Decentralized Scheduling for Concurrent Tasks in Mobile Edge Computing via Deep Reinforcement Learning

Ye Fan, Jidong Ge, *Member, IEEE*, Sheng Zhang, *Member, IEEE*, Jie Wu, *Fellow, IEEE*, and Bin Luo

Abstract—Mobile Edge Computing (MEC) is a promising solution to enhance the computing capability of resource-limited networks. A fundamental problem in MEC is efficiently offloading tasks from user devices to edge servers. However, there still exists a gap to deploy in real-world environments: 1) traditional centralized approaches needs complete information of edge network, ignoring the communication costs generated by synchronization, 2) previous works do not consider concurrent computation on edge servers, which may cause dynamic changes in the environment, and 3) the scheduling algorithm should deliver individualized decisions for different users independently and with high efficiency To solve this mismatch, we studied a multi-user task offloading problem where user devices make offloading decisions independently. We consider the concurrent execution of tasks and formulate a non-divisible and delay-aware task offloading problem to jointly minimize the dropped task ratio and long-term latency. We propose a decentralized task scheduling algorithm based on DRL that makes offloading decisions without knowing the information of other user devices. We employ Double-DQN, Dueling-DQN, Prioritized Replay Memory, and Recurrent Neural Network (RNN) techniques to improve the algorithm's performance. The results of simulation experiments show that our method can significantly reduce the long-term latency and dropped task ratio compared to the baseline algorithms.

Index Terms—Mobile edge computing, task offloading, resource allocation, deep reinforcement learning, deep q-learning.

1 Introduction

With the rapid growth of computing and communication technologies, especially mobile network services, many computation-intensive and data-intensive technologies have emerged, such as augmented reality, virtual reality, Internet of Things and Internet of Vehicles, etc. This has led to explosive growth in the amount of data generated at the user end. The traditional cloud computing architecture can-not meet the needs of low latency, high bandwidth, and localized processing required by massive mobile devices. Mobile edge computing (MEC), also known as fog computing, is a new mobile service architecture to improve application performance [1]. MEC sinks computing, storage, and processing functions from cloud servers to edge servers, providing users with proximity computing and processing capabilities, reducing network latency, improving user experience, effectively preventing network congestion and latency problems.

Edge computing improves the quality of services by placing edge servers close to users but still face several challenges. A significant problem is properly assigning server resources to tasks to reduce computation costs and maximize the edge network's long-term efficiency. To address this issue, the first question is whether or not the computing tasks generated by the user devices should be offloaded. The second question is which edge node the user device should offload if it chooses to do so. Users can obtain a faster response time by offloading work to edge nodes because the compute capabilities of edge nodes are higher than those of user devices. Because heterogeneous devices, dynamic networks, and geographic differences affect the processing speed of offloaded tasks, it is difficult for the scheduling algorithm to consider all factors. The user's tasks can be split into directed acyclic graphs (DAGs) that can be run on different edge nodes. In this paper, we focus on the problem of indivisible task offloading. This is because the focus of this paper is not to make the scheduler obtain optimal efficiency but to achieve better results than centralized scheduling using a decentralized scheduler.

Some previous works [2]–[5] have used centralized approaches to solve the offloading problem. These algorithms can generate optimal or near-optimal solutions for offloading decision-making, but they require complete information about the edge network. Algorithms in this centralized manner require synchronization of all user device information and produce a large communication overhead. This centralized manner is not suitable for the natural environment where each user device is independently executed.

In contrast to these centralized methods, we focus on decentralized task offloading in this work. In a decentralized algorithm, each user device makes offloading decisions without needing information from other devices. A decentralized algorithm is more challenging to design but

- *Y. Fan, J. Ge and B. Luo are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210008, China, and also with the Software Institute, Nanjing University, Nanjing 210008, China.*
  E-mail: mg21320002@smail.nju.edu.cn, {gjd, luobin}@nju.edu.cn.
- *S. Zhang is with the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210008, China.*
  E-mail: sheng@nju.edu.cn.
- *J. Wu is with the Center for Networked Computing, Temple University, Philadelphia, PA 19122, USA.*
  E-mail: jiewu@temple.edu.

has several advantages over centralized algorithms. First, it avoids the communication overhead imposed by synchronizing the information of all devices. In edge computing systems, the communication overhead is typically larger than the computation overhead. Neglecting the communication overhead required to synchronize states is not practical for real systems. Second, the size of the edge network usually varies over time. A decentralized approach can decouple the decision-making from the model training, thereby improving scalability. Third, there are many heterogeneous devices in the edge network, which have different processing performance and tasks, thus having different needs for offloading policies. Decentralized algorithms can learn offloading policies for each device independently, thus adapting to heterogeneous environments.

In this work, we employ DRL to make offloading decisions. This is because previous studies [6]–[10] have shown that DRL algorithms can quickly adapt to complex systems and efficiently output offloading decisions with a low long-term cost. It is not practical to directly apply deep reinforcement learning algorithms to the task offloading decision problem. DRL algorithms require a large amount of interaction with the environment and vast amounts of trial and error to learn. DRL algorithms often require millions of training episodes since DRL algorithms need to learn from random explorations. Second, previous works [6], [7], [10] only consider serial executions of all tasks. However, services on edge servers are often deployed in containers and different services are encapsulated into containers running concurrently on the server [11]. In this way, the tasks offloaded later and the ones offloaded earlier can be executed concurrently, and the schedule decisions affect the state of the system and the execution of the already offloaded tasks.

In this work, we consider a dynamic, concurrently executing environment and a long-term optimization problem. We suppose that all tasks are executed concurrently on the server and that the network, geographic location, and execution speed change randomly in the environment. Tasks are indivisible and have a deadline; if the deadline is exceeded, then the task fails to be executed. We propose a **D**ecentralized Task **O**ffloading **S**cheduling **A**lgorithm called **DOSA** for edge computing tasks based on deep reinforcement learning. It can make offloading decisions independently for each user device by observing the state of the system, deciding which edge node to offload to, and quickly adapting to different environments. The neural network model in our DRL algorithm is also trained by interacting with multiple environments in order to improve the model's adaptability. In our algorithm, task execution and transmission can fill several time slots, and we fine-tune the DQN (Deep Q-learning) algorithm [12] asynchronously update reward to adapt to this latent reward scenario. Our primary contributions are summarized as follows:

- *Task offloading problem in MEC systems*: We formulate a task offloading problem for non-divisible and delay-aware tasks. Tasks are performed concurrently at edge nodes and transmitted through wireless network queues. Our technique aims to minimize the dropped task ratio and long-term latency of offloaded tasks jointly.
- *DRL-based Task offloading Algorithm*: We developed a decentralized algorithm based on deep reinforcement learning to make offloading decisions without knowing the complete information of the edge network. In order to decrease the transmission cost and the computing cost of user devices, we partition the neural network into two parts: the intermediate representation component on the server and the decision component on the user device. Therefore, only two network layers need to be deployed on each user device. We asynchronously update reward to accelerate algorithm convergence. We employ Dueling DQN, Double DQN, and Prioritized Replay Memory techniques to improve the model training speed and RNN to encode context information to improve the adaptivity of the model.
- *Performance Evaluation*: We performed simulation experiments to evaluate the algorithm. Compared with the baseline algorithms, our algorithm can rapidly learn better decision policies and significantly reduce the system's long-term latency and dropped task ratio.

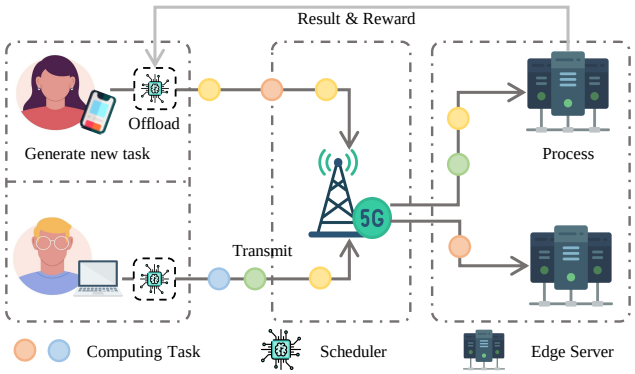


Fig. 1. System illustration of multiple edge nodes, base stations and user devices edge environment.

The remaining chapters of this paper are organized as follows. We motivates our work in Section 2. In Section 3, we describe the model of the system and formulate the task offloading problem. In Section 4, we describe the DRL network architecture and present the decentralized offloading algorithm. We present the results of the experimental evaluation in Section 5. We present related work in Section 6 and give the conclusion of the paper in Section 7.

