Resource Allocation Based on Deep Reinforcement Learning in IoT Edge Computing

Article *in* IEEE Journal on Selected Areas in Communications · April 2020

DOI: 10.1109/JSAC.2020.2986615

4 authors:

Xiong Xiong
Beijing University of Posts and Telecommunications
19 PUBLICATIONS **1,401** CITATIONS
SEE PROFILE

Kan Zheng
Beijing University of Posts and Telecommunications
375 PUBLICATIONS **12,052** CITATIONS
SEE PROFILE

Lei Lei
University of Guelph
139 PUBLICATIONS **6,674** CITATIONS
SEE PROFILE

Hou Lu
Beijing University of Posts and Telecommunications
38 PUBLICATIONS **1,560** CITATIONS
SEE PROFILE

Resource Allocation Based on Deep Reinforcement Learning in IoT Edge Computing

Xiong Xiong, Kan Zheng, *Senior Member, IEEE*, Lei Lei *Senior Member, IEEE*, and Lu Hou

***Abstract*—By leveraging mobile edge computing (MEC), a huge amount of data generated by Internet of Things (IoT) devices can be processed and analyzed at the network edge. However, the MEC system usually only has the limited virtual resources, which are shared and competed by IoT edge applications. Thus, we propose a resource allocation policy for the IoT edge computing system to improve the efficiency of resource utilization. The objective of the proposed policy is to minimize the long-term weighted sum of average completion time of jobs and average number of requested resources. The optimization problem is formulated as a Markov decision process (MDP). A deep reinforcement learning approach is applied to solve the problem. We also propose an improved deep Q-network (DQN) algorithm to learn the policy, where multiple replay memories are applied to separately store the experiences with small mutual influence. Simulation results show that the proposed algorithm has a better convergence performance than the original DQN algorithm, and the corresponding policy outperforms the other reference policies by lower completion time with fewer requested resources.**

***Index Terms*—Internet of Things (IoT); Mobile edge computing (MEC); Markov decision process (MDP); Reinforcement learning.**

I. Introduction

Mobile edge computing (MEC) is an emerging technology that provides cloud-computing capabilities within the radio access network (RAN) in close proximity to terminal devices. It enables a wide variety of applications and services to run at the edge of the mobile networks, which not only considerably reduces service latency, but also alleviates congestion in other parts of mobile core networks. The presence of MEC system at the edge of the RAN also allows applications to exploit local content and real-time information about local-access network, which can be used to offer content-related and location-awareness services. As such, the applications and services deployed on the MEC system can offer the good quality of user experience [1]–[3].

Internet of Things (IoT) applications can benefit greatly from the MEC technology. The IoT devices are normally constrained in terms of computation and storage capabilities. They have to upload sensor data to the remote server for further processing or storage [4], [5]. The huge amount of traffic generated by IoT devices may congest mobile networks, and processing the sensor data in the remote IoT cloud may not ensure the latency requirement of some IoT applications [6]. With the aid of MEC system, the sensor data can be processed and analyzed immediately at the edge of the mobile network instead of forwarded directly to the remote IoT cloud, which not only facilitates low latency, but also eases security and backhaul impacts [7]–[11].

The MEC system is built based on the virtualization technology that has been widely used in network functions virtualization (NFV) [12]. A MEC framework consists of several general entities, especially involving mobile edge host, mobile edge host level management, etc [13]. The mobile edge host level management offers functionality for managing the mobile edge host and the applications running on it. The mobile edge host contains a mobile edge platform and a virtualization infrastructure which provides virtual compute, storage, and network resources. All the idle virtual resources are shared and competed by the mobile edge applications. Once an application requests some idle resources, the requested resources are associated to the application and become isolated to others. Benefiting from the flexibility of virtualization, the requested resources can also be adjusted on demand[14].

In this paper, we focus on the resource allocation problem in the MEC system for IoT applications. Compared with the IoT cloud, the MEC system is normally equipped with limited resources. Different IoT edge application may require different amounts of resources to ensure the quality of service [15]. One important challenge is how to coordinate the limited resources for each application to achieve high resource utilization. That is, each application has to decide whether to request more idle resources or release some of the requested resources to avoid the shortage or wasting. Moreover, there is also a challenge for each application of how to allocate the requested resources to ensure that more sensor data can be processed in the MEC system as soon as possible. Therefore, it is necessary to find an optimal policy to utilize the limited resources in a high efficient and reasonable way.

As a classic formulation of sequential decision making, Markov decision process (MDP) is a suitable theory for formulating this problem. To solve MDP problems, reinforcement learning provides a bunch of solution methods, such as dynamic programming, Monte Carlo methods, temporal-difference learning, etc [16]. More modern reinforcement learning methods apply nonlinear function approximation to solve the problem with large state and action spaces, e.g., deep Q-network (DQN) [17], trust region policy optimization (TRPO) [18], etc[16], [19]. Recently, many researches have applied reinforcement learning algorithms to solve resource

Corresponding author: Kan Zheng (e-mail: zkan@bupt.edu.cn).

Xiong Xiong, Lu Hou and Kan Zheng are with the Intelligent Computing and Communications (IC^2) Lab, Wireless Signal Processing and Networks Lab (WSPN), Key Lab of Universal Wireless Communications, Ministry of Education, Beijing University of Posts and Telecommunications, Beijing, China, 100088. e-mail: zkan@bupt.edu.cn.

Lei Lei is with the School of Engineering, University of Guelph, Guelph, ON N1G 2W1, Canada.

allocation problems in either MEC or IoT systems [20]–[26]. For example, the dynamic programming methods are applied to solve the resource allocation problem in fog nodes for IoT applications with the objective of maximizing the average total serve utility and minimizing the idle time [27]. In [28], a reinforcement learning based method is used to jointly optimize the offloading decision and computational resource allocation with the objective of minimizing the sum of delay and energy consumptions. The same problem is also studied in [29], where a Q-learning based resource allocation is proposed for the offloading decision problem in the IoT edge network. In [30], the authors propose a DQN-based strategic computation offloading algorithm for MEC environment to minimize the long-term weighted sum of execution delay, the handover delay and the computation task dropping cost. Also, a DQN-based task offloading and resource allocation is proposed to minimize the offloading cost including the energy, computation, and delay cost [31]. In [32], the authors propose a resource allocation strategy for orchestrating the networking, caching and computing resources in vehicular network, which is obtained by the double dueling DQN algorithm. In [33], the joint computation offloading and multi-user scheduling problem is formulated as an infinite-horizon average-reward continuous-time MDP model, and solved by a deep reinforcement learning method to minimize the long-term average weighted sum of delay and power consumption. An actor-critic reinforcement learning method is applied to solve the joint caching, computing, and radio resource control problem in fog-enabled IoT with the objective of minimizing the average end-to-end delay [34].

In this paper, we propose a resource allocation policy for the IoT edge computing system to minimize the long-term weighted sum of average completion time of jobs and average number of requested resources, which has been seldom investigated in the related works. The optimization problem is formulated as a MDP model, which is solved by an improved DQN algorithm. The main contribution of this paper are summarized as follows:

- We improve the DQN algorithm by applying multiple replay memories to separately store the experiences with small mutual influence, which can further improve the training process of Q-networks.
- A new mechanism is designed in the MDP formulation, where the decision epoch is decoupled from the real timestep. By this mechanism, the actions for scheduling jobs and adjusting resources can be separated into two subspaces, which can reduce the size of action space.
- We also design a new architecture of the Q-network for our proposed algorithm, where a filter layer is added at the end of network to filter out the state-action values of invalid actions.

The rest of the paper is organized as follows. Section II introduces the system model and the MDP formulation. In Section III, the improved DQN algorithm is proposed as the solution of the MDP problem. In Section IV, the performances of the proposed algorithm and the corresponding policy are evaluated by simulation, which are compared with the original DQN algorithm and the other reference policies. The conclusions are drawn in Section V.

