

2.3.3 Local Offload Threshold η_i

For each task A_i waiting to be processed at the user device, the device filter the task by a binary offloading policy deciding either it is offloaded completely to one of the remote server, or process the task locally. Let $x_i \in \{0, 1\}$ be the indicator that denotes whether a task i is offloaded remotely. When $x_i = 1$, task A_i is offloaded to a remote server and $x_i = 0$ indicates that it's processed locally. For each task A_i at user device, whether to offload to remote server is filtered by a delay threshold η_R at round r. We estimate how long the task A_i will be executed locally, and denote it as T_i^l. Then, compare this estimated local execution time with the delay threshold η_R, and the task will be offloaded to remote server if the estimated local computation time is greater than the delay threshold. Noted that this delay threshold will be fixed for u rounds and then be updated to the latest

doi:10.6342/NTU202302089

threshold η_{R+1}. This latest threshold is generated by the the decision model at the base station. Here, setting a simple offloading strategy to decide whether the task is executed locally or remotely is decided as follows:

For each task A_i,

η_R - Offloading Delay Threshold at Update Round R

T_i^l - Local Execution Time of A_i

x_i - Offloading Indicator

$$x_i = \begin{cases} 1 & \text{if } T_i^l \geq \eta_R \\ 0 & \text{otherwise} \end{cases}, \forall i$$

For each task A_i, divide the expected number of cycle $D_i X_i$ by the local computation frequency of the respective user i, f_i^l, we can get the estimated computation time at local.

$$T_i^l = \frac{D_i X_i}{f_i^l}$$

Noted that for each user, we assume the computation frequency is the same among all tasks generated by this same user. If the user device has amount of computational resource to match the fixed computation frequency f_i^l then the user device will not assign the remaining resources and viewed as full. In the future work, we can add into the model how each user device assign computational resources to each local dynamically.

doi:10.6342/NTU202302089

Chapter 3 The Allocation Model

In the network, a task will be processed at one of the two places: local user device or one of the MEC servers. After a task is sent to a base station, the system has to decide which MEC server to allocate this task. To evaluate the performance, we'll go through how to evaluate the processing performance at different places. To determine where to allocate each task efficiently and fairly, a centralized allocation model will decide where to allocate.

For each task, since it will be executed either in local or at the MEC, we define two computing mode: the Local Computing Mode is when the task is executed locally, and the MEC Computing Mode is when the task is offloaded to MEC servers to execute. In the following sections, we will go through how each mode a task is evaluated its time consumption details.

3.1 Local Computing Mode

When a task called A_i is processed locally at a user device, one can estimate the local computation delay T_i^l by the required CPU cycle of the task A_i and the local computation frequency f_i^l. Since the task doesn't transmit through the network, their is no transmission

doi:10.6342/NTU202302089

delay. The local computation delay is:

$$T_i^l = \frac{D_i X_i}{f_i^l} \tag{3.1}$$

In this allocation model, the total cost of local computing C_i^l for task A_i is the weighted cost of delay cost where w_t^l is a given fixed constant. The cost can be given by

$$C_i^l = w_t^l T_i^l \tag{3.2}$$

3.2 MEC Computing Mode

For the case of offloading a task to a MEC server, the total amount of time needed include the transmission time of the task from the edge device to the MEC server, the computational time at the server, and the transmission time sending back the results. Denote the transmission delay of task A_i to the remote server k as $T_{i,k}^{t_u}$, and it is given by

$$T_{i,k}^{t_u} = \frac{D_i}{R_{i,k}^u} \tag{3.3}$$

where $\mathrm{R}_{i,k}^u$ is the up-link rate for task A_i to the remote server k. Secondly, the computation delay at the remote server k can be derived as

$$T_{i,k}^c = \frac{D_i X_i}{f_{i,k}} \tag{3.4}$$

where $f_{i,k}$ is the allocated computation frequency at the server k. Denote the transmission delay of the result D_r from the MEC server k to task A_i as $T_{i,k}^{t_d}$,

$$T_{i,k}^{t_d} = \frac{D_r}{R_{i,k}^d} \tag{3.5}$$

doi:10.6342/NTU202302089

where $R^d_{i,k}$ is the down-link rate from the remote server k to task A_i . The Total MEC Server Execution Delay is thus given by

$$T^r_i = \frac{D_i}{R^u_{i,k}} + \frac{D_i X_i}{f_{i,k}} + \frac{D_r}{R^d_{i,k}}. \tag{3.6}$$

The objective function will be maximizing the number of tasks that have been successfully executed. Denote y_i as the indicator of task A_i finish executed or not, y_i is defined as:

$$y_i = \begin{cases} 1 & \text{if } T_i \leq \tau_i \\ & \\ 0 & \text{otherwise} \end{cases}, \forall i \tag{3.7}$$

Then the objective problem is formulated as follows.

$$\max_{\mathcal{A}} \sum_{i=1}^{N} y_i \tag{3.8}$$

$$s.t.\ C_1 : x_i \in \{0, 1\},\ \forall i$$

$$C_2 : T_i = (1 - x_i)T^l_i + x_i T^r_i \leq \tau_i,\ \forall i$$

3.3 Deep Reinforcement Learning Model

With the environment setting done as the state input, the reinforcement learning algorithm can then be applied to approximate the allocation decision-making. In the following section, we will introduce the widely-used DDQN-based method to try to solve the allocation to the MEC servers.

doi:10.6342/NTU202302089

3.3.1 State, Action, and Reward Definition

To begin with, there are three key elements construct the basic structure of the employed reinforcement learning model: state, action, and reward.

1. State

   The system environment is described as a vector combining several index. The first part is the total number of undone tasks in the system, denoted as N^r. Next, a vector $[N_S^r, N_M^r, N_L^r]$ counts the number of non-allocated tasks by its data size type: small, medium and large, respectively. Since the data size is classified as three types, here we can use three variables to store the count of tasks that aren't allocated. Next, the state record the counting number of tasks in each MEC queue by its data size type. Each MEC requires three values to store, hence need $N_{MEC} \times 3$ scalars. Denote MEC i's number of tasks in queue by task size type as $[M_S^{(i)}, M_M^{(i)}, M_L^{(i)}]$. Last but not least, the state append a vector describe which task size type the following task that is preparing to assign. The vector will be a one-hot vector like $[1, 0, 0]$ for type 1 task type, and $[0, 1, 0]$ for task 2. Here denote the task size type vector as $[P_T(c = 0), P_T(c = 1), P_T(c = 2)]$, where $P_T(c = 0)$ states the probability of the task T's task type is 0. Combining all information mentioned above, the state at mini-round r before allocate the following task T will be:

   $$s_T^r = (N_S^r, N_M^r, N_L^r, N^r, M_S^{(1)}, M_M^{(1)}, M_L^{(1)}, ..., M_L^{(K)}, P_T(c = 0), P_T(c = 1), P_T(c = 2))$$

   This state will be used specifically for this task T, the allocation model take this state to make the allocation decision. The task will be sent to MEC and the number count that are affected by this allocation will be updated so that the state for the next

doi:10.6342/NTU202302089

task will be updated.

2. Action

As we apply reinforcement learning to the allocation model in MEC system, an action to describe how the allocation model can interact with the environment is needed. The action should decide which MEC server to allocation for this task, which is presented as a one-hot vector fashion. Suppose the number of MEC servers in the system is K, a task's allocation decision is presented as $[a_1^{(1)}, a_2^{(1)}, ..., a_K^{(1)}]$. Hence the action vector a is as follow:

$$a = [a_1^{(1)}, ..., a_K^{(1)}]$$

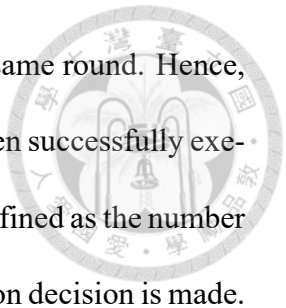
$$a_i^{(j)} \in \{0, 1\}$$

$$\sum_i a_i^{(j)} = 1$$

3. Reward

In reinforcement learning, reward R is used to present how many positive result were obtained. Given a current state s_t, after making an action a, the amount of reward received after a fixed duration of time section is called the immediate reward R_t. In general, the reward is related to the objective function. In this optimization problem, we aim to maximize performance of finishing as many tasks as possible and the fairness between tasks and users. In other words, we aim to increase completion rate and fairness. To maximize completion rate means that the MEC system needs to maximize the utilization rate of the MEC. Higher utilization rate means

doi:10.6342/NTU202302089



that the percentage of time the MEC is occupied is high in the same round. Hence, the value of reward is defined to be the number tasks that has been successfully executed within its respective deadline. The immediate reward is defined as the number of tasks that are finished within one mini round after the allocation decision is made. We define the immediate reward after taking action a at state s as $r(s,a) = N_{Task}^{(r)}$. Noted that $N_{Task}^{(r)} = \sum_i y_i$.