2 Motivation

2.1 Cost of Centralized Scheduling

There are two advantages of edge computing. First, it minimizes the communication overhead between device and server. Second, it protects the user's privacy because there is no need to upload user data to the cloud server. However, these two advantages are largely surrendered when centralized work offloading is performed. Since the scheduler needs to gather information about each task before making a decision, this violates the user's privacy and also raises the cost of communication. Higher communication costs can

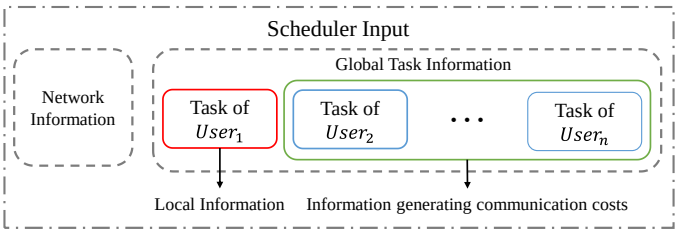


Fig. 2. The illustration of information needed by a task scheduler. The centralized scheduler needs to get global information. The local scheduler only needs the network information and local task information, and the local information does not need to be transmitted over the network, so the communication cost is significantly reduced.

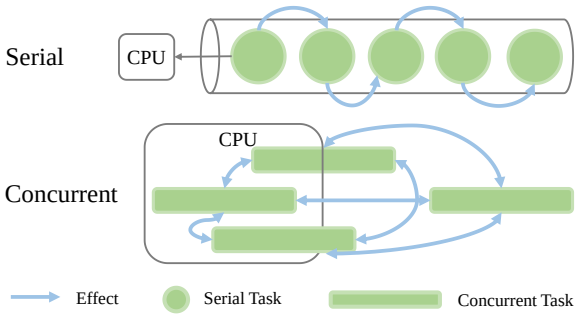


Fig. 3. The illustration of impact between serial or concurrent tasks.

significantly increase the money cost of network bandwidth required. Adopting decentralized scheduling can save thousands of dollars every month if edge network traffic reaches several terabytes per month. Network overhead for edge computing is lower than for cloud computing, but network fees are still a relatively significant expense.

A common method is allowing users to make scheduling decisions locally and decide whether tasks should be uploaded to the edge server. When the scheduler requires other users' information, the communication cost required by the scheduling algorithm grows with the square of the number of users. In contrast, when the algorithm requires just local information, the communication cost of the algorithm does not change as the number of users changes. And in this way, the task information of one user is not exposed to other users' schedulers, protecting the privacy of the user.

2.2 Concurrently Running Tasks

At a time when containerized applications are prevalent, edge servers can create independent processes to run for each user task, which can greatly reduce the response time of the task and improve the throughput of the system. However, the previous scheduling algorithms seldom considered the scenario of concurrently running tasks. After analysis, we found that the concurrent task running scenario is more complex than the traditional serial scenario.

Fig. 3 shows a diagram of serial and concurrent tasks. As you can see, in the serial scenario, the runtime of all tasks is only determined by the tasks and environment state before they start running. But concurrent tasks are also influenced by tasks that are added later. Since the uploading of tasks is often random, traditional optimization methods are not appropriate.

Edge computing networks are also highly heterogeneous and dynamic. For example, the size and number of user tasks are frequently unpredictable, and the network connectivity situations are not uniform. A network in the same server room may have high bandwidth, but the user's network may be slow. This makes it important for the algorithm to dynamically adjust to the different and changing environments.

Deep reinforcement learning can learn the implicit probabilistic relationships between parameters in complex environments, and the system grows more effective as the amount of data increases. It is worth trying the DRL algorithm for scheduling edge networks in such complicated environments.

2.3 Convergence of DRL

We designed a decentralized algorithm that requires only one user's task information and runs locally on user device, which reduces communication costs and protects the user's privacy. This also comes at a cost: deep learning methods such as the DRL algorithm usually require a large number of samples for training. And the decentralized approach makes the model receive only local user information, which makes the algorithm difficult to converge while the amount of information is greatly reduced. Because the complexity of the environment is increased by considering concurrency scenarios, the cost required for algorithm training is even higher. To solve these problems, we've made the following efforts:

- We adopt asynchronous reward updating. If the updates are synchronous, there will be a delay between the rewards obtained by the model and the decision, which requires many rounds to be corrected. Using asynchronous updating makes it not necessary for the model to learn the match between reward and decision by itself, which significantly speeds up the training. Concurrently running increases the uncertainty of the environment. Using only local information will reduce the amount of information in the inputs to the model. Both settings can make it more difficult for the model to converge. Asynchronous updates can simplify the relationship between the model's learning to the state and the reward, thus accelerating the convergence of the model.
- Allowing different users' local models to use different parameters so that each user can schedule based on their own experience instead of using the same strategy. This also speeds up the training process.

The comparison of the convergence effects of the algorithms is shown in Section 5.

3 Problem Formulation

As the illustration given in Fig. 1, we consider a set of edge server nodes $\mathbb{E} = \{1, 2, ..., E\}$, and a set of user mobile devices $\mathbb{U} = \{1, 2, ..., U\}$. Both user devices and edge nodes have certain computing and network communication capabilities. We apply discrete fixed time slots $\mathbb{T} = \{1, 2, ..., T\}$ to simulate the real environment execution, with each time slot lasting Δ seconds. The system goes through a cycle in each time slot from task generation, scheduling, execution, and generating rewards.

3.1 User Device

Many edge computing tasks are employed for embedded device management. For example, path planning [13], intelligent home control [14], smart early warning systems [15], etc. The user devices must wait for the results before performing the next task. We assume that if the user device u checks that there is no running task at the beginning of each time slot, then it has a certain probability p_u of generating a new computational task. Due to the program's locality, it's costly to arbitrarily divide a task into different slices. We focus on the problem of indivisible task offloading, where each task is either executed on the local user device or offloaded to the edge server. Once a task has been generated, the task-relevant information is sent to the task scheduler, which makes the offload decision. The scheduler is deployed on the local user device. If the scheduler decides to process the task locally, the task will start running on the user device until it ends. Otherwise, the scheduler decides which edge server to offload the task to, and the user device transmits the task to the edge server through the wireless network channel. When the task has been transmitted, the edge server starts executing this task concurrently until it finishes, and the result of the task execution is returned to the user device.

3.1.1 Task

Denote the task created by user device u at time t as $w_{u,t}$. Each task has several basic attributes as described below. Denote $s(w)$ as the size of the data for task w, i.e., the amount of data that needs to be transferred is $s(w)$ *bit* when the task is transmitted through the wireless network channel. Denote the amount of computation for task w as $c(w)$, i.e., $c(w)$ CPU cycles are required to process the entire task. Before the t-th time slot, the computation amount has been executed of task w is denoted as $c(w,t)$. Each task w has a deadline $d(w)$. Suppose the current time is t'. If $t' - t > d(w_{u,t})$ and the task has not finished executing, i.e. $c(w_{u,t}, t') < c(w_{u,t})$, the system immediately stops the execution of the task and returns a notification that the task has failed. The deadline setting we use is the same as some previous work [2], [6]. A deadline is provided by users. If the task runs out of the deadline, it means that the task is being executed too slowly and is considered by users as disposable. Of course, we can set it to continue running after the deadline, but this will cause too many tasks to run on the server and affect the availability of the edge system.

3.1.2 Offloading Decision

For each new work $w_{u,t}$ generated in user device u_i, the user device separately creates an offloading decision utilizing the scheduling algorithm for each user device. Let the choice be denoted as $A(w_{u,t})$ with $A(w_{u,t}) \in \{0, 1, ..., E\}$. We set $A(w) = 0$ if the scheduler decides to run the task on the local user device. If $A(w) = e$, $e \in \mathbb{E}$, then the scheduler makes the decision to offload the task $w_{u,t}$ to the edge node e. We assume that task processing needs a minimum speed. So, while executing tasks concurrently on an edge server, there needs to be a maximum task amount, otherwise, it may result in a single task running too slowly. The scheduler examines if the task amount on the edge node has reached the maximum range of the edge node, and if it has reached the maximum, i.e., $n(t,e) \geq n^{max}(e)$, then the scheduler adjusts its decision $A(w)$ to 0, setting the task to execute locally. We set different running speed parameters for different hardware. We generated different devices by random combinations of this hardware. Due to implementation cost considerations, we did not consider the nonlinear variation in running speed for the same task under different hardware. However, due to the randomness of the operating parameters of the server device, the simulation environment can simulate this heterogeneity feature within a certain range.

TABLE 1
Summary of Main Notations

Notation	Description
$\mathbb{E}, e$	the set of edge computing servers and the e-th edge server
$\mathbb{U}, u$	the set of user devices and the u-th user device
$\mathbb{T}, t$	the set of time slots and the t-th time slot
$\mathbb{A}, a$	the action space and action of offloading decision
$w_{u,t}$	the task generate by user device u at t-th time slot
g_λ	Channel gain of channel λ
$d_{u,e}$	Distance between user device u and edge server e
$f(e)$	the computing capacity of equipment e
$p(u)$	the probability of generating a new task for user device u
$\lambda_{e,k}$	the k-th wireless channel of edge server e
ω	Background noise frequency
W	bandwidth of wireless channel
p^l	Path loss exponent
$\pi(a\|s;\theta)$	the policy of DRL algorithm
$c(t,w)$	the computation amount of task w at time slot t
$s(w)$, $c(w)$, $d(w)$	the transmission data size, computation amount and time limit of task w
$l^t(w)$, $l^e(w)$, $l^f(w)$	the time of waiting for idle channel, transmitting and executing of task w
$n(t,e)$, $n^{max}(e)$	the task pool size at time slot t, and the maximum amount of task pool size of edge server e
$r(w)$, $r^{co}(w)$, $r^{ch}(w)$, $r^{su}(w)$	the total reward, computation reward, transmit reward and success reward of task w

3.1.3 Local Computing

In the environment we designed, there can be only one running task on a local user device simultaneously. Locally executed tasks do not need to go through channel data transmission. So when the scheduler decides to execute the task locally, we can explicitly calculate the time when the task ends.

$$\begin{aligned} l^f(w_{u,t}) &= \frac{c(w_{u,t})}{f_u}, \\ A(w_{u,t}) &= 0, \end{aligned} \tag{1}$$

where f_u denotes the CPU frequency of user device u, and $l^f(w_{u,t})$ is the finish time of task $w_{u,t}$.