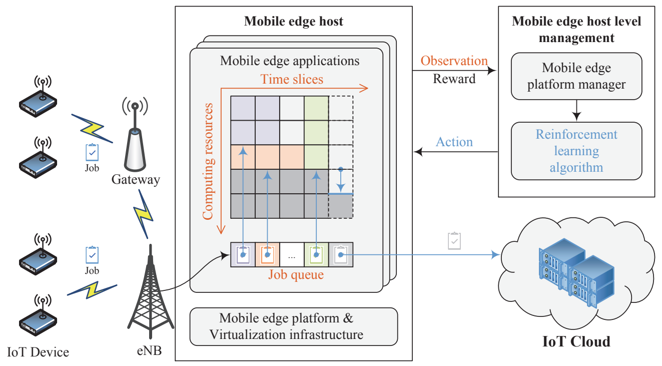

Mobile edge host
Mobile edge applications
Time slices
Computing resources
Job queue
Mobile edge platform & Virtualization infrastructure
Observation
Reward
Action
Mobile edge host level management
Mobile edge platform manager
Reinforcement learning algorithm
IoT Cloud
Job
Job
Gateway
eNB
IoT Device

Fig. 1: Illustration of the system model.

II. System Model and Problem Formulation

As illustrated in Fig. 1, we consider a MEC system deployed at the base station in a single-cell cellular network. In its serving area, a mass of IoT devices are randomly distributed, and upload sensor data to the network over machine-to-machine (M2M) connectivity. After arriving at the MEC system, the sensor data are queued and waiting to be scheduled.

The mobile edge host level management monitors the status information of IoT edge applications, including their resource allocation and the jobs waiting in the queue. In the MEC context, a job is considered as the encapsulation of the sensor data sent from an IoT device. Based on the monitored information, the mobile edge host level management can decide how to schedule each job, e.g., allocating virtual resources to the job for local processing, or forwarding the job to the remote IoT cloud for further processing. Moreover, it also makes decision on whether to adjust the requested virtual resources (i.e., request more resources, or release some of the requested resources) for each application to minimize the on-demand risk of shortage or plethora of virtual resources.

For simplicity, we consider only one mobile edge application deployed on the MEC system, and only focus on the allocation of computing resources. Due to massive data generated by the IoT application, one objective is to complete more jobs in the MEC system as soon as possible, not only to reduce the data transfer in the backhaul, but also to ensure low latency performance. Meanwhile, processing jobs with fewer requested computing resources is another objective, which can improve the utilization rate of limited resources in the MEC system. To achieve these goals, we formulate the problem as a MDP. The mobile edge host is considered as the **environment**, and the mobile edge host level management plays the role of the **agent** which performs decisions continually to interact with the environment. The environment of a MDP model contains several key elements, including state, action and reward. For our problem, the status information of the application is adopted as the state. The action refers to scheduling the jobs or adjusting the computing resources, which is taken by the agent based on current state. The environment responds to the action and steps into a new

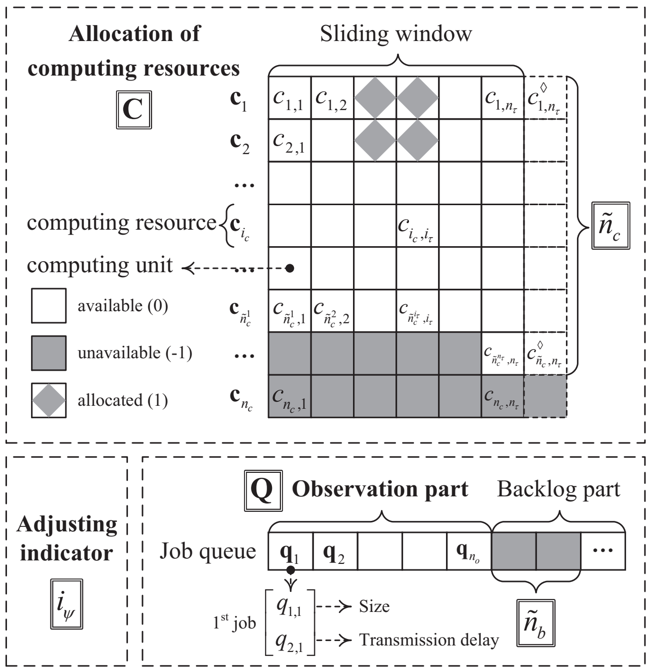


Fig. 2: Illustration of the state $s = (\mathbf{C}, \mathbf{Q}, i_\psi, \tilde{n}_c, \tilde{n}_b)$. $\mathbf{C}$: allocation of computing resources to jobs; $\mathbf{Q}$: observation part of job queue; i_ψ: adjusting indicator; $\tilde{n}_c$: number of computing resources requested in the next timestep; $\tilde{n}_b$: number of jobs in the backlog part of job queue.

state along with a reward signal, which is designed to coincide with the optimization objective. The details of the environment are introduced in the following subsections respectively.

A. State

The state $s \in \mathcal{S}$ describes the status information of the mobile edge application deployed on the MEC system, which can be given as

$$\mathcal{S} = \{s \mid s = (\mathbf{C}, \mathbf{Q}, i_\psi, \tilde{n}_c, \tilde{n}_b)\}, \quad (1)$$

where $\mathbf{C}$ denotes the allocation of computing resources to jobs, $\mathbf{Q}$ signifies the state of the observation part of job queue, i_ψ is the value of adjusting indicator, $\tilde{n}_c$ is the number of computing resources that is requested by the mobile edge application in the next timestep, and $\tilde{n}_b$ denotes the number of jobs in the backlog part of job queue. As shown in Fig. 2, each part of s is elaborated as below.

The allocation of computing resources $\mathbf{C}$ is formulated as a $n_c \times n_\tau$ matrix, where n_c is the number of total computing resources on the mobile edge host and n_τ denotes the number of time slices in the sliding window. That is, each row of $\mathbf{C}$ represents a **computing resource** $\mathbf{c}_{i_c}$, $i_c \in \{1, 2, \ldots, n_c\}$, which is scheduled for allocation starting from current time slice and looking ahead n_τ time slices into the future. The duration of one **time slice** is called a **time unit**, which is the same as the duration of one **timestep**. The sliding window moves on by one time slice as one timestep goes on. The specific value of a time unit depends on the practical

TABLE I: Summary of Main Notations

Symbol	Description
State II-A	
$\mathcal{S}$	the state space
s	the state
$\mathbf{C}$	the allocation of computing resources with the shape of $n_c \times n_\tau$
c_{i_c,i_τ}	the i_c-th computing unit at the i_τ-th time slice
n_c	the number of total computing resources on mobile edge host
n_τ	the number of time slices in the sliding window
$\tilde{n}_c$	the number of computing resources that will be requested in the next timestep
$\tilde{n}_c^{i_\tau}$	the number of computing resources that have been requested in the i_τ-th time slot
$\mathbf{Q}$	the state of the observation part of job queue with the shape of $2 \times n_q$
$\mathbf{q}_{i_q}$	the information of the i_q-th job, $\mathbf{q}_{i_q} = \left[q_{1,i_q},\ q_{2,i_q}\right]^\intercal$
q_{1,i_q}	the job size of the i_q-th job
q_{2,i_q}	the transmission delay of the i_q-th job
$\tilde{n}_q$	the number of jobs in the job queue
$\tilde{n}_b$	the number of jobs in the backlog part of job queue
n_o	the max length of the observation part of job queue
n_b	the max length of the backlog part of job queue
n_l	the max size of job
i_ψ	the value of adjusting indicator
n_ψ	the max value of adjusting indicator
Action II-B	
$\mathcal{A}$	the action space, $\mathcal{A} = \mathcal{A}_c \cup \mathcal{A}_\psi$
a	the action
$\mathcal{A}_c$	the action subspace of scheduling actions,
$\mathcal{A}_c(s)$	the set of valid scheduling actions in state s, $\mathcal{A}_c(s) \subseteq \mathcal{A}_c$
$\mathcal{A}_\psi$	the action subspace of adjusting the number of computing resources requested in the next timestep
$\mathcal{A}(s)$	the set of valid actions in state s
(δ_τ, δ_c)	the action that allocates δ_c computing resources to the first job starting from the δ_τ-th time slice in the sliding window
$(-1, \varnothing)$	the action that does not allocate any computing units to the job
$(-2, \delta_\psi)$	the action that adjusts $\tilde{n}_c$ to δ_ψ
State transition II-C	
t	the current decision epoch
τ	the current timestep
$\diamond$	a symbol with the superscript "$\diamond$" denotes the value in the next timestep
$'$	a symbol with the superscript " $'$ " denotes the value at the next decision epoch
Reward II-D	
$\hat{r}_c(s,a)$	the completion time of job scheduled by the action a
$\hat{r}_\psi(s,a)$	the total number of computing units requested by the action a before the next timestep for adjusting
$r(s,a)$	the reward function
β	the weight coefficient with the range from 0 to 1
$G_{c(t)}$	the sum of completion time of all jobs
$G_{\psi(t)}$	the sum of total requested computing units
$G_{(t)}$	the return from the decision epoch t, $G_{(t)} = -(1-\beta)G_{c(t)} - \beta G_{\psi(t)}$
Solution III	
$\pi(a\|s)$	a policy
$Q(s,a;\theta)$	the prediction network
$Q(s,a;\theta^-)$	the target network
$*$	a symbol with the superscript $*$ denotes the optimal value, e.g., optimal policy π^*, optimal action-value funtion $Q^*(s,a)$, etc.
γ	the discount rate
α	the learning rate
e	an experience $e = (s,a,r,s')$
D_{i_m}	the i_m-th replay memory, $i_m \in \{0,1,\ldots,n_c\}$
$\mathrm{L}(\theta)$	the loss function
ε_{i_m}	the parameter of ϵ-greedy method for the situation where the replay memory D_{i_m} is used
$\tilde{D}$	a batch of experiences sampled from one of D_{i_m}