The reward function defined above count the total amount of finished tasks, we call this the *basic reward function*. An potential issue that the basic reward function might face is that it treat each finished tasks equally without regards of task size. This way would favor smaller tasks to be executed over large tasks since with the same amount of computational resource, the number of small tasks can be executed more than larger ones. Therefore, adding weight to each reward can possibly balance this inequality issue. In the chapter 4, various kind of *weighted reward function* will be tested to see which performs better than others.

If we re-examine several previous related work, such as [8], the reward vector space consist of the offloading decision part and the resource allocation part, where the resource allocation part output how much resource the server need to allocate to each task. Although this type of definition can possibly provide thorough action space so that the model have chance finding optimal allocation destination and quantity, the resource allocation decision drastically increase the action space dimension, causing more difficulty and computation overhead at each round of allocate decision making. Hence, in order to decrease the action space, the model will only make the offloading destination decision, and at each allocation round, each MEC server will equally allocation available resource at the moment. To avoid the proportion of allocate resource been sliced too small, a minimum slicing size

doi:10.6342/NTU202302089

of resource allocation is set so that resource won't be sliced smaller than the minimum slicing size.

3.3.2 Q-learning method, Deep Q-learning

One of the most popular method in reinforcement learning is the Q-learning algorithm ([4],[8]). Q-learning is a reinforcement learning techniques used to solve Markov Decision Processes (MDPs) ([8], [14]) without prior knowledge of the environment. Given a state s and an action a, a Q value is the output of the Q function $Q(\cdot)$ that states how much expected reward value are going to get from this state to the end state. Each state-action pair have a real value $Q(s, a)$, representing expected reward afterward. Normally when training the reinforcement learning, the Q-function only need to input the state s and the output will be a scalar vector with the dimension of the number of all possible action. Each value refers to the Q value of the state and one specific action a. We can then choose the action with the highest Q value as the desired best action choice. After each step, each encountered state-action pair will derive a Q value $Q(s, a)$, which can be stored in a Q-table so that it can be checked in the future lookup if the state reappears.

The Q-learning algorithm can be typically executed through an iterative process, such as Q-learning or Deep Q-learning, where the agent explore the environment, observes state transitions and rewards, and updates the Q-values based on the observed experience. The goal is to find an optimal policy that maximize the expected long-term reward by selecting actions in each state. The Q-function is often represented and stored in a Q-table so that whenever a state is occur, the agent can lookup to the Q-table to obtain history maximal reward from that state. Ideally, when the agent explore and interact the environment often enough, most of the state are visited and explored, we can then always often optimal

doi:10.6342/NTU202302089

actions.

However, the Q-table approach does not work well when the state-action space is too large and complex, because the agent is impossible or too time-costly to explore all state-action pairs. Therefore, researchers comes up ideas to use approximate functions to approximate the actual Q-table, where the function takes the state-action pair as function input, and output an expected accumulated reward. As the deep neural network getting powerful and easier to train in recent years, many researches have developed to use deep neural network model to approximate the Q-function. Deep Q-learning extends the basic Q-learning algorithm by utilizing deep neural networks to approximate the Q-function. Instead of using a tabular representation of the Q-values, a deep Q-network (DQN) is employed to estimate the Q-values for high-dimensional state spaces. In the following research, we denote DRL as the allocation method applying the Deep Q-learning, and the DRL will be applied.

Since Q value can be viewed as a long-term cumulative reward, the $Q(s_t, a)$ can be decomposed to two part: immediate reward and the future reward, i.e.

$$Q(s_t, a) = r(s_t, a) + \gamma * \max_{a \in \mathcal{A}} Q(s_{t+1}, a)$$

where the (s, a) is the current state-action pair, s' is the state after current state s takes the action a. At the next state s' the future reward is defined as the maximal possible reward among all possible actions at state s'. The γ is the depreciated multiplier reflecting how much the future reward takes account as the action made at the current state. The multiplier falls in $0 \leq \gamma \leq 1$. If the depreciated multiplier is set close to 1, it means the future reward hugely depends on the current action made at the state. On the other hand, if

doi:10.6342/NTU202302089

the multiplier is set close to 0, it means the action only affect immediate rewards, and the future reward should not take much credit from this action at the state. In our scenario, the action decide each tasks' offloading decision, and the immediate reward how well each server performs in the next round. In the simulation scenario in this research, s_r denotes a state at a mini round r, the Q-function can be defined as:

$$Q(s_r, a) = r(s_r, a) + \gamma * \max_{a \in \mathcal{A}} Q(s_{r+1}, a) \tag{3.9}$$

where $r(s_r, a)$ is the total number of tasks that are successfully executed in this mini round r among all MECs.

As mentioned before, Deep Q-Learning apply deep neural network models to approximate the Q-function. In order to do so, we need to be able to learn the Q-function from interacting with the environment. In order to train a Q-function that generate correct output, the engine need to know how 'accurate' the output the Q-function is during training. Normally in DNN training process, we can compare the model-generated value to ground true value, which is called the supervised learning. However, since we don't know the optimal accumulated reward from every state-action pair, Q-learning method is difficult to formulated as a supervised learning way. Instead, Temporal Difference (TD) learning is a popular approach to optimize the Q-function in reinforcement learning. The key idea behind TD learning is to update the Q-values based on estimation from subsequent states rather than waiting for the final outcome. According to the definition of equation 3.9, the Q value the state-action pair (s_r, a) should equal to the sum of immediate reward at this mini round $r(s_t, a)$ taking action a and the expected cumulative reward of state s_{r+1} at the next mini round $r + 1$. Therefore, during the training, the difference between $Q(s_r, a)$ should be as close as possible as $r(s_r, a) + \gamma * \max_{a \in \mathcal{A}} Q(s_{r+1}, a)$, we can then compute

doi:10.6342/NTU202302089

the average difference as the loss to back propagate and optimize the model. To sum up, we can update the parameters by state-action-reward pair as:

$$(s_r, a_r, r_r, s_{r+1}) \tag{3.10}$$

Noted that since the model will be called every time it allocate a task, the current state s_r is different from other tasks that are waiting to be allocated in the same mini round. As mentioned beforehand the state include tasks count of not-allocated ones in the BSs and the tasks count of tasks in each MEC queue to execute this mini round, every time after the model allocate a task, the environment state will be updated to the following state for the next task. In this way, the model can see how many tasks left not allocated and how each MEC servers have been distributed tasks up to date. The model then possibly can make adjustments for the next allocation. Since the immediate reward is defined as the total number of tasks being executed during the whole mini round, the proper definition of the next state is at the beginning of the next mini round $r + 1$. Hence, for each tasks in this mini round will be stepping to the same next state s_{r+1}. The last part of state where it denote the task type of the task prepared to assigned is set to all zeros in the next state s_{r+1}.

In [9], the author design the model to input fixed number of tasks as a batch to allocate simultaneously, and output action include where to allocate and how much CPU frequency to allocate. Although with this flexibility the model has a chance to obtain global optimal allocate solution, the action space grow exponentially as the number of tasks to allocate the same time. Hence, in this research I reduce the action space dimension so that the model could be more likely to handle larger amount of tasks and more number of MEC

doi:10.6342/NTU202302089

servers to choose without too long decision delay.

3.3.3 The Public Round and The Mini Round

In the simulation, Two level of round are used: Public Round and Mini Round. At the beginning each public round, all base stations collect tasks that have been sent from mobile users. These are the tasks preparing to allocate to MEC servers. The goal is to allocate and execute all tasks collected at the beginning of the public round. In another word, there will have no new tasks being considered to allocate within the same public round. All tasks that arrived the base station will wait until the next public round to allocate MEC server. The duration of a public round is 1 sec.