3.2 Edge Server

Concurrency and parallelism techniques have been widely employed in real-world systems, and using these techniques can idle I/O operations and effectively lower the average

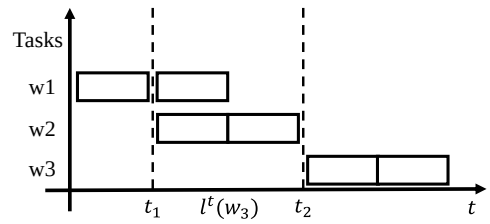


Fig. 4. An example of task transmission on one channel. The task $w3$ was generated at time slot t_1, but it takes $l^t(w_3)$ time slots to find an idle channel for transmission.

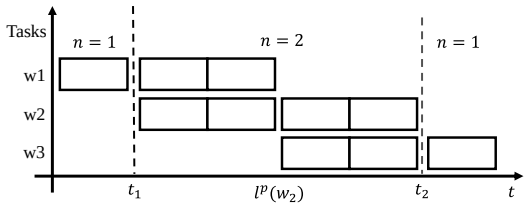


Fig. 5. An example of task execution on edge node. The edge server runs tasks concurrently between t_1 and t_2.

waiting time for tasks. Through virtualization approaches, it has become possible to standardize the speed of program execution control in cloud and edge computing services, in order to provide fair and stable computing resources for each user application. As a result, we assume that jobs are run concurrently in each edge node and that the computational speed gained by tasks run concurrently on each node is the same. The main difference between concurrent and serial execution is that concurrently run tasks are influenced not just by previously run tasks but also by tasks that join in later. In contrast, serially executed tasks are only affected by the tasks that came into the waiting queue before them.

3.2.1 Transmission Queue

The wireless network channel transmission follows a FIFO manner. We set that each edge server has a fixed positive integer of K channels for communicating with user devices, denoted by the notation $\lambda_{e,k}, k \in \{1, 2, ..., K\}$. Each channel can be connected to a single user device. After the user device that occupies the channel first completes its transmission, other user devices can connect to this channel and offload the task through it. The computational model of channel transmission speed is shown below.

All channel transmissions in our wireless network communication model are orthogonal to each other, and the channel communication speed is mainly affected by background noise power, path loss, and small-scale fading. The transmission rate from a mobile device u to an edge node e during task offloading can be calculated as follows:

$$r(u,e) = W \ \log_2 \ (1 + \frac{p_u g_{\lambda_{e,k}}}{\omega_{u,e} + \sum_{u' \in \mathbb{U}, u' \neq u} p_{u'} g_{\lambda_{e,k}}}). \quad (2)$$

Here we use $g_{\lambda_{e,k}} = d^{-p_l}(e,u)$ to denote the channel gain of the channel $\lambda e, k$, $d(e,u)$ represents the distance between the edge node e and the user device u, p_l represents the path loss exponent and the task is transmitted through the channel $\lambda e, k$. The transmission frequency of user device u is denoted using p_u, and g denotes the channel gain. W denotes the channel's bandwidth, and w denotes the background noise frequency of the channel. We assume that the transmission rate r of the channel is fixed.

At each time slot, after the scheduler on the user device makes the offloading decision, the tasks determined to be offloaded are connected to the channel of the corresponding edge node for transmission. However, since the channel amount of the edge node is limited, the latecomer tasks can only wait until the previous tasks have finished transmitting. Suppose a task w needs to be offloaded to the edge node e. Denoting the remaining transmitting time of the channel $\lambda_{e,k}$ at the t-th time slot is $l(t, \lambda_{e,k})$, indicating the channel will be idle until $t + l(t, \lambda_{e,k})$. If $l(t, \lambda_{e,k}) = 0$, then the task can choose the channel to transmit at time t.

So if there is no other task waiting for transmiting, the wait time to start transmission of task $w_{u,t}$ is $l^t(w) = \min\limits_{1 \leq i \leq K} l(t, \lambda_{e,k})$. If there are other tasks waiting for transmiting to the edge node e before $w_{u,t}$, denote them as $w_i, i \in \{1, 2, ..., n\}$. Suppose the task $w_{u,t}$ start transmitting at time slot $l^t(w_{u,t})$, it can be calculated as:

$$\begin{aligned} l^t(w_{u,t}) &= \min_{1 \leq i \leq k} l(t + T_b, \lambda_{e,k}), \\ T_b &= \max_{1 \leq i \leq n} l^t(w_i). \end{aligned} \quad (3)$$

As the example given in Fig. 4, the task w_3 need to wait until the former task finishes transmission and there is an idle channel. Although some tasks are generated at the same time, we order them according to their generation order, so there is no problem with many tasks competing for one channel.

If the task takes too long to transmit, the deadline is exceeded before the transmission is completed. Then the task is dropped. Thus, we can calculate the task transmission time as

$$l^e(w) = min \ \{\frac{s(w)}{r(u,e)}, \ d(w) - l^t(w)\}. \quad (4)$$

3.2.2 Edge Computing

After the tasks are transmitted in the channel, they will enter the task pool in the edge node for execution. We assume that the task pool size of each edge node is capped, and that all tasks running on the same machine can be executed at the same speed. Suppose the CPU computing frequency that device e can provide for task processing is f_e, where there are $n(t,e)$ tasks being executed in the task pool, including the tasks which just finished transmitting at the $t-1$-th time slot. Then each task on edge server e is executed at a CPU frequency of $\frac{f_e}{k}$.

Denote the remaining computation to be processed for task w at moment t is $c^r(t,w)$ with an initial value of $c(w)$ and at the end of the time slot its remaining computation size is

$$c(t+1, w) = c(t,w) - \frac{f_e \Delta}{n(t,e)}. \quad (5)$$

If the remaining computation size is zero, i.e. $c(t,w) = 0$, then the task execution ends. The task $w_{u,t}$'s execution time can be calculated as:

$$\begin{aligned} l^p(w_{u,t}) &= \min \ t_i, \\ s.t. \ c(w_{u,t}) &\leq \sum_{t=t+l^e(w_{u,t})}^{|\mathbb{T}|} \frac{f_e \Delta}{n(t,e)}. \end{aligned} \quad (6)$$

As the example given in Fig. 5, the three tasks have the same amount of computation, all of which are two units. However, due to concurrent execution, the edge server divides its computational resources equally, and task w_2 takes 4 time slots to finish.

The task process result can be calculated as:

$$Succ(w) = \begin{cases} True, & c(l^p(w), w) = 0, \\ False, & c(l^p(w), w) > 0. \end{cases} \tag{7}$$

Since the execution speed of a task is also affected by tasks that are added to the task pool later, we cannot predict precisely when the execution of a task will end, but we can know its range. If we assume that no new task will be added to the task pool of edge node e_j after time slice t, then the computation time of task w in it should be the minimum value of the task end time. The maximum value of the task end time is the end time of the task when the task pool keeps running at full capacity:

$$\frac{c(w_{u,t})}{f_e \Delta} \leq l^p(w_{u,t}) \leq \frac{n^{max}(e) c(w_{u,t})}{f_e \Delta}. \tag{8}$$

For terminated tasks, i.e., failed or completed tasks, the edge node returns the execution information of the task to the user device. Following the previous work, we assume that the size of the return value of a task is often much smaller than the size of the task input and the task itself. So, we do not consider the time required for the task output return in this work.