application, e.g., tens of milliseconds. Each element of $\mathbf{C}$ defined as the **computing unit** indicates a computing resource during one time slice, which is the smallest unit of resource allocation. For a certain computing unit c_{i_c,i_τ}, it indicates the allocation state of the i_c-th computing resource at the i_τ-th time slice, where $i_\tau \in \{1, 2, \ldots, n_\tau\}$. Moreover, the value of computing unit c_{i_c,i_τ} is one of the possibilities in the set of $\{-1, 0, 1\}$, which represents three different states as follows:

- **unavailable** ($c_{i_c,i_\tau} = -1$): the computing unit can not be allocated to any jobs, since it has not been requested;
- **available** ($c_{i_c,i_\tau} = 0$): the computing unit is requested by the mobile edge application but has not been allocated;
- **allocated** ($c_{i_c,i_\tau} = 1$): the computing unit is allocated to a job.

Initially, some of the total n_c computing resources are requested by the application, and all of them are in the available state. Then, the requested computing units are allocated on demand to the jobs waiting in the queue. After being allocated, the state of the computing units is changed to "**allocated**", i.e., $[c_{i_c,i_\tau} = 0] \rightarrow [c_{i_c,i_\tau} = 1]$.

The number of requested computing resources should be adjusted according to the shift of demand by jobs. Considering the startup overhead, the computing resources can be only adjusted periodically in specified timesteps with the fix interval n_ψ instead of every timestep, i.e., the adjusting action has a larger period than the scheduling action. Therefore, an additional value $\tilde{n}_c \in \{1, 2, \ldots, n_c\}$ is required to record how many available computing resources will be requested at the n_τ-th time slice in the next timestep. In other words, when the system goes into the next timestep, the sliding window of computing resources moves on by one time slice, and the new number of available computing units at the last column of $\mathbf{C}$ is indicated by $\tilde{n}_c$. For simplicity, we assume that the $\tilde{n}_c$ computing units with the lowest indices at the newly coming time slice are available, while the others are unavailable, i.e., $c^{\diamond}_{i,n_\tau} = 0, \forall i \in \{1, \ldots, \tilde{n}_c\}$, and $c^{\diamond}_{i,n_\tau} = -1, \forall i \in \{\tilde{n}_c + 1, \ldots, n_c\}$, where the notation with the superscript "$\diamond$" denotes the value in the next timestep. Moreover, due to the value of $\tilde{n}_c$ possibly adjusted, the number of computing resources requested at different time slice in the sliding window varies as time goes on. Thus, the notation $\tilde{n}_c^{i_\tau}$ is also introduced here to denote the number of computing units requested at i_τ-th time slice, where $i_\tau \in \{1, 2, \ldots, n_\tau\}$.

For the job queue, there are $\tilde{n}_q \in \{1, 2, \ldots\}$ jobs held and waiting for allocation at the start of a timestep, and all the $\tilde{n}_q$ jobs are scheduled until the next timestep (the details will be explained in Section II-B). That is, $\tilde{n}_q$ also indicates the arrival rate of jobs. To keep the state space small, we logically divide the job queue into two parts, i.e., the **observation part** that contains first n_o jobs, and the **backlog part** that contains the rest with the max length n_b. It is assumed that only the information of jobs in the observation part, denoted as $\mathbf{Q}$, can be fully monitored, regardless of how many jobs are actually in the queue. Intuitively, it is reasonable to only pay attention to the earlier-arriving jobs in order to avoid long waiting time [35]. Moreover, this assumption also facilitates the state representation to be applied as input to a neural network. Thus, the state of the observation part $\mathbf{Q}$ can be represented by a n_o-column matrix, i.e.,

$$\mathbf{Q} = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \ldots & \mathbf{q}_{n_o} \end{bmatrix}, \tag{2}$$

where $\mathbf{q}_{i_q}$ is a two dimension column vector that indicates the information of the i_q-th job, for $\forall i_q \in \{1, 2, \ldots, n_o\}$. Furthermore, $\mathbf{q}_{i_q}$ can be denoted as $\begin{bmatrix} q_{1,i_q}, & q_{2,i_q} \end{bmatrix}^\intercal$, where $^\intercal$ denotes matrix transposition, q_{1,i_q} signifies the **job size**, i.e., the number of computing units required to process the job, and q_{2,i_q} indicates the **transmission delay** which is the amount of time of the job transmitted from device to MEC system. For a certain mobile edge application, we assume that the sizes of all jobs are discrete random variables with the same distribution and the maximum value of n_l, and the transmission delays also obey another coincident distribution. Due to the diversity of mobile edge applications, the distributions of job size, transmission delay and arrival rate $\tilde{n}_q$ can be regarded as configuration to different scenarios.

It is worth noting that there are two particular cases for the state of the observation part $\mathbf{Q}$ according to different number of jobs in the queue $\tilde{n}_q$. When the actual number of jobs is less than the length of observation part (i.e., $\tilde{n}_q < n_o$), the last $n_o - \tilde{n}_q$ columns of $\mathbf{Q}$ are padded with dummy jobs that have zero size and delay, i.e., $\mathbf{q}_i = [0, 0]^\intercal$, $\forall i \in \{\tilde{n}_q + 1, \ldots, n_o\}$. While $\tilde{n}_q > n_o$, the jobs out of backlog part are discarded, and the other jobs beyond first n_o are recorded in the backlog part, which can be retrieved latter after some of the first n_o jobs scheduled. The number of jobs in the backlog part is indicated by $\tilde{n}_b$ as a part of the system state, which is subjected to $\tilde{n}_b = \min\left(\max\left(\tilde{n}_q - n_o, 0\right), n_b\right)$.

As the rest part of s, the value of adjusting indicator $i_\psi \in \{1, 2, \ldots, n_\psi\}$ counts the number of timesteps elapsing after the last time when adjusting $\tilde{n}_c$. The value of i_ψ increases by one per timestep, and i_ψ is reset to one if larger than n_ψ. When $i_\psi < n_\psi$, the decisions of how to schedule the jobs are made in these timesteps, e.g., how many available computing units are allocated to them. When $i_\psi = n_\psi$, the agent has a chance to adjust $\tilde{n}_c$ in this timestep. The details of i_ψ and $\tilde{n}_c$ are to be further explained in Section II-C.

B. Action

As introduced above, the action $a \in \mathcal{A}$ serves two functions for different purposes, i.e., one for specifying how to schedule the jobs in the observation part of queue; and the other for appointing how many computing resources should be requested in the future. However, combining these two functions in one action and scheduling all the jobs in queue may lead to a large action space, which may make the problem too complicated.