Each public round is split in 20 fixed even length small execution round called mini round. For every mini round the model allocate a batch of tasks to MEC servers. The detail procedure of **one mini round** can be split into following step:

1. **Randomly pick a task.** From all tasks in the base station waiting queue, randomly pick a task prepare to allocate to one MEC server.

2. **Wrap the state.** Write the initial state s_r for the task, check status including the MEC waiting queue tasks counting and the picked task's information.

3. **Select action.** During the training, the action will be selected either by the policy net's prediction output or by random selection. This method choice is controlled by a threshold, and the threshold will monotonically decrease the chance a random action is selected. The reason to include random selection during the training is to increase the exploring chance such that the model won't satisfy by local optimal solutions. The policy model takes the state as a one-dimension vector input and output

doi:10.6342/NTU202302089

a predicted Q value for each action. This predicted vector have the dimension of the number of possible actions in action space $\mathcal{A}$. Say the number of MEC server is N_{MEC}, then the dimension of action space will equal to N_{MEC} since the model allocate one tasks a time.

4. **Send to MEC server** . Send the task to the assigned MEC server. Record the transmission delay to add to this task's time cost and remainder tasks' queue delay.

5. **Update state.** Update the number count of unassigned tasks in base station and the number of tasks in MEC queue.

6. **Assign all tasks in batch.** By repeating process 1-5 to allocate and send all tasks in batch to MEC servers.

7. **MEC Execution** After all tasks in batch have been assigned to MEC server, each MEC server execute tasks in queue in parallel. For each server, it randomly pick one task in the queue to execute. If the task is estimated not able to finish within the ongoing mini round, then the MEC server will not proceed to execute this task but instead find another tasks in queue until it find the next task that can finish within that mini round or no other task left not-visited. The utilization rate of a MEC server in a mini round is defined as the percentage of time it is used to execute tasks.The utilization rate per mini round are recorded and and evaluated that can demonstrate how well MEC servers are utilized through out all 20 mini round in a public round.

After finish executing a mini round, all tasks that failed to be completely executed will be sent back to base station to the waiting queue so that in the future mini round the allocation engine can possibly allocate those tasks again. The reason to sent unfinished task back to base station instead of keep those at the server they're at is that, MEC server

doi:10.6342/NTU202302089

can be more likely to adjust their allocation decisions and wrong allocation decisions won't be easily accumulate causing worse performance. In our running scenario, since large tasks are a lot larger than small tasks, if too many large tasks are assigned into one MEC server might cause the server stuck by large tasks for too long. Hence, for those unfinished tasks at MEC servers at the end of a mini round, re-allocate them in the next mini round can lead to more stable and better completion rate.

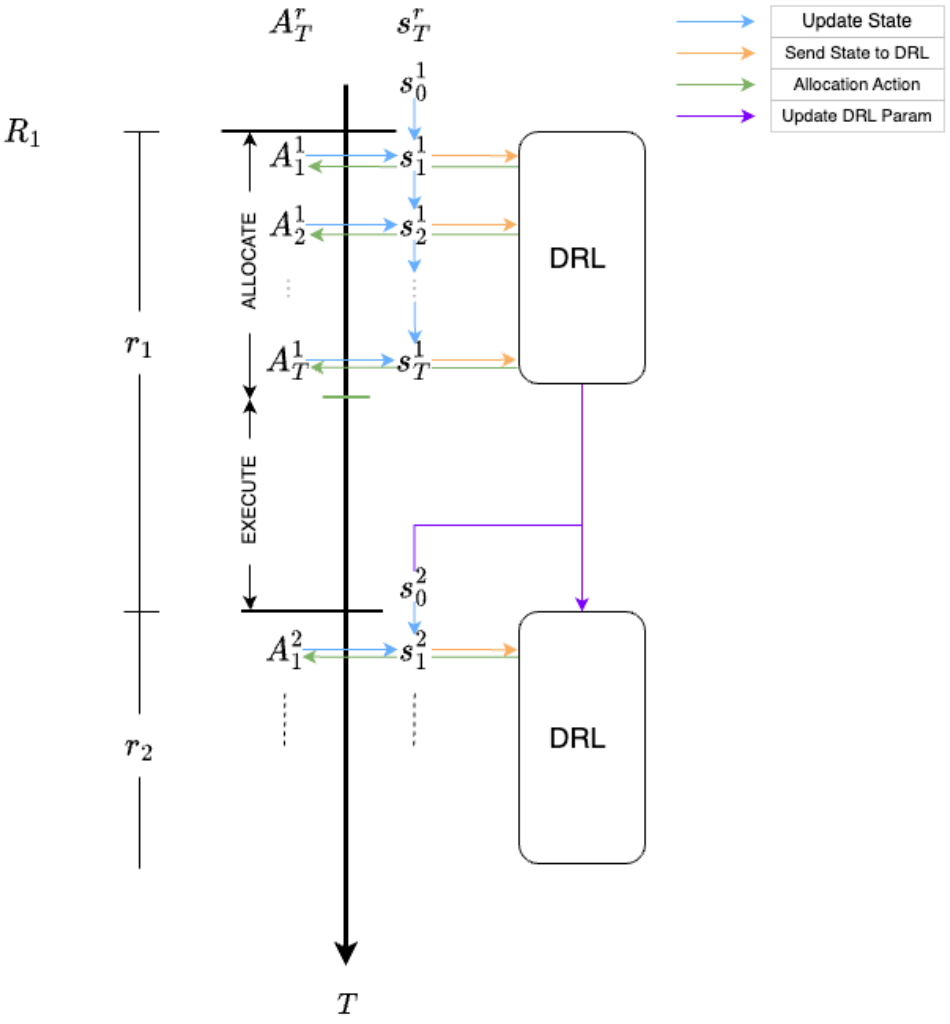


Figure 3.1: Flowchart for Mini Round Operations

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

Chapter 4 Performance Evaluation

4.1 Multi-Round Simulation General Performance Analysis

In this section, we run multi round simulation to estimate the performance of the system on the proposed algorithm. Setting random allocation as the baseline to compare, the simulation ran through multiple rounds of training under different intensity of task arrival rate.

4.1.1 Environment System Setup

Consider a fixed amount number of mobile users and MEC servers randomly scattered in a urban district as a circle area with 2km radius. The district has $N_B = 4$ Base Stations and $K = 4$ MEC Servers. Each MEC Servers is distributed separately base stations, each base station has access and same priority to each MEC severs. The total number of mobile users in the environment $I = 40$. For each users, the tasks generated by them follows the Poisson distribution $Poi(\lambda)$ with the arrival rate equals λ. The arrival rate $\lambda \in [8, 10, 12, 14]$ ranging from less traffic to tense traffic.

doi:10.6342/NTU202302089

4.1.1.1 Data Size Type

For the data size type, there are three data size type: Small, Medium and Large, each is also denoted as Type0, Type1 and Type2 respectively. For each data size type, the task is uniformly sampled over a range respected to its data size type. The size between difference data size type can be two to ten times larger. In such environment, it is important to cleverly allocate tasks so that tasks can be fairly executed as much as possible without sacrificing the overall performance. Each task have difference probability to be each task size type. Here, from small to large, the probability each tasks is which task size type are $[0.5, 0.3, 0.2]$. This way of definition such that the system has small task type the most and large task size type the least, which is similar to real-world interaction environment. The detail setting of parameter can see Table 4.1.

Table 4.1: PARAMETER SETTING

Parameters	Values
Number of BSs (N_B)	3
Number of MECs (K)	4
Number of Users (I)	40
Task arrival rate λ per User	$[8, 10, 12, 14]$
Task Type	$[0, 1, 2]$
Small Data size (Type0)	3-5 kbits
Medium Date size (Type1)	6-10 kbits
Large Data size (Type2)	15-50 kbits
Probability of each data type	[0.5, 0.3, 0.2]
The Computation Intensity	1000 (cycles/bit)
Task deadline	0.3-1 sec.
Public Round	1 sec.
Mini Round	0.05 sec.
MEC CPU frequency of 4 MEC	[1, 1, 1, 2] GHz

doi:10.6342/NTU202302089

4.1.1.2 MEC CPU Frequency

At the MEC server side, four MEC servers have CPU frequency = $0.5, 0.5, 1, 2$ respectively. Instead of setting all MEC server the same CPU frequency as previous paper set ([8]), by setting different CPU frequency increase the robustness of the model to adapt to a more complex scenario that in fact closer to the real world situation. Usually MEC servers scattered in urban area are maintained separately, when a new generation MEC server comes out, the company that organize MEC servers often change servers one at a time. In such way, MEC servers are likely to have very different computational power.