3.3 Problem Definition

For a task w that has completed its run, independent of whether it succeeds or fails in the run outcome, we define the amount of data completed in its transmission in the channel as $s^f(w)$, and the channel reward as:

$$r^{ch}(w) = 1 - \frac{s^f(w)}{s(w)}. \tag{9}$$

Denote the entire computation in the computing device at the end of the task is $c^f(w)$, and the computation reward can be computed as:

$$r^{co}(w) = 1 - \frac{c^{fin}(w)}{c(w)}. \tag{10}$$

Define the success reward of the task w as $r^{su}(w)$. If the task fails, the algorithm needs to get a penalty. If the task is successfully finished, then it will generate a positive reward. So we define the success reward as:

$$r^{su}(w) = \begin{cases} C_f, & Succ(w) = False, \\ C_s, & Succ(w) = True, \end{cases} \tag{11}$$

where C_f and C_s are both constant value, and $C_f < 0, C_s > 0$. We calculate the total reward of one task result as:

$$r(w) = r^{ch}(w) + r^{co}(w) + r^{su}(w). \tag{12}$$

Our objective is to minimize the total task latency and dropped task rate, which is equivalent to maximizing the long-term reward of all tasks. Therefore, our offloading scheduling problem can be formulated as follows:

$$\begin{aligned} R(A) = &\text{maximize} \sum_{t \in \mathbb{T}, u \in \mathbb{U}} r(w_{u,t}), \\ &\text{s.t. constraints (1) - (12) .} \end{aligned} \tag{13}$$

This formula means we want to find the best function mapping tasks to offloading decisions and obtain the maximum long-term reward. Our fundamental assumptions are that: 1.The scheduling algorithm can only get local information, not global information. 2.The task fails if it exceeds the deadline and stops running directly. 3.The edge server distributes the CPU capacity equally for each task. 4.The time usage of the scheduling algorithm is ignored. 1.These settings make our environment more complex while the state space is quite small, and the algorithm convergence will be more challenging. 2.We do not address the simulation of heterogeneous real servers, and also, the simulation of concurrency may not be accurate enough. 3.We do not take into account the time of the scheduling algorithm running, which is different from the real environment. Because the tasks are running concurrently on edge servers, this problem is NP-Hard and unpredictable. Traditional prediction or optimization-based approaches have difficulty achieving optimality in unstable environments. Deep learning models can capture the underlying probabilistic relationships between variables and hence acquire convergent methods even if the environment is uncertain. Also, if the environment changes dynamically, the approach may adapt with it. In the following chapter, we will present the DRL-based solution.

4 Algorithm Design

We propose a decentralized DRL algorithm in this chapter for tackling the offloading problem defined in the previous section. This scheduling algorithm works on the user devices and therefore does not require complete knowledge of the edge network. However, when there is a lack of information, the original DRL algorithm has difficulty converging. To address this issue, we improved the original DRL algorithm by incorporating asynchronous reward updating and localized model parameters, which improved the program's convergence performance.

4.1 DRL Model

In DRL, q-value can be viewed as the expected reward of taking an action in the current state. Traditional algorithms use a q-value table to capture the state and q-value information and are therefore not generalizable. Utilizing neural networks to fit the q-function enables agents to adapt to complex, dynamic systems without knowing the complete information of the system.

4.1.1 State

After the initialization of all user devices in each time slot is completed, the user devices observe the status information of all edge servers, channels, and tasks. This state information is used to obtain the corresponding offload decisions. For user device u, define $S_u(t) = (\mathcal{T}, \mathcal{Q}, \mathcal{C}, \mathcal{U})$ as the observation of the system at the t-th time slot:

- $\mathcal{T}$ denotes the information about the task $w_{u,t}$ which will be scheduled, including task size $s(w_{u,t})$, computation size $c(w_{u,t})$, and task deadline $d(w_{u,t})$.
- $\mathcal{Q}$ denotes the status information of channel transmission in edge servers, containing the percentage of transmission of tasks in all channels, the channel

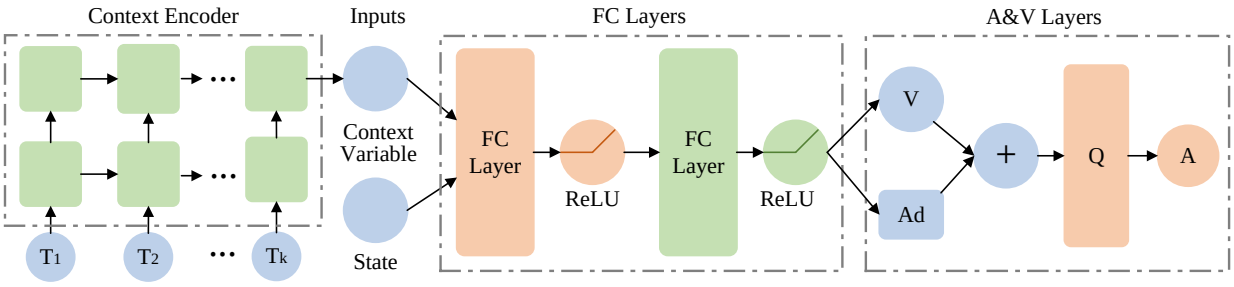


Fig. 6. The DOSA neural network architecture. T_i denotes the context information saved in user devices at time slot i. Q denotes the expected value of future reward. Ad denotes the advantage function which outputs offsets of action can bring, and V denotes the mean value of Q. A denotes the predicted best action.

transmission speed, and the maximum task capacity of the channel.

- C denotes the execution information of tasks in the edge server (ES), containing the percentage of tasks executed, the CPU execution speed of ES, the maximum task capacity of ES, and whether the task reaches the deadline.
- $\mathcal{U}$ denotes the status of all user devices, containing their processing capacity, transmission efficiency, etc.

The dimension size of the states may change in each time slot. For example, the task amount in the task pool of edge nodes and the number of tasks in the transmission channel may change over time. To guarantee that the dimension of all states remains constant, we set the dimension size of the states to stay at the maximum feasible value, with all 0s at vacancies.

For each user $u, u \in \mathbb{U}$, the state information it sees is different. Since our method is decentralized, the decisions of each user device are independent of any other and do not need the complete information of the system. Therefore, the observation of a single user device does not include other user devices' states.

4.1.2 Action

After observing the environment and obtaining the state of the environment in this time slot, the system feeds the observation to the scheduler and gets the offloading decision, $A(w), A(w) \in \mathbb{A}$. Defining $\mathbb{A}$ as action space, ie., $\mathbb{A} = \{0, 1, ..., E\}$, where $A(w) = 0$ means the task is running in the local user device and $A(w) > 0$ means offloading to the $A(w)$-th edge server.

It is worth noting that the offloading decision generated by the scheduler is not always the actual policy executed by the environment. When the tasks on the server exceed the maximum quantity, the scheduler may still produce the decision to offload to the server, but this task can only be executed locally. We want the choice made by the model each time to be the optimal choice, so if the model chooses a server that has exceeded the max task amount, it should only be run locally. If not, it is not beneficial for the model to learn the ability to choose the optimal server at the first time.

4.1.3 Context

To learn from the past experience, user devices save the context information in DOSA. Denote the observed states, actions, and rewards obtained by the user device u at the t-th time slot as $S_{u,t}$, $A_{u,t}$, and $R_{u,t}$, respectively, and the matrix of all observed state sequences between time t_1 and t_2 as $S_u[t_1, t_2]$, $A_u[t_1, t_2]$,and $R_u[t_1, t_2]$.

Context is defined as the collection of states, actions, and rewards observed between a range of time slots. Setting the context size to k, we get $C_{u,t}(k) = (S_u[t-k, t], A_u[t-k, t], R_u[t-k, t])$, which is the matrix of all states, actions, and rewards in the first k time slots, respectively.

4.1.4 Reward

The deep reinforcement learning algorithm's reward function directly impacts how successfully the learning is done. When a task is finished, we compute the reward value derived from its offloading choice depending on the task's execution results and return it to the user device. We set the reward function to match the optimization problem in (13) as follows:

$$R(S_{u,t}, A_{u,t}) = r(w_{u,t}). \tag{14}$$

The purpose of a DRL algorithm is to find the optimal policy π_u for each user device u, which is a mapping from the system state to the action that maximizes the system's long-term reward. It can be represented as follows:

$$\pi_u^* = \underset{\pi_u}{\operatorname{argmax}} \; E\left[\sum_{t \in \mathbb{T}} \gamma^{t-1} R(S_{u,t}, A_{u,t}) \mid \pi_u\right]. \tag{15}$$

4.2 Network Architecture

We propose a neural network to fit the DRL algorithm's Q-value function. To reduce the computational cost of user devices, we divide the network into two parts. The first part requires more computation, which we place on the server. The second part, which is less computationally intensive, is placed on the user devices. Each user device trains the algorithm using its own neural network, which implies that the network parameters are different across user devices. This helps with the algorithm's fast convergence.

Fig. 6 depicts the network architecture of our DRL method. The input of the first part is the context. The network layer is an RNN, and the output is hidden states. This is used to determine the uncertainty of our decisions for the environment. The second part's input is the output of GRU and the system's state at the current time slot, and the network structure is two fully connected layers. This component is trained to obtain the features of the system

TABLE 2
The Neural Network and Training Hyperparameters

Hyperparameter	Value	Hyperparameter	Value
Encoder Layers	2	Encoder Layer Type	GRU
Learning Rate β	3×10^{-4}	Discount Factor γ	0.99
Encoder Hidden Units	10	Clipping Constant ϵ	0.1
Optimization Method	Adam	Activation function	Relu

state. The third part is the output layer, a one-layer, fully connected layer. The output of this layer is the predicted q-values and is used to select the optimal action output based on the expected q-value of each action state pair. Denote θ as the parameters of the network, which contains the parameters of all layers. In DOSA, every mobile user device has its own decision network, so we denote θ_u as the network parameters for user device u.