Therefore, a new mechanism is designed to reduce the size of action space. We first decouple the **decision epoch** t from the real **timestep** τ. There are more than one decision epochs to execute actions in each timestep. At each timestep, time is frozen to schedule each job in the queue (including both observation part and backlog part) from head to tail by sequence, until the last job in the queue is scheduled, or the number of requested computing resources is adjusted. For the sake of illustration, Fig. 3 gives an example of this mechanism,

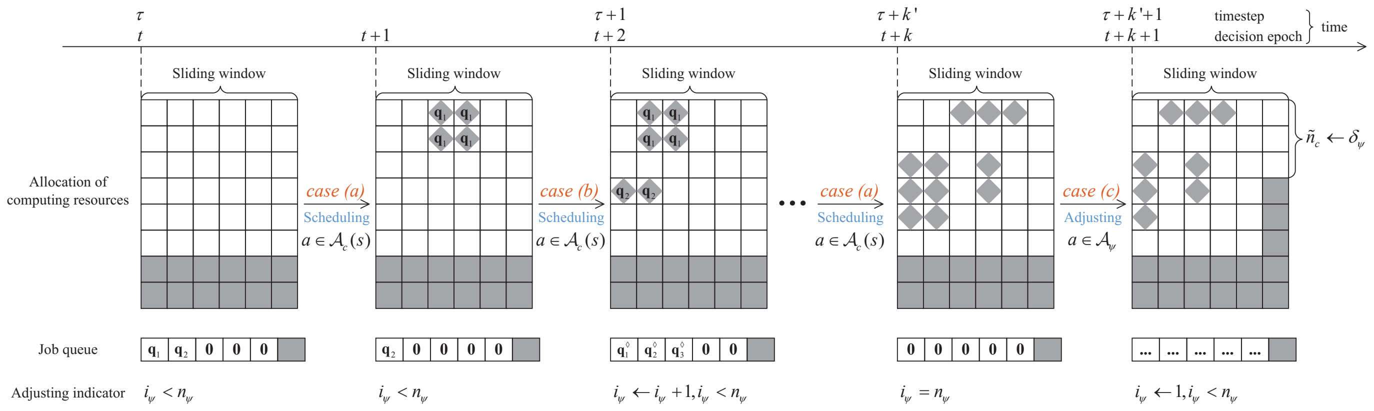


Fig. 3: Illustration of the mechanism where the decision epoch and timestep are decoupled, and the action space $\mathcal{A}$ are also divided into two subspace, i.e., $\mathbf{A}_c$ for scheduling jobs and $\mathbf{A}_\psi$ for adjusting requested computing resources. The timestep is frozen except when the last job in the queue is scheduled at the epoch $t+1$, and the number of requested computing resources is adjusted at the epoch $t+k$.

where the decision epoch and timestep are out of sync, and the timestep goes on only at the epoch $t+2$ and $t+k+1$. This mechanism will be further elucidated in the following Section II-C.

By this mechanism, multiple jobs can be scheduled at the same timestep, and the two functions are also decoupled into two corresponding action subspaces. Thus, the action space $\mathcal{A}$ are divided and given as

$$\mathcal{A} = \mathcal{A}_c \cup \mathcal{A}_\psi, \quad \mathcal{A}_c \cap \mathcal{A}_\psi = \emptyset, \tag{3}$$

where $\mathcal{A}_c$ consists of the actions of scheduling the first job $\mathbf{q}_1$ in queue, $\mathcal{A}_\psi$ includes the actions of adjusting the number of computing resources requested in the next timestep, and the notation $\emptyset$ stands for the empty set. As a consequence, each action is responsible for a unique task, either scheduling the first job $a \in \mathcal{A}_c$ (when $i_\psi < n_\psi$), or adjusting the number of requested computing resources $a \in \mathcal{A}_\psi$ (when $i_\psi = n_\psi$), which can keep the action space small. The details of scheduling action subspace $\mathcal{A}_c$ and adjusting action subspace $\mathcal{A}_\psi$ are described respectively as follows.

The scheduling action subspace $\mathcal{A}_c$ can be further denoted as

$$\mathcal{A}_c = \left\{ (\delta_\tau, \delta_c) \middle| \begin{matrix} \delta_\tau \in \{1, 2, \ldots, n_\tau\}, \\ \delta_c \in \{1, 2, \ldots, n_l\} \end{matrix} \right\} \cup \{(-1, \varnothing)\}, \tag{4}$$

where each action $a \in \mathcal{A}_c$ is described by a tuple, $(-1, \varnothing)$ means not allocating any computing unit to the first job which is forwarded directly to the remote IoT cloud for further processing, and (δ_τ, δ_c) denotes allocating δ_c computing resources to the first job $\mathbf{q}_1$ continuously from the δ_τ-th time slice in the sliding window until the job completion. In other words, the action $a = (\delta_\tau, \delta_c)$ is to find available computing units in $\mathbf{C}$ to place $\mathbf{q}_1$.

However, in a specific state s, not all of actions in $\mathcal{A}_c$ are valid due to some limitations. For example, for the job $\mathbf{q}_1 = [q_{1,1},\ q_{2,1}]^\intercal$, the action (δ_τ, n_l) is invalid when $q_{1,1} < n_l$. Thus, only a part of actions are valid in a state s, which make up a subset of $\mathcal{A}_c$ denoted by $\mathcal{A}_c(s) \subseteq \mathcal{A}_c$. For simplicity, the action $a = (\delta_\tau, \delta_c)$ are assumed as follows, i.e.,

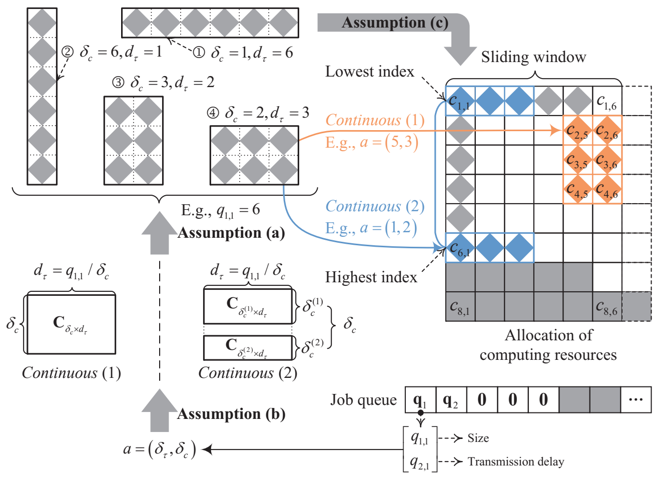

Assumption (c)
Sliding window
② $\delta_c = 6, d_\tau = 1$
① $\delta_c = 1, d_\tau = 6$
③ $\delta_c = 3, d_\tau = 2$
④ $\delta_c = 2, d_\tau = 3$
Lowest index
Continuous (1)
E.g., $a = (5,3)$
Continuous (2)
E.g., $a = (1,2)$
Highest index
E.g., $q_{1,1} = 6$
Assumption (a)
$d_\tau = q_{1,1} / \delta_c$
$\mathbf{C}_{\delta_c \times d_\tau}$
Continuous (1)
Continuous (2)
Assumption (b)
Allocation of computing resources
Job queue
$a = (\delta_\tau, \delta_c)$
Size
Transmission delay

Fig. 4: Illustration of the action $a = (\delta_\tau, \delta_c)$ with an example where $q_{1,1} = 6$, $n_c = 8$ and $n_\tau = 6$ (i.e., $\mathbf{C}$ is a 8×6 matrix).

(a) The size of the first job $q_{1,1}$ must be divisible by δ_c, i.e., $q_{1,1} \pmod{\delta_c} = 0$. Thus, δ_c is subjected to $\delta_c \in \{i \mid q_{1,1} \pmod{i} = 0, i \in \{1, \ldots, q_{1,1}\}\}$. Also, the number of time slices required to process the job can be obtained by $d_\tau = q_{1,1}/\delta_c$. For example, as shown in Fig. 4, when $q_{1,1} = 6$, $\delta_c \in \{1, 2, 3, 6\}$.

(b) Only the continuous δ_c available computing resources with the same starting time slice δ_τ can be allocated to the job $\mathbf{q}_1$. Here "continuous" means that the indices of computing resources are consecutive. For a specific $i_\tau \in \{1, \ldots, n_\tau\}$, all computing units from $\{c_{i,i_\tau} \mid i \in \{j, \ldots, j + \delta_c - 1\}\}$, $\forall j \in \{1, \ldots, \tilde{n}_c^{i_\tau} - \delta_c + 1\}$ are considered continuous, e.g., $\{c_{2,5}, c_{3,5}, c_{4,5}\}$ indicated by orange in Fig. 4. In addition, the requested computing resources with the lowest and highest indices are also considered continuous, i.e.,

$\{c_{i,i_\tau} \mid i \in \{1,\ldots,j+\delta_c-1-\tilde{n}_c^{i_\tau}\} \cup \{j,\ldots,\tilde{n}_c^{i_\tau}\}\}$, $\forall j \in \{\tilde{n}_c^{i_\tau}-\delta_c+2,\ldots,\tilde{n}_c^{i_\tau}\}$. For example, in Fig. 4, $\{c_{1,1}, c_{6,1}\}$ in blue are continuous.