4.1.2 Completion Rate over different Arrival Rate

4.1.2.1 Hard Completion Rate and Soft Completion Rate

Here we compare two kind of completion rate performance over different task arrival rate λ: **Hard Completion Rate** and **Soft Completion Rate**. The soft completion rate count the ratio of tasks that have been successfully executed within a public round regardless of their respective delay tolerance. Hard completion rate, on the other hand, count the ratio of tasks not only have finished their execution but also finished within delay tolerance. As the tasks' delay tolerance fall between $[0.3, 1]$, it is challenging to have most tasks finish in time before delay tolerance, hence we also evaluate the performance on soft completion rate to get a broader view. The hard completion rate is demonstrated at Figure 4.1. As the traffic getting heavier, the proposed DRL allocation method manage to have a steady and better performance over random allocation method.

doi:10.6342/NTU202302089

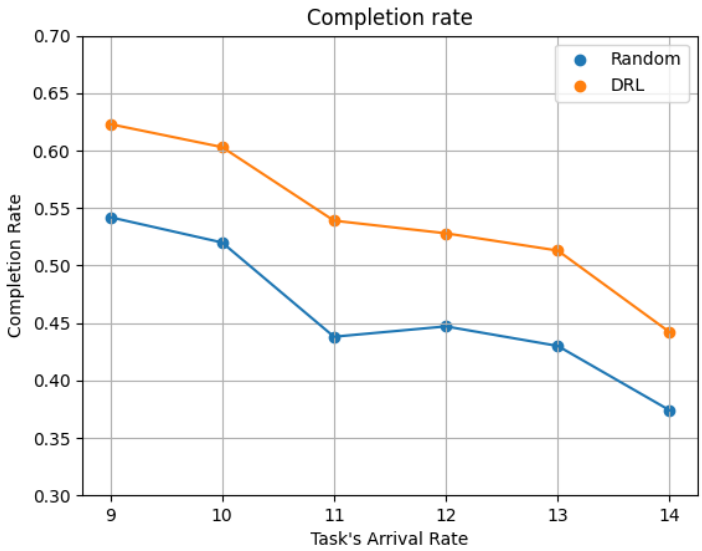


Figure 4.1: Hard Completion Rate over Different Task Arrival Rate

If we look at the soft completion rate that release the delay tolerance constrain, we can observe that both DRL and Random method have promising results. Compared to the random allocation, DRL allocation managed to have a better performance no matter in heavy traffic or light traffic task arrival rate.

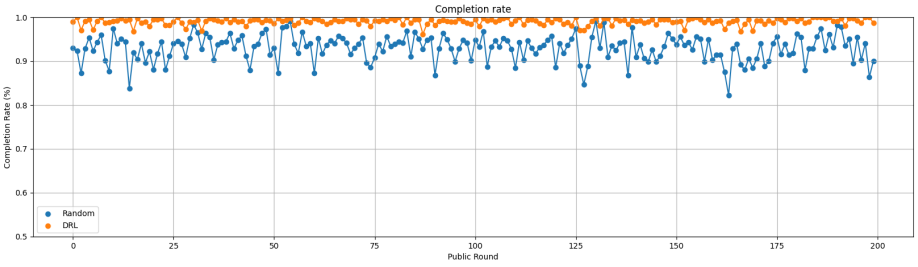


Figure 4.2: Soft Completion Rate ($\lambda = 9$)

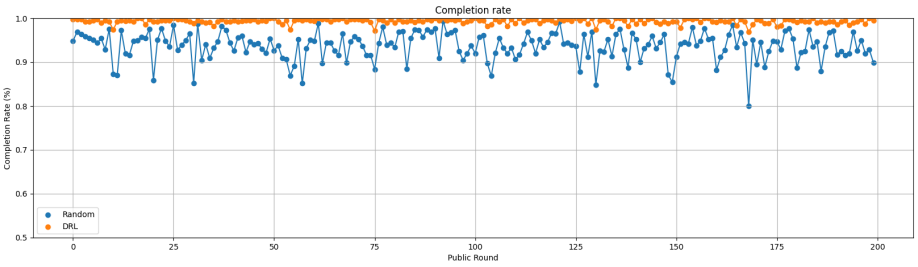


Figure 4.3: Soft Completion Rate ($\lambda = 10$)

doi:10.6342/NTU202302089

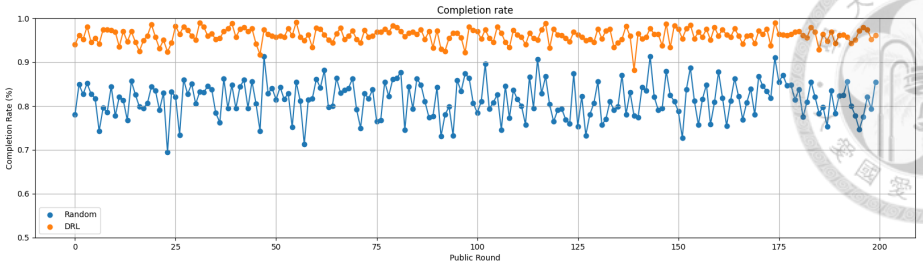


Figure 4.4: Soft Completion Rate ($\lambda = 11$)

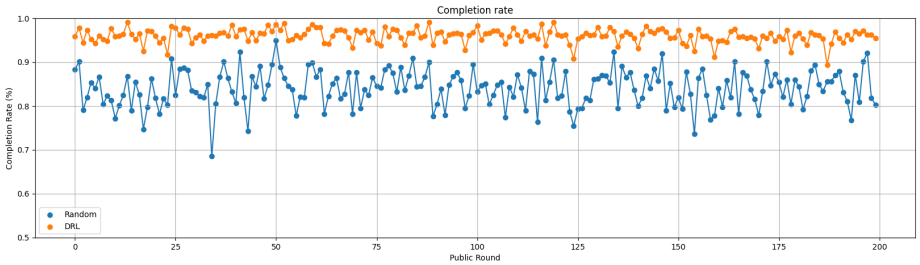


Figure 4.5: Soft Completion Rate ($\lambda = 12$)

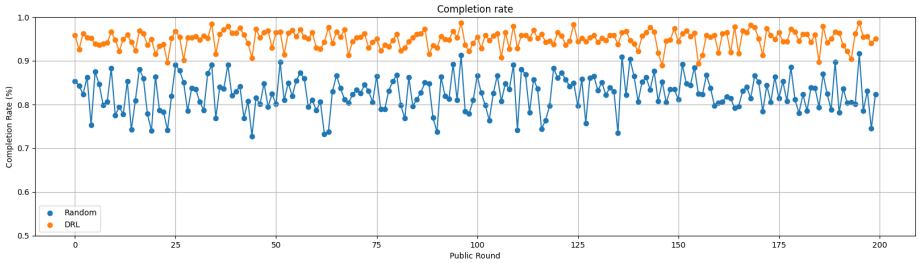


Figure 4.6: Soft Completion Rate ($\lambda = 13$)

4.1.3 Utilization per Mini Round Performance

In this subsection, we observe and analysis the performance of utilization over all MEC servers every mini round. In the real world scenario, allocation decision is a chain of multiple decisions, and each allocation decision will affect its immediate successor and even multiple descendent from it. The resources allocated and the choice it make to assign which task to which MEC server will have huge impact on when and how much each tasks be executed. Ideally, if a allocation scheme is well designed and well optimized, we should be able to observe high utilization average over mini rounds. Also, a better allocation will

doi:10.6342/NTU202302089

be able to see the utilization early drop at the ending of the mini rounds. This can be observed in Figure 4.9. The reason is that, if the amount of computation requirement is affordable within a public round, a optimized allocation decisions will maximize the percentage of utilization from the early mini round. In such way, tasks will be executed earlier and the idle computation resource will appear at the end of the mini rounds. In contrast, a less utilized allocation scheme will have a less average utilization and late to non utilization drop at the end of mini rounds.