4.2.1 Context Layer

Some previous studies have employed meta-reinforcement learning approaches to improve the capacity of models to adapt to new environments swiftly. However, some research has shown that using a latent context variable also increases generalizability and reaches results comparable to meta-learning approaches. To increase the efficiency of our algorithm in a real-world edge environment, we employ GRU [16] to encode context information, which has higher computational efficiency than LSTM [17], while there is no significant difference in prediction accuracy. The input of GRU is the context information of the previous n steps. Fig. 2 shows the hyperparameters of the network, and each GRU unit has a hidden neuron that learns the interrelationships between the inputs, through which the model's estimation of the environmental uncertainty is revealed. When the model meets a new environment, these feature values may enable the model to adjust its policy swiftly.

4.2.2 FC Layer

The Fully Connected (FC) layers extract information of the state of the environment and tasks from the output from the context layer and the system observation at current time slot. The observations and the context variables of different user devices are different from each other, while the edge node and channel transmission information are the same.

4.2.3 Output Layer

We use the Dueling DQN technique [18] to estimate the q-values of state-action pairs. The primary approach is to split the q-values into two parts: action advantage values and a state value. Previous works show that the Dueling DQN technique accelerates the learning efficiency and improves the training effectiveness.

Both advantage (A) and state value (V) layers consist of a fully connected layer and a Rectified Linear Unit (ReLU). The A layer is responsible for proposing the probability corresponding to the action. The V layer is responsible for proposing the average probability corresponding to the state. Denote $A_u^{net}(S, a, C; \theta_u)$ and $V_u^{net}(S, C; \theta_u)$ as the output

Algorithm 1 : DOSA Algorithm on User Device u

```
Initialize context buffer q_u;
Initialize local neural network parameters θ_u for Advantage and State value Layer;
for time slot t ∈ 𝕋 do
    if new task w_{u,t} then
        Send S_{u,t} to scheduling server, receive the intermediate representation O_{u,t}.
        Select action A(w_{u,t}) through (18);
        if A(w_{u,t}) > 0 then
            Offload the task to the A(w_{u,t})-th edge server;
        else
            Run the task at user device;
        end if
    end if
    if receive r(w_{u,t'}) then
        Obtain S_{u,t}, r(w_{u,t'}) and C_{u,t'}(k);
        Send the local offloading history (S_{u,t'}, A(w_{u,t'}), r(w_{u,t'}), C_{u,t'}(k), S_{u,t}) to scheduling server;
        Update local network parameters;
        Train the local Advantage layer and State Value layer by local memory;
    end if
end for
```

of advantage layer and state value layer, we can calculate the output Q value as:

$$\begin{aligned} Q_u(S_{u,t}, a, C_{u,t}(k);\ \theta_{u_i}) = V_u^{net}(S_{u,t}, C_{u,t}(k); \theta_u) + \\ A_u^{net}(S_{u,t}, a, C_{u,t}(k); \theta_u) - \\ \frac{1}{|\mathbb{A}|} \sum_{a \in \mathbb{A}} A_u^{net}(S_{u,t}, a, C_{u,t}(k); \theta_u). \end{aligned} \tag{16}$$

Using the Q values for the user device u we may obtain the optimal offloading decision given by the model at the t-th time slot as:

$$a_{u,t}^* = \underset{a \in \mathbb{A}}{argmax}\, Q_u(S_{u,t}, a, C_{u,t}(k); \theta_u). \tag{17}$$

4.3 DOSA Algorithm

This chapter introduces our scheduling algorithm. We deploy the DRL model proposed in the previous chapter to each user device. As the system is running, the model constantly optimizes the local scheduling strategy based on the reward earned from each offloading decision. Unlike traditional centralized scheduling approaches, only local user information is required as our approach is decentralized directly to the user. It also makes it harder for the algorithm to converge since the environment takes into consideration concurrent running jobs. We apply an asynchronous update reward method while supporting a personalized model for each user i.e. the parameters of the user model might be varied, which speeds up the convergence of the algorithm.

4.3.1 Offloading Algorithm on User Device

The methods in previous works deploy a scheduler in each user device that contains a complete policy network. This is impractical and not necessary. Since neural networks are rather computationally expensive and the local policy

neural network needs to synchronize the parameters on scheduling server-side, user devices will consume too much time and computational resources in communication and policy network computation. In this work, we cut the policy network such that each user device maintains only the Advantage and State Value layers while the other layers are deployed on the scheduling server. This way, just two layers of parameters need to be synchronized periodically and used for the local decision computation.

The user devices retain the context information in a queue and add the information for this choice when each task gives the result. The user device periodically requests that the algorithm model's parameters deployed in the scheduling server be updated locally throughout each episode of the algorithm run. If a user device generates a new task, it enters the observed system state at the time slot, information about the new task participating in scheduling, and fixed-length context information into the scheduling server model and gets the intermediate representation (IR). The local Advantage and State value layers use the IR to make the offload decision.

Because the model knows nothing about the environment when the algorithm initially starts, random exploration is necessary. Otherwise, the possibility of falling into a local optimum exists. As a result, we define a stochastic parameter $\epsilon \in (0, 1)$, and the user device makes a choice for each new job as follows:

$$A(w_{u,t}) = \begin{cases} a_u^*, & w.p.\epsilon, \\ \text{random action in } \mathbb{A}, & w.p.1-\epsilon, \end{cases} \tag{18}$$

where "w.p. ϵ" means the probability for the choice is ϵ.

The ϵ is raised at each decision point, gradually increasing the proportion of model decisions but not allowing the model to make 100% of them. This method enables the model to escape from local optimum spots effectively.

If the task must be executed locally, the user device executes it until it is finished. If the job is to be executed on the edge node, the user device transmits this task throughout the channel and waits for the task termination message. When the end-of-task message is returned, the user device calculates the reward obtained by this task offloading decision and reports it to the scheduling server. This is an asynchronous updating procedure. In conventional reinforcement learning, when each task decision is made and accomplished, the environment delivers the reward for that decision. This does not match the requirements of the task scheduling environment since the execution of the job requires numerous time slots. Therefore, changes need to be performed asynchronously. After computing the task's gain, the user device will transmit the complete decision information (state, context, choice, and reward) to the buffer of the edge node to enable future training.

4.3.2 Training Algorithm on Edge Server

Due to the need to synchronize information, our algorithm is designed to be trained mainly on a broker node server. The server must keep two deep neural network models: an evaluation network Net_{eval} and a target network Net_{target}. The evaluation network determines which action corresponds to the state. The target network is used to forecast the Q value of the states listed below to maximize the long-term reward. The evaluation network is periodically updated with the target network's parameters during training, and the two networks have identical structures.

Algorithm 2 : DOSA Algorithm on Scheduling Server

```
Randomly initialize the parameters of meta policy, θ;
Initialize prioritized replay memory M;
while True do
    if receive transition from device u then
        Store the message (S_{u,t'}, y(w_{u,t'}), r(wu, t'),
        C_{u,t'}(k), S_{u,t}) in replay memory M;
        Random sample P experiences in M;
        for sample p in P do
            Compute Q-value loss through (21);
            Update parameters θ_eval by Adam optimizer;
            Compute TDE through (19);
            Update sample p's priority in M;
        end for
    else if update request from user device u then
        Sent parameters of Value State Layer and
        Advantage Layer to u;
    else if receive observation from u then
        Generate intermediate representation O_{u,t} by
        Net_eval and send it to u;
    end if
    if time to update Net_target then
        θ_target := θ_eval;
    end if
end while
```

The scheduling algorithm deployed on a broker node principally collects data from each user device to train the neural network. Whenever a task terminates, the user device calculates its reward value and transmits it to the scheduling server. Since we are not sure when the task terminates, at each moment, the scheduling server may receive task execution information from the user devices. In order to store this information in real-time, we maintain a replay memory in the scheduling server that saves all task offloading information for subsequent model training.

Also, to reduce the communication time, we update the model parameters on the server to the user devices when the server receives the task information from the users. Since the user device may generate a new task only after the previous task is finished, this reduces the time of parameter updating and guarantees that each device is synchronized with the newest network parameters.

In order to speed up the training, we use the Prioritized Experience Replay Buffer technique [19]. We use TD error [20] as the priority of experience, i.e., the larger the loss gradient generated by this experience, the higher its priority to be sampled. The temporal difference error (TDE) is calculated as follows:

$$\begin{aligned} TDE = r(w_{u,t}) - Q_u(A(w_{u,t}), y(w_{u,t}), C_{u,t}(k)) + \\ \gamma Q_u(S_{u,t'}, y(w_{u,t'}), C_{u,t'}(k)). \end{aligned} \tag{19}$$

For model training, we employ the Double DQN technique (DDQN) [21]. DDQN can effectively address the q-value overestimation problem. Two neural networks Net_{eval} and Net_{target} are deployed on scheduling server. Whenever

the server receives a task intermediate representation request, it generates the output by the Net_{eval}. The Net_{target} updates its parameters by loading Net_{eval}'s parameters periodically.