(c) Despite of many potential continuous δ_c available computing resources as described in the assumption (b), only those of them with the lowest starting index j are adopted for allocation.

According to the above assumptions (a) and (b), the available computing units allocated by action $a = (\delta_\tau, \delta_c)$ to the job $\mathbf{q}_1$ must fully satisfy its resource requirement $q_{1,1}$, which should be a $\delta_c \times d_\tau$ submatrix of $\mathbf{C}$ (denoted as $\mathbf{C}_{\delta_c \times d_\tau}$), or two decomposed submatrixes $\mathbf{C}_{\delta_c^{(1)} \times d_\tau}$ and $\mathbf{C}_{\delta_c^{(2)} \times d_\tau}$, where $\delta_c^{(1)} + \delta_c^{(2)} = \delta_c$. Moreover, the action $a = (\delta_\tau, \delta_c)$ can uniquely identify the available computing units for the job $\mathbf{q}_1$ according to the assumption (c), as shown in Fig. 4, when $a = (5,3)$, $\{c_{i,j} \mid \forall j \in \{5,6\}, \forall i \in \{2,3,4\}\}$ are allocated to the job instead of $\{c_{i,j} \mid \forall j \in \{5,6\}, \forall i \in \{3,4,5\}\}$ or others. With all the above assumptions, the set of valid scheduling actions $\mathcal{A}_c(s)$ can be derived as

$$\begin{aligned}
&\mathcal{A}_c(s) = \{(\delta_\tau, \delta_c)\} \cup \{(-1, \varnothing)\}, \\
s.t.\ &\forall \delta_\tau \in \{1,\ldots,n_\tau\}, \\
&\forall \delta_c \in \{i \mid q_{1,1} \pmod{i} = 0, i \in \{1,\ldots,q_{1,1}\}\}, \\
&\exists \mathbf{C}_{\delta_c \times d_\tau} = \mathbf{0} \vee \left(\exists \mathbf{C}_{\delta_c^{(1)} \times d_\tau} = \mathbf{0} \wedge \exists \mathbf{C}_{\delta_c^{(2)} \times d_\tau} = \mathbf{0}\right), \\
&\delta_c^{(1)} = j + \delta_c - 1 - \tilde{n}_c^{\delta_\tau}, \\
&\delta_c^{(2)} = \tilde{n}_c^{\delta_\tau} - j + 1, \\
&\forall k \in \{1,\ldots,\tilde{n}_c^{\delta_\tau} - \delta_c + 1\} \text{ (used by } \mathbf{C}_{\delta_c \times d_\tau}), \\
&\forall j \in \{\tilde{n}_c^{\delta_\tau} - \delta_c + 2,\ldots,\tilde{n}_c^{\delta_\tau}\},
\end{aligned} \tag{5}$$

where $\mathbf{0}$ denotes the zero matrix with the same shape as that on the left side of the equation, and the submatrix $\mathbf{C}_{\delta_c \times d_\tau}$ can be further denoted by

$$\mathbf{C}_{\delta_c \times d_\tau} = \begin{bmatrix} c_{k,\delta_\tau} & \cdots & c_{k,\delta_\tau+d_\tau-1} \\ \vdots & \ddots & \vdots \\ c_{k+\delta_c-1,\delta_\tau} & \cdots & c_{k+\delta_c-1,\delta_\tau+d_\tau-1} \end{bmatrix}. \tag{6}$$

Similarly, the submatrixes $\mathbf{C}_{\delta_c^{(1)} \times d_\tau}$ and $\mathbf{C}_{\delta_c^{(2)} \times d_\tau}$ are given as

$$\mathbf{C}_{\delta_c^{(1)} \times d_\tau} = \begin{bmatrix} c_{1,\delta_\tau} & \cdots & c_{1,\delta_\tau+d_\tau-1} \\ \vdots & \ddots & \vdots \\ c_{\delta_c^{(1)},\delta_\tau} & \cdots & c_{\delta_c^{(1)},\delta_\tau+d_\tau-1} \end{bmatrix}, \tag{7a}$$

$$\mathbf{C}_{\delta_c^{(2)} \times d_\tau} = \begin{bmatrix} c_{j,\delta_\tau} & \cdots & c_{j,\delta_\tau+d_\tau-1} \\ \vdots & \ddots & \vdots \\ c_{\tilde{n}_c^{\delta_\tau},\delta_\tau} & \cdots & c_{\tilde{n}_c^{\delta_\tau},\delta_\tau+d_\tau-1} \end{bmatrix}. \tag{7b}$$

For example, the set of valid scheduling actions $\mathcal{A}_c(s)$ in Fig. 4 is equal to $\{(1,1),(1,2)\} \cup \{(2,2),(2,3),(2,6)\} \cup \{(3,2),(3,3),(3,6)\} \cup \{(4,2),(4,3),(5,3),(6,6)\}$ according to (5).

On the other hand, each action $a \in \mathcal{A}_\psi$ is also described as a tuple for uniformity, and the adjusting action subspace $\mathcal{A}_\psi$ is given as

$$\mathcal{A}_\psi = \{(-2, \delta_\psi) \mid \delta_\psi \in \{1,2,\ldots,n_c\}\}, \tag{8}$$

where δ_ψ denotes the new value of $\tilde{n}_c$ after adjusting.

C. State transition

TABLE II: State transition cases.

	$\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 \neq \mathbf{0}$	$\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 = \mathbf{0}$	$\mathbf{q}_1 = \mathbf{0}$
$i_\psi < n_\psi$	*case* (a)	*case* (b)	–
$i_\psi = n_\psi$	*case* (a)	*case* (a)	*case* (c)

The state transition is a function of a pair of state and action (s, a). For a certain state $s = (\mathbf{C}, \mathbf{Q}, i_\psi, \tilde{n}_c, \tilde{n}_b)$ at the decision epoch t, an action is taken either from $\mathcal{A}_c(s)$ or $\mathcal{A}_\psi$ to make the state move to a successor state $s' = \left(\mathbf{C}', \mathbf{Q}', i'_\psi, \tilde{n}'_c, \tilde{n}'_b\right)$, i.e., $s \xrightarrow{a} s'$. Different from "$\diamond$", the notation with the superscript "$\,'\,$" denotes the value at the next decision epoch. There are several different cases of state transition according to the state s as shown in Table II. Each case of the state transition is explained in detail as follows.

(a) When $i_\psi \leq n_\psi$ and there are more than one jobs waiting in the queue (i.e., $\mathbf{q}_1 \neq \mathbf{0}$ and $\mathbf{q}_2 \neq \mathbf{0}$), or when $i_\psi = n_\psi$ and $\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 = \mathbf{0}$, an action is selected from $\mathcal{A}_c(s)$ to schedule the first job $\mathbf{q}_1$, which should be either $a = (\delta_\tau, \delta_c)$ or $a = (-1, \varnothing)$. If $a = (\delta_\tau, \delta_c)$, the available computing units described by $\mathbf{C}_{\delta_c \times d_\tau}$, or $\mathbf{C}_{\delta_c^{(1)} \times d_\tau}$ and $\mathbf{C}_{\delta_c^{(2)} \times d_\tau}$, are allocated to the job $\mathbf{q}_1$. After allocation, these computing units in $\mathbf{C}$ should change to the allocated state in the next state s', thereby $\mathbf{C}' \neq \mathbf{C}$. If $a = (-1, \varnothing)$, it just forwards the job to the remote IoT cloud without allocating any computing unit, which has nothing to do with $\mathbf{C}$ (i.e., $\mathbf{C}' = \mathbf{C}$). Meanwhile, after scheduling the job, the first job $\mathbf{q}_1$ is dequeued. Thus, the new state of $\mathbf{Q}'$ is derived by rolling one column of $\mathbf{Q}$ along the row, i.e., $\mathbf{q}'_i = \mathbf{q}_{i+1}$ for $\forall i \in \{1,\ldots,n_o-1\}$, and the tail of observation part $\mathbf{q}'_{n_o}$ is assigned by the first job loaded from the backlog part if $\tilde{n}_b > 0$. Otherwise, the job queue is padded with a dummy job $[0,0]^\intercal$ (also denoted as $\mathbf{0}$). Moreover, $\tilde{n}'_b = \max(\tilde{n}_b - 1, 0)$, and the rest parts of s' are the same as before $i'_\psi = i_\psi$, $\tilde{n}'_c = \tilde{n}_c$. Till now, all parts of the state are updated, and then the system moves to the next decision epoch (i.e., $t' = t+1$) with the new state s'. In this case, the decision epoch passes away after state transition, while the timestep is frozen (i.e., $\tau' = \tau$). For example, as shown in Fig. 3, the state transitions at the decision epoch t and $t+2$ belongs to this case.