We compare the DRL method and random method over 200 public rounds to see the average utilization over each mini rounds. When the traffic is heavy ($\lambda = 13$), both method hold a steady utilization over each mini round. As we can see DRL has the better average utilization. When the traffic is less heavy ($\lambda = 9$) where both method has over 90 percentage of soft completion rate. DRL utilization drop at mini round 18 from its steady average utilization, whereas random method holds similar utilization throughout all mini rounds. In light traffic ($\lambda = 7$) situation, both methods manage to finish most tasks before the end of mini round, indicating that when traffic is not too heavy, both method can handle finishing the tasks well. However, when the heavy traffic comes, DRL managed to have better performance over all mini rounds.

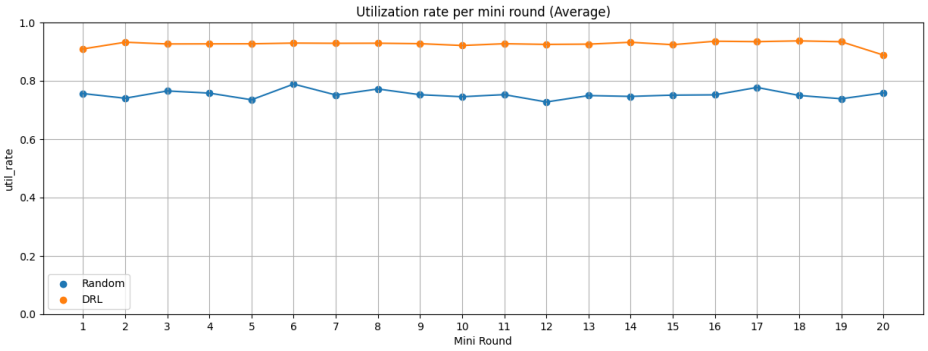


Figure 4.7: Utilization ($\lambda = 13$)

doi:10.6342/NTU202302089

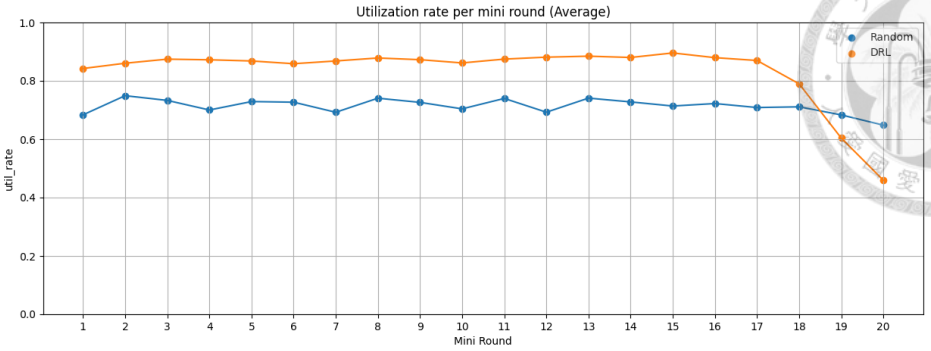


Figure 4.8: Utilization ($\lambda = 9$)

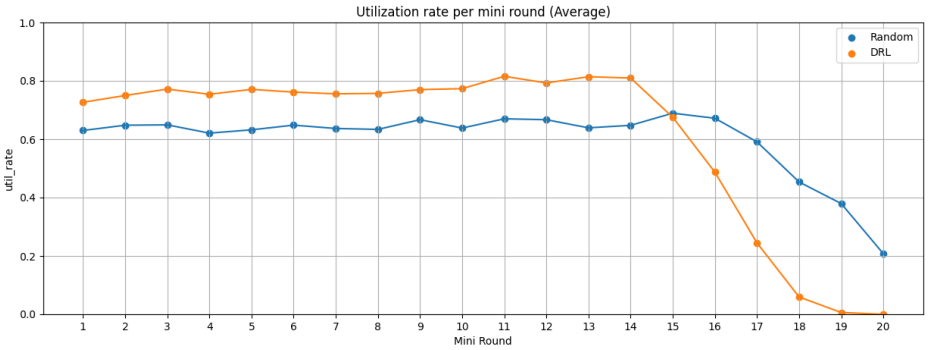


Figure 4.9: Utilization ($\lambda = 7$)

4.1.4 Fairness Analysis

Aside from finishing the tasks as fast as possible and as much as possible, it is important while easily neglected to make sure the fairness among tasks are considered and maintained. Throughout the survey papers applying DRL-based method to solve MEC server resource allocation problem, Their objectives focus either on energy consumption [9] or a mix average of energy and time cost [8]. In this research, aside from optimizing the number of tasks being successfully executed, we also evaluate the performance of fairness between tasks. Here we apply the Maximal Task Fairness Ratio and the Average Task Fairness Ratio defined in [12].

doi:10.6342/NTU202302089

1. Maximal Task Fariness Ratio

For each task A_i, the required deadline is defined as τ_i, and the actual total time consumption from being requested from the user to being executed is denoted as t_k, then define an auxiliary variable $G_i = \frac{t_i}{\tau_i}$. In another words, this is the ratio of actual delay to maximum delay tolerance of the task. In the following article, this ratio is called the **task fairness ratio** G_i of task A_i. For each task, getting minimum task fairness ratio imply faster execution, which implying better satisfaction. On the contrary, if the task fairness ratio goes larger than 1, it means the task isn't executed within deadline. Consider $\mathcal{A}$ as the task set including all tasks in the system per public round, then define:

$$G_{max} = \max_{A_i \in \mathcal{A}} G_i \tag{4.1}$$

where G_{max} denotes the maximum task fairness ratio G_i among all tasks in a public round. To determine a method to be more fair than another, smaller G_{max} implies a smaller upper bound of task fairness ratio G_i, hence a more fair method.

Table 4.2: G_{max} among different task arrival rate

	DRL	Random
$\lambda = 9$	**3.443**	3.819
$\lambda = 10$	**3.288**	3.712
$\lambda = 11$	4.011	**3.880**
$\lambda = 12$	3.995	**3.898**
$\lambda = 13$	4.088	**3.926**

As we can see from Table4.2, DRL have much better when arrival rate $\lambda = 9, 10$, meaning a better allocation under light traffic. In heavy traffic, however, random method got slightly better G_{max} but the difference is small.

2. Average Task Fairness Ratio

doi:10.6342/NTU202302089

Another point of view is to look at the average of all task fairness ration G_i, which is denoted as G_{avg}.

$$G_{avg} = \frac{1}{n_{task}} \sum_{A_i \in \mathcal{A}} G_i \tag{4.2}$$

In this way, the average task fairness ratio represent a more general fairness among all tasks, as G_{max} simply demonstrate the worst task's task fairness ratio, the remaining tasks' task fairness ratio distribution are unknown.

Table 4.3: G_{avg} among different task arrival rate

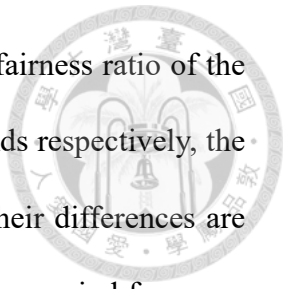
	DRL	Random
$\lambda = 9$	**0.985**	1.079
$\lambda = 10$	**0.944**	1.117
$\lambda = 11$	**1.059**	1.218
$\lambda = 12$	**1.081**	1.222
$\lambda = 13$	**1.131**	1.244

Looking at the result of G_{avg} between DRL and Random over different arrival rate λ, DRL all yield a better performance on G_{avg}, which we can imply that DRL provide a better allocation decision through the point of view of fairness. Not only the DRL method have better performance on G_{max}, but also maintain a fair performance on individual task's worst fairness ratio G_{max}.

4.1.5 Model Training

For each simulation, the model run through 200 public rounds and see the performance throughout the whole process. Since the model try to make allocation decision to MEC servers, the initial model allocation decision is similar as randomly allocate each tasks. Noted that the performance of a random allocation will not be obviously bad comparing to other scenario when applying RL to gaming. Therefore the model does not need

doi:10.6342/NTU202302089



many public rounds to warm up. When we look at the average task fairness ratio of the first 50 public rounds, last 100 public rounds, and last 50 public rounds respectively, the average fairness ratio when $\lambda = 16$ is 1.236, 1.231, 1.221. Since their differences are within 1 %, such results show that the model does not need a warm up period for many public rounds. Hence in the simulation we directly illustrate the performance throughout all public rounds without a training period.