The scheduling server trains the neural network once in each time slot. First, the algorithm randomly samples a particular amount (batch size) of historical records P in the replay memory and sends these records into the network for training to minimize the gap between the anticipated Q value and the actual reward earned. For a sample $(S_{u,t'}, y(w_{u,t'}), r(w_{u,t'}), C_{u,t'}(k), S_{u,t})$, we can first derive the action for maximum long-term reward in the next time slot $a^*(t)$ using Net_{target}:

$$a^*(t) = \underset{a \in \mathbb{A}}{\operatorname{argmin}} Q(S_{u,t}, a, C_{u,t}(k); \theta_{eval}). \tag{20}$$

Then we calculate the expected long-term reward Q_u^{target} as follows:

$$Q^*_{target,u} = r(w_{u,t'}) + \gamma Q(S_{u,t}, a^*(t), C_{u,t}(k); \theta_{target}). \tag{21}$$

Loss function describe the distance between the expected Q value and our estimation, we calculate the loss as follows:

$$\begin{aligned} L(\theta_{eval}, Q^*_{target}) = & \frac{1}{|P|} \sum_{p \in P} | \, Q(p(S), p(A), p(C); \theta_{eval}) \\ & - Q^*_{target,p}|^2. \end{aligned} \tag{22}$$

Then we can perform the gradient decent algorithm and Adam optimizer [22] to iterative optimization the Net_{eval}. Then we calculate the TDE (TD error) of the sample and update it's sample priority in replay memory.

5 Experimental Results

In this part, we analyze the performance of DOSA using extensive simulations. We first present the evaluation settings. Then, we introduce the baseline algorithms for comparison. Finally, we provide the ablation experiments for context variable and prioritized replay memory.

5.1 Environment Setup

We consider a 5 edge nodes edge computing environment. The parameters of the edge network configuration are listed in the table below. We presume that the scheduling algorithm does not consume time slots. In the simulated environment, task processing and transmission occupy time slots. Because random parameters influence task generation in the environment and network transmission, the results of each running episode may differ.

We clear all tasks in the system, set the time slot index to 0, and randomly initialize the neural network model in the algorithm at the start of each episode. We specify the maximum time slot index, and if the algorithm does not finish within this time slot, the system forces the episode to stop. We compare DOSA to the five algorithms listed below:

- All locally (AL): All tasks are executed locally on the user device. This may cause a high ratio of dropped tasks.
- Greedy (GR): The greedy algorithm assumes that no new tasks are created in the system afterwards, calculates the total possible latency for each offloading decision, and selects the decision that is likely to result in the shortest latency. Because the greedy algorithm cannot predict future changes in the system state, it chooses to complete each task as quickly as possible, potentially causing network congestion.
- All to server (AS): This approach offloads all tasks to the edge server node, analogous to the greedy algorithm, which eliminates the option of running tasks on the local user device. It is more likely to cause network congestion than the greedy algorithm.
- All random choose (RA): Choose an offloading decision at random from all possible offloading decisions.
- Multi-agent Deep Deterministic Policy Gradient (MADDPG): The MADDPG algorithm [23] is a state-of-the-art multi-agent DRL algorithm where decentralized agents learn a centralized critic based on the observations and actions of all agents.
- Proximal Policy Optimization (PPO): The PPO algorithm [24] is a widely used state-of-art DRL algorithm. It is more robust and stable in training than other algorithms, and it is well suitable for learning with a high-dimensional state.

TABLE 3
Range of environment configuration parameters

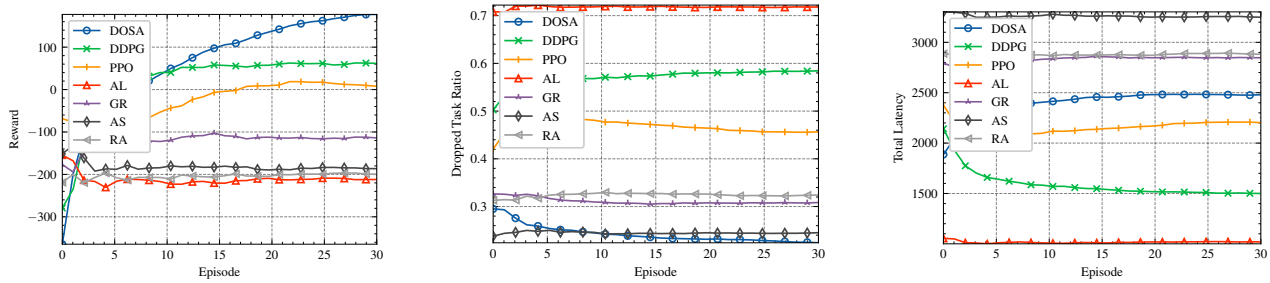
Parameter	Range
Edge node frequency	5 to 25 GHz
Task Size	50 to 5000 MB
User Device frequency	0.1 to 2 GHz
Channel Speed	400 to 2000 MB/s
Probability of task generate	0.3 to 0.5
Edge node task pool capacity	5 to 20 tasks

5.2 Result Analysis

To illustrate the performance of the DOSA algorithm, we run it in the same environment as other baseline algorithms. The methods employed in the DOSA algorithm are then exposed to ablation tests. The following table lists the parameter settings in the simulated environment.

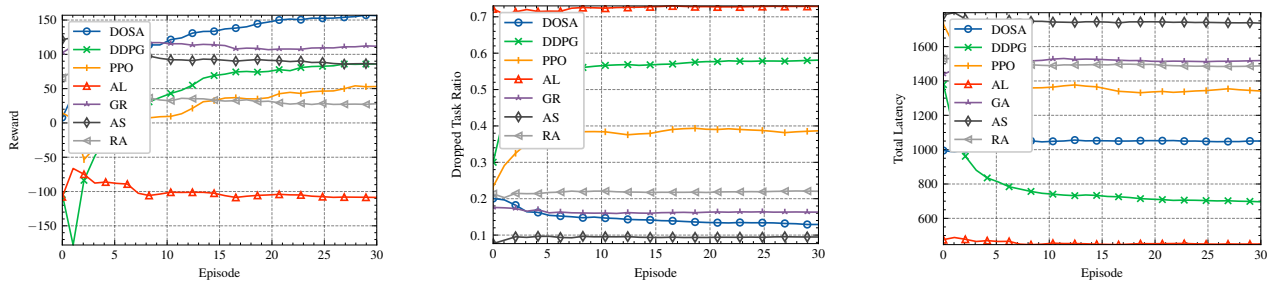
5.2.1 Performance and Convergence

To validate the convergence of the model, the variation of the rewards with episode for 30 episodes of training our algorithm in the same environment is displayed in Fig. 7. We can see that the final reward of the DOSA algorithm is substantially higher than the other algorithms. The learning speed of the PPO algorithm is not as fast as that of the DOSA. To exhibit the flexibility of DOSA in different environments, the number of user devices is raised to see the change in the algorithm impact. It can be observed that DOSA still retains optimal outcomes in different scale environments. The experimental results show that MADDPG is slightly better than the PPO algorithm and DOSA is still the best. We argue that this is due to the fact that the MADDPG algorithm is continuous, and although the gumbel softmax [25] can be employed to make it applicable to discrete environments, it still does not learn as well as the discrete DQN algorithm. Second, the unpredictability of our environment is already high, and there are



(a) Reward vs. Epside. (b) Dropped task ratio vs. Epside. (c) Average Latency vs. Epside.

Fig. 7. Evaluation results for 75 ue and 5 es environment.



(a) Reward vs. Epside. (b) Dropped task ratio vs. Epside. (c) Average Latency vs. Epside.

Fig. 8. Evaluation results for 35 ue and 5 es environment.

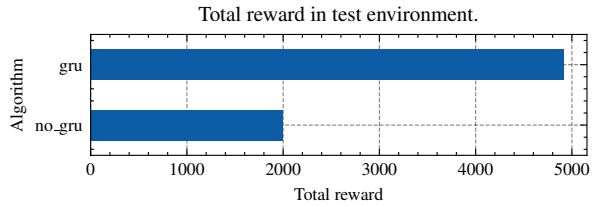


Fig. 9. Total reward of DOSA and no context variable DOSA in different environments.

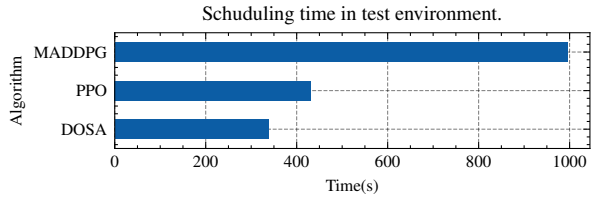


Fig. 10. Scheduling time in test environment of different DRL algorithms.

not enough environmental parameters, which makes it more difficult for the multi-intelligence algorithm to converge due to the setting that global information cannot be utilized. We believe that the PPO algorithm does not work well for the same reason. Since our environment returns a more frequent and accurate rewards, the value-based learning algorithm is naturally superior to the Actor Critic method.