(b) When $i_\psi < n_\psi$ and there is only one job waiting in the queue (i.e., $\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_i = \mathbf{0}$ for $\forall i \in \{2,\ldots,\tilde{n}_o\}$), the state transition from s to s' firstly follows *case* (a). Besides, there are some additional operations. After the only job scheduled, the job queue is empty, and the system steps into not only the next decision epoch (i.e., $t' = t+1$) but also a new timestep (i.e., $\tau' = \tau + 1$). Thus, the adjusting indicator increases by one timestep, i.e., $i'_\psi = i_\psi + 1$. In the new timestep, $\tilde{n}'_q$ new jobs arrive at the MEC system, which fill up the job queue and decide the value of $\mathbf{Q}'$ and $\tilde{n}_b'$. For example, the state transition at the epoch $t+1$ as shown in Fig. 3 is in this case.

(c) When $i_\psi = n_\psi$ and there is no job waiting in the queue (i.e., $\mathbf{q}_1 = \mathbf{0}$), it is the decision epoch to adjust $\tilde{n}_c$. In this case, the action $a = (-2, \delta_\psi)$ is chosen from $\mathcal{A}_\psi$, and the new value of $\tilde{n}_c$ is set to $\tilde{n}_c' = \delta_\psi$. After adjusting $\tilde{n}_c$, the system also steps into a new timestep similar to *case* (b), but the adjusting indicator is reset to one (i.e., $i_\psi' = 1$). Also, the newly arriving jobs are revealed in the queue state $\mathbf{Q}'$ and $\tilde{n}_b'$ in the next timestep. In Fig. 3, the state transition at the epoch $t+k$ is an example of this case.

The state transition repeats iteratively according to the cases mentioned above. Therefore, the state transition in the successor state can also be derived. When $i_\psi = n_\psi \wedge \mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 = \mathbf{0}$ in the state s, the state transition follows *case* (a). After then, the state transition in the successor state follows *case* (c), which is donated as $s \xrightarrow[\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 = \mathbf{0} \wedge i_\psi = n_\psi]{\textit{case (a)}} s' \xrightarrow{\textit{case (c)}} s''$. Similarly, the other state transition cases in the successor state are summarized as follows:

- $s \xrightarrow[\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 \neq \mathbf{0} \wedge \mathbf{q}_2' \neq \mathbf{0} \wedge i_\psi < n_\psi]{\textit{case (a)}} s' \xrightarrow{\textit{case (a)}} s''$,
- $s \xrightarrow[\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 \neq \mathbf{0} \wedge \mathbf{q}_2' = \mathbf{0} \wedge i_\psi < n_\psi]{\textit{case (a)}} s' \xrightarrow{\textit{case (b)}} s''$,
- $s \xrightarrow[\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 \neq \mathbf{0} \wedge i_\psi = n_\psi]{\textit{case (a)}} s' \xrightarrow{\textit{case (a)}} s''$,
- $s \xrightarrow[\mathbf{q}_1 \neq \mathbf{0} \wedge \mathbf{q}_2 = \mathbf{0} \wedge i_\psi < n_\psi]{\textit{case (b)}} s' \xrightarrow{\textit{case (a)}} s''$,
- $s \xrightarrow[\mathbf{q}_1 = \mathbf{0} \wedge i_\psi = n_\psi]{\textit{case (c)}} s' \xrightarrow{\textit{case (a)}} s''$.

Therefore, combining these state transition cases and formulas (5) and (8), the valid action space of a certain state s can be given as

$$\mathcal{A}(s) = \begin{cases} \mathcal{A}_c(s), & \mathbf{q}_1 \neq \mathbf{0} \\ \mathcal{A}_\psi, & \mathbf{q}_1 = \mathbf{0} \end{cases}. \tag{9}$$

Due to the complexity of our problem, it is hard to directly derive the state-transition probability for each state $p(s'|s,a)$. However, without $p(s'|s,a)$, the problem can still be solved by model-free reinforcement learning algorithms, which is further explained in Section III.

D. Reward

A reward signal is designed to coincide with the goal of an optimization problem. In this paper, the goal is to complete each job arriving at the MEC system as soon as possible with the least computing resources requested. In another word, the objective is defined as minimizing the weighted sum of the average completion time of jobs and the average number of requested computing units. By decomposing the objective into small pieces at each decision epoch, the reward $r(s,a)$ can be also constructed as a function of state s and action a, which is generated along with each state transition $s \xrightarrow{a, r(s,a)} s'$.

As described in Section II-C, for a given state s where $\mathbf{q}_1 \neq \mathbf{0}$, the state transition follows *case* (a) or (b), and an action $a \in \mathcal{A}_c(s)$ is applied to schedule the job $\mathbf{q}_1$. Then, the completion time of job $\hat{r}_c(s,a)$ can be calculated by

$$\hat{r}_c(s,a) = \begin{cases} q_{2,1} + \delta_\tau + q_{1,1}/\delta_c, & a = (\delta_\tau, \delta_c) \wedge \mathbf{q}_1 \neq \mathbf{0} \\ q_{2,1} + d_0, & a = (-1, \varnothing) \wedge \mathbf{q}_1 \neq \mathbf{0} \end{cases}, \tag{10}$$

where d_0 denotes the total time spent on sending the job from the MEC system to the IoT cloud, and processing it after then. For simplicity, it is assumed that d_0 is a constant and d_0 is much greater than the span of the sliding window n_τ (i.e., $d_0 > n_\tau$). Thus, if $a = (-1, \varnothing)$, the completion time of job should be the sum of its transmission delay $q_{2,1}$ and d_0. If $a = (\delta_\tau, \delta_c)$, the completion time of job is calculated by adding up its transmission delay $q_{2,1}$ and its processing time $\delta_\tau + q_{1,1}/\delta_c$, which is derived by the number of timesteps elapsing till the computing units allocated at last time slot have been executed.

On the other hand, when $\mathbf{q}_1 = \mathbf{0}$, the state transition follows *case* (c). Based on the adopted action $a = (-2, \delta_\psi) \in \mathcal{A}_\psi$, it is easy to forecast the total number of requested computing units before the next timestep for adjusting, which is given by

$$\hat{r}_\psi(s,a) = \delta_\psi \cdot n_\psi, \quad \text{if } a = (-2, \psi) \wedge \mathbf{q}_1 = \mathbf{0}. \tag{11}$$

Typically, in the short term, the higher reward the state transition generates, the better the decision is made in that state [16]. Therefore, the reward $r(s,a)$ is further designed as

$$r(s,a) = \begin{cases} -(1-\beta) \cdot \hat{r}_c(s,a), & \mathbf{q}_1 \neq \mathbf{0} \\ -\beta \cdot \hat{r}_\psi(s,a) & \mathbf{q}_1 = \mathbf{0} \end{cases}, \tag{12}$$

where β is a weight coefficient between the completion time of job $\hat{r}_c(s,a)$ and the requested computing units $\hat{r}_\psi(s,a)$, $0 < \beta < 1$, and the negative sign is also introduced to perform the mathematical conversion.