4.2 Performance Analysis of Multi-Round Simulation over Different Data Size

In this section we break down the view to look at performance over different sets of size types. Inside the simulation, three types of data size tasks are randomly generated: **Small(Type0), Medium(Type1), Large(Type2)**. The data size for each type is uniformly sampled within its respected range, and the average data size different between different tasks can have up to 5-10 times difference. In real world scenario, various kind of tasks requests come and go to MEC servers. These tasks can be small computation requests that do not need to be strictly finished within their deadline, or can also be huge computation offloading requests that require fast completion like AI model training in reference. As the world is getting more and more less tolerant to delay, it is challenging to satisfy all types of tasks computing requests. On the other hand, to nicely evaluate pros and cons between various kind of allocation methods are challenging as well, because different point of views can have very different conclusions. For example, if an allocation scheme aims to maximize the utilization among all MEC servers, chances are the scheme can favor larger tasks because just by running large tasks can easily fill up the utilization.

doi:10.6342/NTU202302089

In such way, however, smaller tasks could possibly face timeout. On the other hand, if an allocation mechanism mainly focuses on minimizing total energy consumption on the local devices, large tasks might cause longer time to finish since the allocation model could possibly execute smaller tasks first for less energy consumption. Therefore, whether different size of tasks can all benefit fairly from an allocation model becomes an important yet easily neglected challenge to tackle. In this section, we compare the performance on soft completion rate, max task fairness ratio G_{max}, and average task fairness ratio G_{avg} over three types of data size.

4.2.1 Completion Rate over task size type

Table 4.4: Soft Completion rate among different task size

	DRL			Random		
	Small	Medium	Large	Small	Medium	Large
$\lambda = 8$	**0.908**	0.904	0.783	0.866	0.864	0.757
$\lambda = 10$	0.856	**0.871**	0.803	0.790	0.804	0.711
$\lambda = 12$	**0.846**	0.778	0.491	0.779	0.714	0.442
$\lambda = 14$	**0.770**	0.695	0.388	0.692	0.627	0.344

The simulation runs over 200 public rounds with the same basic parameter setup as Sec.4.1. As we can see, small tasks have a better average completion rate. This is because these small tasks can be quickly executed as long as it's there turn to execute. As the task getting larger, the average soft completion rate decrease gradually. In general, DRL manage to have a better soft completion rate over the Random method. When we compare the best completion rate among task size type, only once at $\lambda = 10$ has medium task

doi:10.6342/NTU202302089

type yield a better completion rate than small data size. Another thing we can observe is that, when the traffic getting heavy, large tasks are effected the most, where the soft completion rate dropped 0.4 percent from $\lambda = 8$ to $\lambda = 14$. This is an inherent challenge because when the system have large number of tasks of all data size types, sacrificing large task's performance can be more likely to maintain small and medium data size's overall performance. In other word, if large data size type tasks are been chosen to execute even slightly more can strongly affect performance of small and medium tasks. Through the simulation can observe that in order to have improvement on all types of tasks size, methods that have the higher utilization rate can achieve this. When $\lambda = 14$, for example, not only the DRL has improvement on large data size type's utilization rate, small and medium data size both have a slight better utilization rate per mini round.

4.2.2 Task Fairness Ratio over data size type

Aside from completion rate, in this section we take a look at the fairness index over different data size type. By looking at the maximum task fairness ratio G_{max} and the average task fairness ratio G_{avg}, we evaluate how fair among each data size type task. The simulation result is shown in Table 4.5.

Table 4.5: G_{max} among different task size

	DRL			Random		
	Small	Medium	Large	Small	Medium	Large
$\lambda = 8$	2.10	2.12	2.14	2.51	2.08	**1.31**
$\lambda = 10$	**2.42**	2.48	2.55	3.36	3.31	3.89
$\lambda = 12$	2.66	2.68	**2.38**	3.7	2.99	3.26
$\lambda = 14$	2.84	2.59	**2.52**	4.30	3.88	2.97

Compared with the random method, DRL has a better and stable G_{max}. When the

doi:10.6342/NTU202302089

traffic is light, both method made fair allocations, and the random method has the best G_{max} at large task size when $\lambda = 8$. However, when λ increases to 14, the DRL method holds on to better G_{max} in three types of task fairness ratio than random. This tendency shows that the DRL method handles well in heavy traffic situations. Also, we can observe that when $\lambda = 12, 14$, the large data size type has the minimum G_{max} comparing to small and medium data size type. To explain, since more large data size type tasks are in the system looking for computation resources, they increase the proportion of time the MEC server executing the large tasks, hence smaller tasks are effected more. As they are more likely need to wait for large tasks finish, the queue time for smaller tasks are likely increase.

Table 4.6: G_{avg} among different task size

	DRL			Random		
	Small	Medium	Large	Small	Medium	Large
$\lambda = 8$	**0.42**	0.45	0.54	0.53	0.56	0.64
$\lambda = 10$	**0.46**	0.51	0.64	0.61	0.66	0.81
$\lambda = 12$	**0.52**	0.56	0.69	0.70	0.75	0.90
$\lambda = 14$	**0.53**	0.56	0.65	0.74	0.77	0.86

At Table 4.6, the average task fairness ratios are demonstrated. From $\lambda = 8$ to $\lambda = 14$, the DRL manage to keep a stable average fairness ratio among different, and the difference between different tasks size type are not too large. In comparison, the random allocation increase faster when task arrival rate increases, for example the small tasks increase 39% of average fairness ratio in random allocation, which is larger than 26% for DRL allocation on small task.

doi:10.6342/NTU202302089

4.3 Large Data Size Difference with Weighted Reward

In the previous section experiments, we set three types of different data size type and compared the performance. To follow up, in the following section we simulate on data size type that have much larger size difference. The motivation to expand this simulation is that, in real world the task size difference can be up to 100 times, how many large tasks being computed before small tasks then greatly affect the computation time of small tasks. Since throughout previous survey papers, none of them ran simulation under complex data size range, this section made a experiment on distant data size type.

Consider three data size, where the small data size can be as small as 0.2 kbits, and the large data size can be up to 70 kbits. The maximal possible data size difference can then be up to more than 100 times difference, which is much larger than the 10 times data size difference in previous simulations. The detail of parameter setup can see Table 4.7.

Noted that the probability distribution of each data size type is 70%, 20%, and 10%. The setup is meant to emulate an environment where extremely large data size tasks exist but doesn't appear frequently. Hence, how to manage these seldom appeared large tasks and maintain the fairness among tasks is the main goal in this simulation. The large task size fall within the range of $[40, 70]$ kbits, which means that some large tasks are not going to fit in a mini round of a small MEC server that have 50 kbits maximal CPU frequency per mini round. Therefore, these large tasks would be better to offload to the large MEC server (with 100 kbits max. CPU frequency /mini round). In the following simulation result can tell that, the proposed DRL allocation method can have better allocation scheme.

doi:10.6342/NTU202302089

Table 4.7: PARAMETER SETTING (LARGE DATA DIFF)

Parameters	Values
Number of BSs	7
Number of MECs	4
Number of Users	40
Task arrival rate λ per User	$[8, 10, 12, 14, 16]$
Task Type $\mathcal{T}$	$[0, 1, 2]$
SMALL	
Small Data size (Type0)	0.2-2 kbits
Prob. of Small Data	0.7
MEDIUM	
Medium Data size (Type1)	10-30 kbits
Prob. of Medium Data	0.2
LARGE	
Large Data size (Type2)	40-70 kbits
Prob. of Small Data	0.1
The Computation Intensity	1000 (cycles/bit)
Task deadline	0.3-1 sec.
Public Round	1 sec.
Mini Round	0.05 sec.
MEC CPU frequency of 4 MEC	[1, 1, 1, 2] GHz
Max. CPU frequency of each MEC in a mini round	[50, 50, 50, 100] kbits

4.3.1 Weighed Reward Function

As the task size variation getting larger and larger, larger tasks should receive larger weight to compensate its time-consuming characteristic. Previously, the reward in DRL is recorded as the number of tasks finished, regardless of the difference of task size. Here we introduce weighted reward depending on the task's size. Denote the task size as x, the weighted reward r is determined by weighted reward function $r = R(x)$. We experiment several kind of weighted reward function and evaluate the performance when the task arrival rate $\lambda = 16$.