Fig. 10 shows the scheduling time required for the different algorithms to schedule the tasks for 30 rounds. It can be seen that the scheduling time of DOSA is the lowest. The reason for this has been pointed out in the Section 2: we reduce the amount of information that needs to be passed in the channel by not using global information. The efficiency of the model operation is also improved due to the reduction of the state space of the system. Secondly, since we only need a neural network, our runs are also faster than MADDPG and PPO which need two networks.

Fig. 8 shows the total latency and dropped task ratio for each episode in training in different environments, and it can be shown that the dropped task ratio of DOSA is lower than that of other algorithms. Total latency denotes the latency of all successfully executed tasks. The reason for the low latency in the All-local method is that the success rate of local runs is low, and most locally executed tasks fail soon, so their latency is not counted. Therefore, the PPO and All-Local algorithms with higher task failure rates will have lower Latency. Hence, our reward value design can effectively reflect the optimization objective. The DOSA algorithm can jointly optimize the dropped task ratio and latency objectives.

In order to evaluate the adaptability in different environments, the network parameters in our environment for each experiment were randomly generated. Table. 3 illustrates the range we specified for the parameters of the network. The DOSA got the best performance in all random environments, and the learning rate was quicker than PPO and MADDPG. When the number of user devices rises, DOSA can efficiently alter the policy and swiftly adapt to the new environment. Although we employ the asynchronous reward update mechanism on the PPO and MADDPG algorithm, the convergence time of the PPO and MADDPG is slower.

5.2.2 Impact of Asynchronous Reward Update

Asynchronous reward update: DOSA delivers the task reward after the task is completed, rather than returning to the environment. We suppose that this will speed up the convergence of the algorithm.

For immediate reward update DOSA, we update the model at each time slot by computing the reward (computation and dropped tasks) generated by the full environment in that time slot. When comparing this DOSA with the immediate reward update to the DOSA with the asynchronous reward update, we observe that the DOSA with the delayed asynchronous reward update is much quicker and surpasses the DOSA without it. Fig. 11 illustrates the reward change across training episodes. It can be seen that the dropped task rate of DOSA without asynchronous updates approaches 50%. This is 20% higher than for DOSA utilizing asynchronous updates. Although the total latency of successful tasks for DOSA is greater than for DOSA without asynchronous updates, this is because DOSA has more finished tasks.

5.2.3 Impact of Prioritized Replay Memory

Since many tasks in the environment contain roughly identical information, only a proportion of the samples will substantially impact model training. Therefore, adopting prioritized replay memory for sampling can enhance the speed of training. Intuitively, the higher the loss created by a sample, the larger the increase it is expected to achieve.

In Fig. 12a, we compare the algorithm that employs prioritized replay memory with the algorithm that uses random sampling. As can be observed, the algorithm without prioritized replay memory learns slower than the algorithm with prioritized replay memory. But eventually, they tend to have the same outcome. As the number of training episodes increases, the randomly sampled model also iterates through all the samples.

5.2.4 Impact of decentralized Training

In DOSA, different offloading policies are generated for different user devices utilizing a decentralized way to train the model. The advantage of this strategy is that the local user devices do not need to know the information of other devices, but need to synchronize the parameters periodically. In Fig. 12b, we remove the Advantage and State Value Layer from all user device schedulers; each task offloading decision is made by the model in the scheduling server and compared with the method for decentralized decision making. The experiments reveal that the decentralized algorithm learns more effectively than the centralized one, and the final output is slightly better than the centralized one.

5.2.5 Impact of Context Variable

Encoding context information: Some meta-reinforcement learning research [26]–[28] has demonstrated that a context variable may significantly increase models' flexibility in different environments. Work on Meta-Q-learning [29] shows that a simple context variable can fulfill this objective. We expect this technique will enable the algorithm to learn quicker in different environments.

We randomly generated ten testing environments in the range of parameters in Table. 3, which are of the same size (number of user devices and edge nodes) but with different network and computational configurations (network speed, computational speed). Ten episodes are trained in each environment to examine the performance of the DOSA algorithm with and without a context variable.

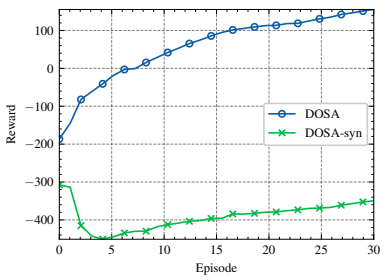
Fig. 9 shows the comparison of the total reward in all environments. Fig. 12c shows the change of the reward of DOSA and no context variable DOSA in one of the environments. It can be seen that the total value of the reward of DOSA is higher than the no context variable DOSA, and the learning rate of the agent with context encoding is a bit faster.

6 Related Work

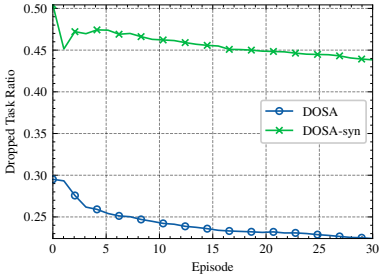
6.1 Task Offloading in MEC

Mobile edge computing techniques can effectively improve the efficiency of application computing in user devices, but as shown in Fig. 1, the variety and connection relationship of user devices are complex, and it is a challenge to match between multiple heterogeneous devices and tasks. Many previous works [3], [30], [31] in MEC treated the task offloading problem as a traditional mathematical optimization problem, solving it with Dynamic Programming, Mathematical Programming, and Genetic algorithms to find the best scheduling policy. For example, Xiao *et al.* [30] study the workload offloading problem in fog computing to maximize the QoS (Quality of Service). They proposed a decentralized ADMM-based method where an edge server can offload the received tasks to a cloud computing server or other edge servers. They formulated a non-convex optimization problem and divided it into convex sub-problems. Zhao *et al.* [3] considered a task offloading problem with task dependencies in their work and considered the impact of service caching and placement in edge devices. This is relaxed to a linear optimization problem and a convex optimization problem, respectively, and solved using a convex optimization solver. Eshraghi *et al.* [31] designed an algorithm that optimizes the offloading choices of mobile devices, considering their unknown computation requirements. The problem with traditional methods is the overall efficiency over long periods is not considered, only one or a small quantitative part of the task is optimized. They are not suitable for edge computing environments that run for long periods and change dynamically.

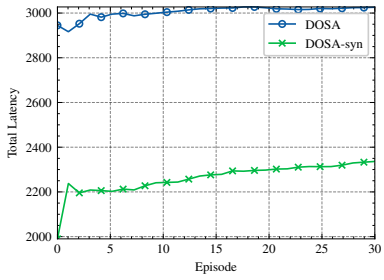
Some other works [4], [32] treat task offloading as a game theory problem, expecting to schedule multiple user devices by finding offloading strategies that can reach a Nash equilibrium state. For example, Chen *et al.* [32] considered a many-to-many task offloading problem, where players have to consider the offloading decision and choose the transmission channel. They proved that the game reaches Nash equilibrium when the beneficiary players reach the maximum value. After that, they developed a decentralized decision algorithm to reach the Nash equilibrium. In the work of Ding *et al.* [4], a game was developed where each user makes a selfish decision and expects to minimize long-term cost. Moreover, two game-based algorithms were developed and proved that they can reach Nash equilibrium. These algorithms are not applicable to dynamically changing environments and are too computationally expensive



(a) Reward vs. Episode of DOSA and synchronized DOSA.

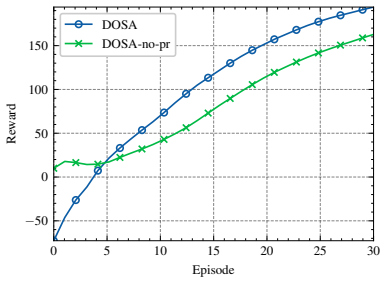


(b) Dropped Task Ratio vs. Episode of DOSA and synchronized DOSA.

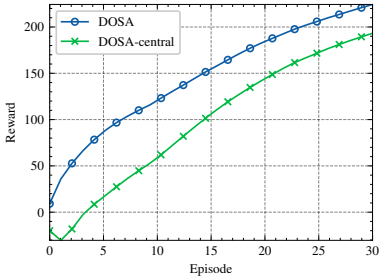


(c) Finished Task Latency vs. Episode of DOSA and synchronized DOSA.

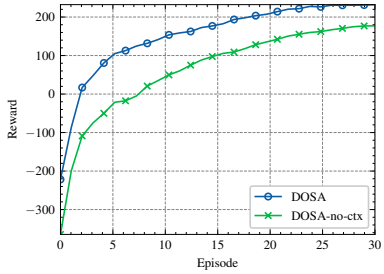
Fig. 11. Ablation experiment result of asynchronous reward update.



(a) Comparison of no prioritized replay memory DOSA and DOSA.



(b) Comparison of centralized DOSA and decentralized DOSA.



(c) Comparison of DOSA and no context variable DOSA.

Fig. 12. Impact of prioritized replay memory, decentralized training and context variable.

with the main value being in the design of the rules, rather than the actual scheduling.