According to (12), we can further obtain the return, which is defined as the cumulative sum of reward generated at each decision epoch over the long run [16]. The return $G_{(t)}$ from the decision epoch t is given as:

$$\begin{aligned} G_{(t)} &= \sum_{k=t}^{\infty} \gamma^{k-t} R_{(k+1)} = \sum_{k=t}^{\infty} \gamma^{k-t} r\left(S_{(k)}, A_{(k)}\right) \\ &= -(1-\beta) G_{c(t)} - \beta G_{\psi(t)}, \end{aligned} \tag{13a}$$

$$G_{c(t)} = \sum_{\substack{k=t, \\ \mathbf{q}_{1(k)} \neq \mathbf{0}}}^{\infty} \gamma^{k-t} \hat{r}_c\left(S_{(k)}, A_{(k)}\right), \tag{13b}$$

$$G_{\psi(t)} = \sum_{\substack{k=t, \\ \mathbf{q}_{1(k)} = \mathbf{0}}}^{\infty} \gamma^{k-t} \hat{r}_\psi\left(S_{(k)}, A_{(k)}\right), \tag{13c}$$

where γ, $0 < \gamma \leq 1$, is the discount rate, and $R_{(k)}$, $S_{(k)}$, $A_{(k)}$ denote the reward, state and action at the k-th decision epoch respectively. To differ from other subscripts, the decision epoch k is grouped by brackets. Accordingly, the symbol $\mathbf{q}_{1(k)}$ indicates the information of the first job in the state $S_{(k)}$. As shown in (13a), the return $G_{(t)}$ consists of two parts, i.e., $G_{c(t)}$ standing for the sum of completion time of all jobs, and $G_{\psi(t)}$ which is the sum of total requested computing units and also indicates the resource utilization. Therefore, to maximize

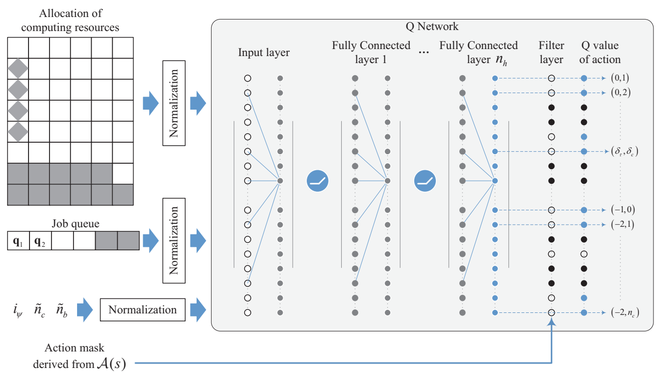

Allocation of computing resources
Job queue
$\mathbf{q}_1$ $\mathbf{q}_2$
i_ψ $\tilde{n}_c$ $\tilde{n}_b$
Normalization
Normalization
Normalization
Q Network
Input layer
Fully Connected layer 1
...
Fully Connected layer n_h
Filter layer
Q value of action
(0,1)
(0,2)
(δ_τ, δ_c)
(−1,0)
(−2,1)
$(-2, n_c)$
Action mask derived from $\mathcal{A}(s)$

Fig. 5: Illustration of the proposed Q-network, where a filter layer is added at the end of the network to filter out the state-action values of invalid actions.

the expected return $\mathbb{E}_\pi\left[G_{(t)}\right]$ is equivalent to the optimization objective, i.e., minimizing the weighted sum of the completion time of jobs and the number of requested computing resources in the long term.

III. Solution

The solution of the MDP problem is to find the optimal policy π^* that maximizes the expected return $\mathbb{E}_{\pi^*}\left[G_{(t)}\right]$. A policy π is a mapping from states to probabilities of selecting each possible action, which can be formally represented by a probability distribution $\pi(a|s)$ over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$. For any MDP, there exists an optimal policy π^* that is better than or equal to all other policies.

We improve the deep Q-network (DQN) algorithm by applying multiple replay memories to further improve the training process. The DQN algorithm evolves from Q-learning, which is a value-based method and involves an action-value function (also called Q-function). For a certain policy π, the action-value function $Q_\pi(s,a)$ is defined as

$$\begin{aligned} Q_\pi(s,a) =& \mathbb{E}_\pi\left[G_{(t)} \mid S_{(t)} = s, A_{(t)} = a\right] \\ & \mathbb{E}_\pi\left[\sum_{k=t}^{\infty} \gamma^{k-t} R_{(k+1)} \middle| S_{(t)} = s, A_{(t)} = a\right]. \end{aligned} \tag{14}$$

The optimal policy π^* always achieves the optimal action-value function $Q^*(s,a)$, which has the largest value for a specific state-action pair, i.e.,

$$Q^*(s,a) = Q_{\pi^*}(s,a) = \max_\pi Q_\pi(s,a). \tag{15}$$

Conversely, if the optimal action-value function $Q^*(s,a)$ is obtained, the optimal policy can be immediately found by maximizing over $Q^*(s,a)$, which is given by

$$\pi^*(a|s) = \arg\max_{a \in \mathcal{A}(s)} Q^*(s,a). \tag{16}$$

Therefore, finding the optimal policy π^* is equivalent to obtaining the optimal action-value function $Q^*(s,a)$. Due to the self consistency, the optimal action-value function can be written in a special form called Bellman optimality equation [16], which is given by

$$Q^*(s,a) = \mathbb{E}_\pi\left[R_{(t+1)} + \gamma \max_{a'} Q^*(s',a') \mid S_{(t)} = s, A_{(t)} = a\right], \tag{17}$$

where γ is the discount rate, $0 < \gamma < 1$. Based on (17), the optimal action-value function can be estimated by temporal-difference updates in an iterative manner, which is shown as

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right], \tag{18}$$

where $Q(s,a)$ denotes the learned action-value function, and α is called learning rate. After multiple iteration, $Q(s,a)$ can converge to $Q^*(s,a)$ independent of the policy being followed.

Different from Q-learning, the DQN uses an artificial neural network $Q(s,a;\theta)$, called prediction network, as a function approximator to estimate the action-value function, $Q(s,a;\theta) \approx Q^*(s,a)$, where θ is the weights of the neural network and the number of θ is much less than the number of states. The input of the prediction network is the state s, and the corresponding values of all possible actions are generated as the output. Moreover, another neural network $Q(s,a;\theta^-)$, called target network, is also used to estimate the target value in (17), i.e., $r(s,a) + \gamma \max_{a'} Q(s',a';\theta^-) \approx r(s,a) + \gamma \max_{a'} Q^*(s',a')$. The target network has the same structure as the prediction network. However, its weights θ^- are copied from θ every fixed number of iterations n_θ instead of every training epoch. Both the prediction network and the target network are called Q-network.

To learn $Q(s,a;\theta)$, a mean-squared error loss function $\mathrm{L}(\theta)$ is used and given by

$$\mathrm{L}(\theta) = \frac{1}{|\tilde{D}|} \sum_{e \in \tilde{D}} \left[\left(r(s,a) + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta)\right)^2\right], \tag{19}$$

where $e = (s, a, r(s,a), s')$ is an experience (also called sample) that represents a state transition with the reward, and $\tilde{D}$ is a batch of experiences. Minimizing $\mathrm{L}(\theta)$ is equivalent to reducing the mean-squared error in the Bellman equation, which can be achieved by updating weights θ using different algorithms, such as Momentum, RMSProp, Adam, etc [36]. Therefore, the DQN substitutes the temporal-different update (18) with the updating weights θ based on (19).

Different from the original DQN algorithm, we apply multiple replay memories $D_{i_m}, \forall i_m \in \{0, 1, \ldots, n_c\}$ in our modification to store experiences of different situations, respectively. The replay memory D_0 stores the samples where adjusting actions $a \in \mathcal{A}_\psi$ are taken, while the sample of scheduling $e = (s, a, r(s,a), s')$, $a \in \mathcal{A}_c(s)$ is pushed into the memory $D_{\tilde{n}_c}, \tilde{n}_c \in \{1, \ldots, n_c\}$ according to the number of requested computing resources $\tilde{n}_c$ of the current state $s = (\mathbf{C}, \mathbf{Q}, i_\psi, \tilde{n}_c, \tilde{n}_b)$. This design stems from our formulation of the problem that scheduling and adjusting actions are taken for two different tasks, and the experiences with different number of requested resources $\tilde{n}_c$ have little relation to each others. Thus, it can improve the training process of Q-networks.

The completed algorithm for training is presented in Algorithm 1. On each interaction with the environment, the agent selects the action according to a ε-greedy rule, i.e., selecting the greedy policy with probability $1-\varepsilon$ and a random action with probability ε, where $\varepsilon_{\min} \leq \varepsilon \leq 1$. It is worth noting that multiple $\varepsilon_{i_m}, \forall i_m \in \{0,1,\ldots,n_c\}$ are also utilized to achieve a better exploration in different situations. Each ε_{i_m} is associated with the situation, where the replay memory D_{i_m} is applied, and decays exponentially according to the factor $\varepsilon_{\text{decay}}$ by each iteration until the minimal value $\varepsilon_{\min}$. After the action taken, a state transition occurs, which is saved into the corresponding memory D_{i_m} as described above. Then, a batch of experiences $\tilde{D}$ used in (19) are randomly selected from the memory D_{i_m} for the purpose of training the Q-network.