Size Type Weighted Reward Function

doi:10.6342/NTU202302089

Define the weighted reward function equals to the task's data size type $\mathcal{T}+1$. Hence, the large size task get weight = 3, medium weight = 2, and small weight = 1.

$$R(x) = \mathcal{T} + 1 \tag{4.3}$$

Linear Weighted Reward Function

Define the linear weighted reward function as the round down value of the task's data actual size in kbits. The task size range $[0.5, 70]$ among all task size type, hence possible weight fall between positive integer no larger than 70.

$$R(x) = \lfloor \frac{x}{10^6} \rfloor \tag{4.4}$$

Square Root Weighted Reward Function

Define the square root weighted reward function by setting the square root of the actual data size as the weight. The threshold of which data size to apply square root is tested to performs best starting at 0.

$$R(x) = \sqrt{\frac{x}{10^6}} \tag{4.5}$$

Power Weighted Reward Function

Define the power weighted reward function as the data size (in kbits) to the power of 1.1. Since using exponential weight can greatly rewards large size task completion, which might contradict the goal of fairness among all task size type. Therefore, here we set a small power to test its performance.

$$R(x) = (\frac{x}{10^6})^{1.1} \tag{4.6}$$

doi:10.6342/NTU202302089

Here we ran the performance among different weighted reward function over three different traffic arrival rate representing intensive ($\lambda = 16$), normal ($\lambda = 12$), and light ($\lambda = 8$) traffic scenario. The performance is evaluate by maximal task fairness ratio G_{max} and average task fairness ratio G_{avg}. By looking at the fairness performance enable us to separate pros and cons between different weighted reward function method.

Table 4.8: Weighted Reward Function Comparison via G_{max} ($\lambda = 16$)

	Data Size Type		Linear		Square Root		Power	
$\lambda = 16$	DRL	Random	DRL	Random	DRL	Random	DRL	Random
Small	**4.99**	7.20	**4.92**	4.52	**5.00**	5.85	**4.97**	6.20
Medium	**5.58**	5.82	**5.29**	6.15	5.37	**5.14**	5.25	**4.09**
Large	**5.36**	5.85	**4.84**	5.19	4.97	**3.46**	4.98	**3.37**

Table 4.9: Weighted Reward Function Comparison via G_{avg} ($\lambda = 16$)

	Data Size Type		Linear		Square Root		Power	
$\lambda = 16$	DRL	Random	DRL	Random	DRL	Random	DRL	Random
Small	**1.03**	1.13	**1.00**	1.10	**1.03**	1.12	**1.08**	1.19
Medium	**1.46**	1.49	**1.36**	1.42	**1.47**	1.51	1.44	**1.43**
Large	1.73	**1.68**	1.75	**1.71**	1.74	**1.69**	1.74	**1.70**

Table 4.10: Weighted Reward Function Comparison via G_{max} ($\lambda = 12$)

	Data Size Type		Linear		Square Root		Power	
$\lambda = 12$	DRL	Random	DRL	Random	DRL	Random	DRL	Random
Small	**4.12**	4.17	**4.10**	4.66	3.75	**3.52**	**3.84**	4.50
Medium	**4.14**	6.21	**4.68**	4.79	**4.05**	4.94	**3.86**	4.14
Large	**4.30**	6.25	**4.61**	6.12	**3.79**	4.51	**4.20**	4.46

doi:10.6342/NTU202302089

Table 4.11: Weighted Reward Function Comparison via G_{avg} ($\lambda = 12$)

	Data Size Type		Linear		Square Root		Power	
$\lambda = 12$	DRL	Random	DRL	Random	DRL	Random	DRL	Random
Small	**0.88**	0.95	**0.91**	0.98	**0.83**	0.92	**0.84**	0.92
Medium	**1.16**	1.30	**1.25**	1.36	**1.17**	1.34	**1.04**	1.17
Large	**1.70**	1.72	**1.77**	1.78	**1.58**	1.65	**1.63**	1.67

Table 4.12: Weighted Reward Function Comparison via G_{max} ($\lambda = 8$)

	Data Size Type		Linear		Square Root		Power	
$\lambda = 8$	DRL	Random	DRL	Random	DRL	Random	DRL	Random
Small	**2.84**	3.16	**3.16**	3.54	2.77	**2.32**	**2.86**	3.40
Medium	**2.88**	3.25	**2.86**	3.09	**2.70**	2.75	**2.62**	3.34
Large	3.33	**3.04**	3.58	**3.30**	**3.3**	3.73	3.26	**3.23**

Table 4.13: Weighted Reward Function Comparison via G_{avg} ($\lambda = 8$)

	Data Size Type		Linear		Square Root		Power	
$\lambda = 8$	DRL	Random	DRL	Random	DRL	Random	DRL	Random
Small	**0.67**	0.71	**0.74**	0.80	**0.68**	0.72	**0.69**	0.72
Medium	**0.91**	1.02	**0.86**	0.99	**0.82**	0.92	**0.82**	0.91
Large	**1.46**	1.54	**1.48**	1.54	**1.49**	1.58	**1.42**	1.50

The Table 4.8 and Table 4.9 shows the simulation integrating different weighted reward functions and evaluate on task fairness ratio G_{max} & G_{avg} under intensive traffic. For each kind of weighted reward function, the task fairness compares over each task size type respectively. The highlights shows the better (smaller) values. When looking at the

doi:10.6342/NTU202302089

maximum task fairness ratio G_{max}, Applying data size type and linear generates better performance in intensive traffic ($\lambda = 16$), since they both manage to have better task fairness ratio over all three task size type. This can also been seen in G_{avg}, where both method have better average task fairness ratio over small and medium tasks. When the traffic is normal ($\lambda = 12$), through Table 4.11 and Table 4.10, all weighted reward function results in good performance as they performs better performance in G_{avg} than the random method. This states the overage fairness is better in DRL. On the other hand, all weighted reward function results in good the maximal fairness ratio G_{max} comparing to the random method as well. Therefore, when the traffic is normal, choosing either way of weighted reward function make no huge difference in fairness performance. In light traffic ($\lambda = 8$) scenario, all weighted methods performs well in general in G_{avg}. This can been seen in Table 4.13. In maximal fairness ratio G_{max}, however, have different performance. The data size type, square root and power methods all have better performance in small medium, but not in large. In summary, when the traffic is normal and light, the choosing of weighted reward function does not make huge difference in fairness ratio performance among different tasks size type. However, when the traffic is intense, size type weighted reward function and linear weighted reward function performs better. In the following simulation, we will apply the linear weighted reward function to calculate the weighted reward.

4.3.2 Fairness Index Evaluation

Here the simulation ran through 5 different task arrival rate ($\lambda = 8, 10, 12, 14, 16$) and compare the performance between the proposed DRL method and the random method size by size. For the small tasks, the DRL method outperform in 3 out of 5 testing. The medium

doi:10.6342/NTU202302089

Table 4.14: G_{max} among different task size type

	Small		Medium		Large	
	DRL	Random	DRL	Random	DRL	Random
$\lambda = 8$	**2.93**	3.85	**2.87**	3.38	3.25	**2.78**
$\lambda = 10$	3.67	**3.16**	3.68	**3.59**	3.75	**3.38**
$\lambda = 12$	**3.88**	4.99	**4.08**	4.37	3.98	**3.47**
$\lambda = 14$	**4.76**	6.50	5.34	**4.67**	5.05	**3.40**
$\lambda = 16$	4.92	**4.52**	**5.29**	6.15	**4.84**	5.19

tasks have the DRL method better performance in 3/5 cases. In the large tasks, however, the random method managed to have a better performance in 4/5 arrival rate scenario. We can interpret the DRL method focus more on finishing small to medium, whereas the random method tends to be more uncertain on performance over different data size type.