6.2 DRL Task Offloading Methods

Unlike traditional methods, some previous works [2], [6], [7], [10], [33] use DRL methods to minimize the computational and communication costs of the system over long-term execution. Since DRL methods perform model training through the system state transitions, they can adapt to dynamic environments. However, this also brings the problem of the high training cost. In the work of Huang *et al.* [2], power consumption and computational efficiency are jointly considered and transformed into a non-convex optimization problem. This non-convex optimization problem is divided into two parts: task offloading decision and channel and charging resource allocation. The first part is approximately solved using a DRL method, and the second part is solved using a convex optimization problem solver. This work does not explore the problem of distributed execution and the cost necessary for task transmission. The work by Qiu *et al.* [7] obtains samples from different environments for training employing decentralized reinforcement learning. They use a deep network neuroevolution algorithm to obtain the optimal network model and employed the n-step learning approach to improve the speed of model training. This work focuses on employing multiple DRL algorithms to enhance the scheduling of the model but ignores the cost required to run the model for scheduling. Tang *et al.* [10] propose a D3QN-based reinforcement learning algorithm where each user device makes decisions independently and uses LSTM to predict the next system state. Wang *et al.* [6] use a meta-reinforcement learning approach [33] to offload optimization for tasks with dependencies and use a seq2seq network to encode the task Directed Acyclic Graphs (DAG). It is shown that the meta-learning algorithm can be trained in different environments and adapt quickly to new environments. Both work of Tang *et al.* and Wang *et al.* suffer from high running costs, and both models are challenging to implement on user devices with low computational capacity. In order to safeguard user privacy and execute lightweight scheduling, it is important to build a new algorithm that can run in a distributed way and move the computational cost to the server.

Unlike these works, our algorithm cuts the neural network so the user device does not have to save large networks. We also consider the scenario of concurrent task execution, which was not considered in previous works.

7 Conclusion

In this work, we study the indivisible and delay-aware task offloading problem in edge computing. We designed a decentralized scheduling algorithm based on deep reinforcement learning, which does not need to know the complete information of the environment but learns in the process of offloading so that it can adapt to the dynamic network. The simulation result shows that our algorithm can significantly reduce task delay and drop the task loss ratio when facing tasks and data-intensive scenarios.

This work can be extended in the following directions: First, it is interesting to consider DAG modeling tasks and offload them to different edge servers for execution. Second, it's helpful to consider edge service cache. Different kinds of servers may cache various services. For different types of tasks, it is necessary to consider the service cache state on the server for scheduling. The above technologies can make our scheduling algorithm better suited to real-world environments.

8 Acknowledgments

This work was supported by National Natural Science Foundation of China (No.61872175), Natural Science Foundation of Jiangsu Province (Grant No.BK20201250). Jidong Ge is the corresponding author.

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, 2016.

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

[3] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading tasks with dependency and service caching in mobile edge computing," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 11, pp. 2777–2792, 2021.

[4] Y. Ding, K. Li, C. Liu, and K. Li, "A potential game theoretic approach to computation offloading strategy optimization in end-edge-cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1503–1519, 2022.

[5] X. Wang, Z. Ning, and S. Guo, "Multi-agent imitation learning for pervasive edge computing: A decentralized computation offloading algorithm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 411–425, 2021.

[6] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 1, pp. 242–253, 2021.

[7] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 5, pp. 1085–1101, 2021.

[8] Z. Chen and X. Wang, "Decentralized computation offloading for multi-user mobile edge computing: a deep reinforcement learning approach," *EURASIP J. Wirel. Commun. Netw.*, vol. 2020, no. 1, p. 188, 2020.

[9] M. Goudarzi, M. S. Palaniswami, and R. Buyya, "A distributed deep reinforcement learning technique for application placement in edge and fog computing environments," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[10] M. Tang and V. W. Wong, "Deep reinforcement learning for task offloading in mobile edge computing systems," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2020.

[11] C. Cicconetti, M. Conti, and A. Passarella, "Architecture and performance evaluation of distributed computation offloading in edge computing," *Simul. Model. Pract. Theory*, vol. 101, p. 102007, 2020.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nat.*, vol. 518, no. 7540, pp. 529–533, 2015.

[13] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[14] P. Verma and S. K. Sood, "Fog assisted-iot enabled patient health monitoring in smart homes," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1789–1796, 2018.

[15] M. Syafrudin, N. L. Fitriyani, G. Alfian, and J. Rhee, "An affordable fast early warning system for edge computing in assembly line," *Applied Sciences*, vol. 9, no. 1, 2019.

[16] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, A. Moschitti, B. Pang, and W. Daelemans, Eds. ACL, 2014, pp. 1724–1734.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[18] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1995–2003.

[19] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016.

[20] G. Tesauro, "Temporal difference learning and td-gammon," *J. Int. Comput. Games Assoc.*, vol. 18, no. 2, p. 88, 1995.

[21] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, D. Schuurmans and M. P. Wellman, Eds. AAAI Press, 2016, pp. 2094–2100.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[23] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 6379–6390.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[25] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *arXiv preprint arXiv:1611.01144*, 2016.

[26] M. J. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015*. AAAI Press, 2015, pp. 29–37.

[27] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," *CoRR*, vol. abs/1512.04455, 2015.

[28] K. Rakelly, A. Zhou, C. Finn, S. Levine, and D. Quillen, "Efficient off-policy meta-reinforcement learning via probabilistic context variables," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 5331–5340.

[29] R. Fakoor, P. Chaudhari, S. Soatto, and A. J. Smola, "Meta-q-learning," in *International Conference on Learning Representations*, 2020.

[30] Y. Xiao and M. Krunz, "Qoe and power efficiency tradeoff for fog computing networks with fog node cooperation," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[31] N. Eshraghi and B. Liang, "Joint offloading decision and resource allocation with uncertain task computing requirement," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1414–1422.

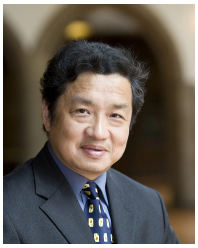
[32] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016.

[33] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 2017, pp. 1126–1135.

Ye Fan received the BS degree from the Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2021, where he is currently working toward the Master degree under the supervision of associate professor Jidong Ge. Currently, his research interests include mobile edge computing and reinforcement learning.

Jidong Ge (Member, IEEE) is an Associate Professor at Software Institute, Nanjing University. He is also a member of the State Key Laboratory for Novel Software Technology. He received his PhD degree in Computer Science from Nanjing University in 2007. His current research interests include decentralized computing and edge computing, services computing, natural language processing and intelligent software engineering, machine learning and deep learning. His research results have been published in more than 100 papers in international journals and conference proceedings including IEEE TPDS, IEEE TMC, IEEE/ACM TNET, IEEE TSC, IEEE/ACM TASLP, ACM TKDD, JASE, COMNET, JPDC, FGCS, JSS, Inf. Sci., JNCA, JSEP, ESA, ExpSys, ICSE, AAAI, EMNLP, ASE, IWQoS, GlobeCom etc.

Sheng Zhang (Member, IEEE) received the BS and PhD degrees from Nanjing University, Nanjing, China, in 2008 and 2014, respectively. He is currently an associate professor with the Department of Computer Science and Technology, Nanjing University, China. He is also a member of the State Key Laboratory for Novel Software Technology. His research interests include cloud computing and edge computing. To date, he has published more than 80 papers, including those appeared in the IEEE Transactions on Mobile Computing, IEEE/ACM Transactions on Networking, IEEE Transactions on Parallel and decentralized Systems, IEEE Transactions on Computers, MobiHoc, ICDCS, INFOCOM, SECON, IWQoS, and ICPP. He received the Best Paper Award of IEEE ICCCN 2020 and the Best Paper Runner-Up Award of IEEE MASS 2012. He is the recipient of 2015 ACM China Doctoral Dissertation Nomination Award. He is a senior member of the CCF.



Jie Wu (F'09) is the Director of the Center for Networked Computing and Laura H. Carnell professor at Temple University. He also serves as the Director of International Affairs at College of Science and Technology. He served as Chair of Department of Computer and Information Sciences from the summer of 2009 to the summer of 2016 and Associate Vice Provost for International Affairs from the fall of 2015 to the summer of 2017. Prior to joining Temple University, he was a program director at the National Science Foundation and was a distinguished professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Mobile Computing, IEEE Transactions on Service Computing, Journal of Parallel and decentralized Computing, and Journal of Computer Science and Technology. Dr. Wu was general co-chair for IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, ACM MobiHoc 2014, ICPP 2016, and IEEE CNS 2016, as well as program co-chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on decentralized Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.

Bin Luo is a full professor with the Software Institute, Nanjing University. He is also a member of the State Key Laboratory for Novel Software Technology. His main research interests include cloud computing, computer network, decentralized computing and edge computing, services computing, natural language processing and intelligent software engineering, machin learning and deep learning. His research results have been published in more than 90 papers in international journals and conference proceedings including IEEE TPDS, IEEE TMC, ACM TKDD, IEEE TSC, COMNET, JPDC, FGCS, JSS, Inf. Sci., JNCA, ESA, ExpSys, EMNLP, GlobeCom etc.