Table 4.15: G_{avg} among different task size type

	Small		Medium		Large	
	DRL	Random	DRL	Random	DRL	Random
$\lambda = 8$	**0.79**	0.75	**0.88**	0.97	**1.45**	1.54
$\lambda = 10$	**0.82**	0.87	**1.05**	1.17	**1.67**	1.73
$\lambda = 12$	**0.87**	0.94	**1.14**	1.27	**1.60**	1.64
$\lambda = 14$	**1.01**	1.10	**1.42**	1.45	1.82	**1.79**
$\lambda = 16$	**1.00**	1.10	**1.36**	1.42	1.75	**1.71**

The average task fairness ratio G_{avg} give us a more general point of view over all tasks. As we can see in Table 4.15, the DRL method outperforms the random methods in both the small tasks and medium tasks, where for each λ, the G_{avg} in DRL in smaller. The DRL method also have better G_{avg} when the traffic is relatively smaller. Although the DRL have larger G_{avg} in larger λ, the difference is close. Therefore, by combining performance over G_{avg} and G_{max}, we conclude the fact that the DRL method manage to handle extreme data size type difference better than the random method. With this robustness, the DRL method has great potential to successfully handle real world complex

doi:10.6342/NTU202302089

task size flow scenario.

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

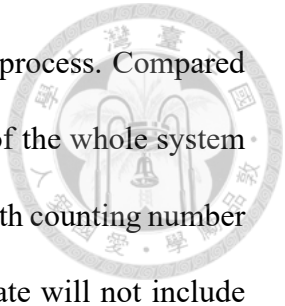
Chapter 5 Conclusion and Future Work

5.1 Conclusion

In this thesis, we have proposed a Deep reinforcement learning based resource allocation algorithm on MEC servers with fairness awareness. In order to design a resource allocation method optimizing desiring objective like time and energy, researchers implied various kind of method to try to tackle this problem. Among those methods, multiple Deep reinforcement learning based allocation algorithms have been proposed to optimize time consumption or energy consumption. However, aside from maximize time consumption among all tasks, it is also important to take the fairness between tasks into consideration. Since high performance overall might neglect some tasks being sacrificed to achieve the result. Therefore, this thesis designed a DRL-based resource allocation algorithm that value task fairness through the maximal task fairness ratio G_{max} and average task fairness ratio G_{avg}. Moreover, we made simulations on scenario that tasks size comes from huge size difference, and ran the model to compare with the random method and shown a promising result.

With the simplified definition on the state, the proposed approach effectively reduces

doi:10.6342/NTU202302089



the state space size and therefore reduce the complexity of the decision process. Compared to previous research's definition [8], where the state is the sum cost of the whole system and the total available computational capacity, we describe the state with counting number of tasks of each type in the system. In this way of definition, the state will not include attributes that are continuous but integer. Hence, the possible number of states became countable. Throughout the simulations have shown the model managed to perform decent allocation scheme under this state definition, which also maintains the fairness between tasks. Also, instead of allocating a batch of tasks at the same time, the thesis proposed a smaller model to allocate one task at a time. In this way the DRL model have the action space with the size of the number of MEC serves in the system, which is much less than allocating multiple tasks at a time.

5.2 Future Work

For the future work, the DRL network have the flexibility to expand the state space, we can include more aspects into state so that the model can observe the environment in a more thorough point of view. For example, in this thesis we mainly focus on maximizing the number of tasks finished, and minimizing time consumption and task fairness. We can also expand the objective function to consider energy consumption among tasks. As the energy consumption takes a huge part of the MEC server data centers' operating cost, which is an crucial part its company will value. Also, we can expand the action decision scenario, and add additional index that need to match, for example the need of GPU resource. These features can be an important and useful topic for the research and for the application to real-world system. To do so, we can modify the definition of the state and include the information that whether each MEC server has a GPU or not. Apart

doi:10.6342/NTU202302089

from the MEC servers computation resources, cloud computation resources can also put into the environment as an offloading destination option. Different from MEC servers, clouds have massive computation power but in a distant location from the mobile user. Therefore, how to strategically leverage cloud resources without long queuing time and transmission time becomes a crucial challenge.

Another possible extension is to apply the model into an multi-core MEC servers system. Since in the thesis we assume each MEC server act as a single core MEC server, we can deploy a mechanism with multiple flow in parallel. For each flows of tasks, every MEC core runs its public rounds and mini rounds respectively and the staring time stamp of each MEC core are split. With this design of splitting stating time among different MEC cores, we can simply assign offloaded tasks to the closest next start of the public round's MEC core. In the one-core design in the thesis, offloaded tasks will have to wait until the next start of the public round to be possibly allocated. In other words, those tasks that arrive soon after a start of a pubic round have to wait a large period of a public round duration before allocation, causing longer waiting time in the system. By integrating into the multi-core MEC system, offloaded tasks will only need to wait until the next starting public round from one of the MEC core, hence those tasks will wait less amount of time before allocation. Also, since each MEC core execute in parallel, multi-core MEC system does not increase the computational complexity, meaning better chance to scale up the size of the employed network.

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

References

[1] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust, "Mobile-edge computing architecture: The role of mec in the internet of things," *IEEE Consumer Electronics Magazine*, pp. 84–91, Oct 2016.

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing —a key technology towards 5g," *ETSI white paper*, pp. 1–16, Nov 2015.

[3] Q. V. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, and et al., "A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art." *IEEE access*, pp. 116 974–117 017, Aug 2020.

[4] N. C. Luong, D. T. Hoang, Gong, S., Niyato, D., Wang, P., Y. C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey." *IEEE Communications Surveys and Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[5] V. Mnih and K. et al., "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602*, 2013.

[6] W. Xiong, T. Hoang, and W. Y. Wang, "Deeppath: A reinforcement learning method for knowledge graph reasoning," *arXiv preprint arXiv:1707.06690*, 2017.

doi:10.6342/NTU202302089

[7] K. Li, T. Zhang, and R. Wang, "Deep reinforcement learning for multiobjective optimization," *IEEE transactions on cybernetics*, vol. 51, no. 6, pp. 3103–3114, 2020.

[8] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for mec," *IEEE Wireless Communications and Networking Conference(WCNC)*, pp. 1–6, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8377343

[9] H. Zhou, K. Jiang, X. Liu, X. Li, and V. C. Leung, "Deep reinforcement learning for energy-efficient computation offloading in mobile-edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 2, pp. 1517–1530, January 2021.

[10] K. Cheng, Y. Teng, W. Sun, A. Liu, and X. Wang, "Energy-efficient joint offloading and wireless resource allocation strategy in multi-mec server systems," *Proc. IEEE ICC*, pp. 1–6, May 2018.

[11] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 11, pp. 2581–2593, November 2020.

[12] J. Zhou and X. Zhang, "Fairness-aware task offloading and resource allocation in cooperative mobile-edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 5, pp. 3812 – 3824, March 2022.

[13] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang, "Traffic engineering in software-defined networking: Measurement and management," *IEEE access*, vol. 4, pp. 3246–3256, 2016.

doi:10.6342/NTU202302089

[14] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *2016 IEEE international symposium on information theory (ISIT)*. IEEE, 2016, pp. 1451–1455.

doi:10.6342/NTU202302089

doi:10.6342/NTU202302089

Denotation

BS_i	The i-th Base Station
N_B	Number of Base Station
M_k	The k-th MEC Server
K	Number of MEC Servers
$Cloud$	Cloud Server
U_i	User i
I	Number of Users
A_{ij}	The j-th Task from User i
N	Number of Tasks in a round
D_{ij}	Computational Data Size (bit) of task A_{ij}

doi:10.6342/NTU202302089

X_{ij}	Computation Intensity (CPU Cycle/bit) of task A_{ij}
τ_{ij}	Requested Completion Deadline of task A_{ij}
g_{ij}	GPU requirement indicator of task A_{ij}
C_{U_i}	Class of User i
I	Individual User
E	Enterprise User
λ_i	Arrival Rate of User i
λ_E	Arrival Rate of Enterprise User
λ_I	Arrival Rate of Individual User
N_{U_i}	Number of Tasks generated by User i
T_{ij}^l	Estimated local computation of task A_{ij}
η_i	Offload threshold of user i (sec.)
N_i^l	Number of Local-Mode Tasks generated by User U_i
N_i^r	Number of Remote-Mode Tasks generated by User U_i
M_i^k	Number of Tasks processed at the k-th MEC Server required by User U_i

doi:10.6342/NTU202302089

C_i	Number of Tasks processed at the Cloud required by User U_i
$A^l_{ij'}$	j'-th Local-Mode Task generated by User U_i
$A^r_{ij'}$	j'-th Remote-Mode Task generated by User U_i
N_{M_k}	Number of Tasks in the k-th MEC Server
N_C	Number of Tasks in the Cloud

doi:10.6342/NTU202302089