In our implementation, a multi-layer neural network is used as illustrated in Fig. 5. Before sent into the network, the state s is preprocessed firstly by normalization, since each part of state $s = (\mathbf{C}, \mathbf{Q}, i_\psi, \tilde{n}_c, \tilde{n}_b)$ has different ranges of value. For example, the computing resources $\mathbf{C}$ is treated as an one-channel image, where each computing unit c_{i_c,i_τ} of $\mathbf{C}$ has the maximal value of 1 and the minimal value of -1, while the size of jobs q_{1,i_q} ranges from 1 to n_l. The normalization method for each part is given by

$$g(x) = \frac{x - x_{\text{mean}}}{x_{\max} - x_{\min}}, \tag{20}$$

where $x_{\text{mean}} = (x_{\max} + x_{\min})/2$. After normalization, all parts of the state are reshaped to 1-dimensional vectors by row, and concatenated together by the sequence $(\mathbf{C}, \mathbf{Q}, i_\psi, \tilde{n}_c, \tilde{n}_b)$. Then, the preprocessed state is sent into the input layer of the neural network which has $(n_c \cdot n_\tau + 2 \cdot n_o + 3)$ neurous, and propagates through n_h fully connected layers. The output of the last fully connected layer has the same size $(n_\tau \cdot n_l + n_c + 1)$ as the total action space $|\mathcal{A}|$, and each element of the output is mapping to an action-value function of action $a \in \mathcal{A}$. However, considering not all actions are valid in the state s, we hence add a filter layer at the end of the network to filter out the state-action values of invalid actions. This filter layer takes another input called action mask, which is a vector derived from valid actions $\mathcal{A}(s)$. The length of action mask is also $|\mathcal{A}|$, and i-th element of the action mask indicates whether the i-th action in $\mathcal{A}$ is valid. If the i-th action is invalid, the i-th element in the output of the Q-network is set to $-\infty$ (i.e., the minimum value of float type in the implementation). Otherwise, the filter layer has no effect on the output of valid actions. This filter layer does not affect the back propagation of neural network, since both the predict and target values of the invalid action are equal to $-\infty$ and reduce each other to zero when calculating the loss function (19). With this filter layer, the operation for getting greedy action $\arg\max_{a\in\mathcal{A}(s)} Q(s,a;\theta)$ can be easily realized.

IV. Simulation Results

To evaluate the performance of proposed model, we develop a python simulator running on Google TensorFlow-1.14 [1]. The main configuration of the simulation is listed in Table III.

[1]https://www.tensorflow.org/versions/r1.14/api_docs

Algorithm 1 deep Q-network with multiple replay memories

```
 1: Initialize prediction network $Q(s,a;\theta)$ with random weights $\theta$
 2: Initialize target network $Q(s,a;\theta^-)$ with weights $\theta^- = \theta$
 3: Initialize replay memories $D_{i_m}, i_m \in \{0,1,\ldots,n_c\}$ with the same capacity $|D|$
 4: Initialize $\varepsilon_{i_m} = 1, \forall i_m \in \{0,1,\ldots,n_c\}$
 5: Initialize state $s = (\mathbf{C}, \mathbf{Q}, i_\psi, \tilde{n}_c, \tilde{n}_b)$
 6: for $t \leftarrow 1,\ldots,\infty$ do
 7:     if $\mathbf{q}_1 = \mathbf{0}$ then                              ▷ adjusting
 8:         $\mathcal{A}(s) \leftarrow \mathcal{A}_\psi$                 ▷ according to (9)
 9:         $i_m \leftarrow 0$
10:     else                                                             ▷ scheduling
11:         $\mathcal{A}(s) \leftarrow \mathcal{A}_c(s)$                 ▷ according to (9) and (5)
12:         $i_m \leftarrow \tilde{n}_c$
13:     end if
14:     Generate random number $p$ from 0 to 1                           ▷ $\varepsilon-$greedy
15:     $a \leftarrow \begin{cases} \text{randomly selected from } \mathcal{A}(s) & 0 \leq p < \varepsilon_{i_m} \\ \arg\max_{a\in\mathcal{A}(s)} Q(s,a;\theta) & \varepsilon_{i_m} \leq p < 1 \end{cases}$
16:     Take action $a$ and perform state transition $s \xrightarrow{a,r(s,a)} s'$
17:     $e \leftarrow (s,a,r(s,a),s')$                                   ▷ according to II-C and (12)
18:     Save $e$ into replay memory $D_{i_m}$
19:     Random sample a batch of experiences $\tilde{D}$ from $D_{i_m}$
20:     Calculate $\mathrm{L}(\theta)$                                   ▷ according to (19)
21:     Update weights $\theta$
22:     if $\varepsilon_{i_m} > \varepsilon_{\min}$ then                 ▷ exponential decay of $\varepsilon_{i_m}$
23:         $\varepsilon_{i_m} \leftarrow \varepsilon_{i_m} \cdot \varepsilon_{\text{decay}}$
24:     end if
25:     if $t \mod n_\theta = 0$ then                                    ▷ every $n_\theta$ steps
26:         $\theta^- \leftarrow \theta$
27:     end if
28:     $s \leftarrow s'$
29: end for
```

In the experiments, a certain mobile edge application is assumed to be deployed on the MEC system, and the IoT devices of the application are scattered uniformly in the serving area of the network. Each packet of sensor data sent to the MEC system suffers from different delays due to various wireless transmissions. For simplify, the transmission delay of each packet is assumed to obey an uniform distribution between 1 and 4 time units. After arriving at the MEC system, each packet of sensor data is encapsulated as a job. The size of jobs follows a discrete uniform distribution with the range from 1 to $n_l = 4$. The overall jobs sent from IoT devices follows a Poisson distribution with the mean rate of $\lambda = 3$ [37], i.e., there are average 3 jobs per timestep arriving at the MEC system and waiting in the queue to be scheduled.

In the MEC system, there are $n_c = 10$ total computing resources on the mobile edge host, and the sliding window for allocating computing resources contains $n_\tau = 6$ time slices, i.e., the shape of $\mathbf{C}$ is 10×6. Both the observation part and the backlog part of the job queue have the same length of $n_o = n_b = 5$. The maximal value of adjusting indicator is $n_\psi = 12$, which is configured as the twice as much as the length of sliding window n_τ. For the reward function (12), the

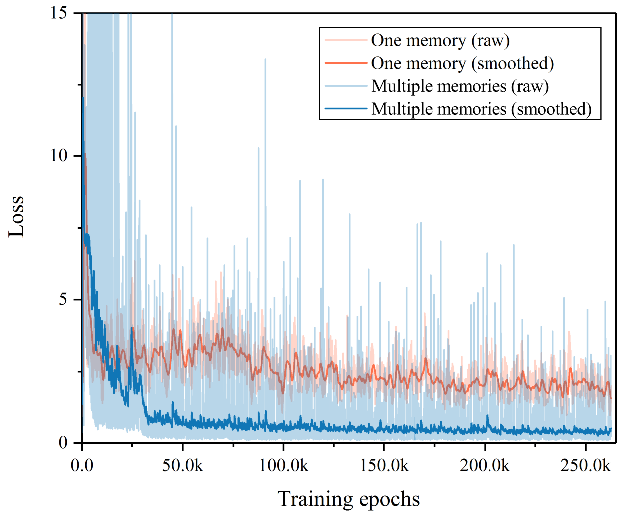
TABLE III: Experiment Configuration

Symbol	Description	Value
	Environment	
n_c	the number of total computing resources on mobile edge host	10
n_τ	the number of time slices in the sliding window	6
n_o	the max length of the observation part of job queue	5
n_b	the max length of the backlog part of job queue	5
n_l	the max size of job	4
n_ψ	the max value of adjusting indicator	12
d_0	the total time of transmission and process for the job forwarded to remote IoT cloud	20
β	the weight coefficient in (12)	0.5
	Agent	
γ	the discount rate	0.99
$\varepsilon_{\min}$	the minimal value of ε_{i_m}	0.01
$\varepsilon_{\text{decay}}$	the decay rate of ε_{i_m}	0.99
α	the learning rate	0.001
$\|D\|$	the capacity of each replay memory D_{i_m}	1200
$\|\tilde{D}\|$	the batch size for training	64
n_h	the number of fully connected layers	2
-	the number of neurons in each fully connected layer	64
n_θ	the interval of iterations to update target network θ^-	16

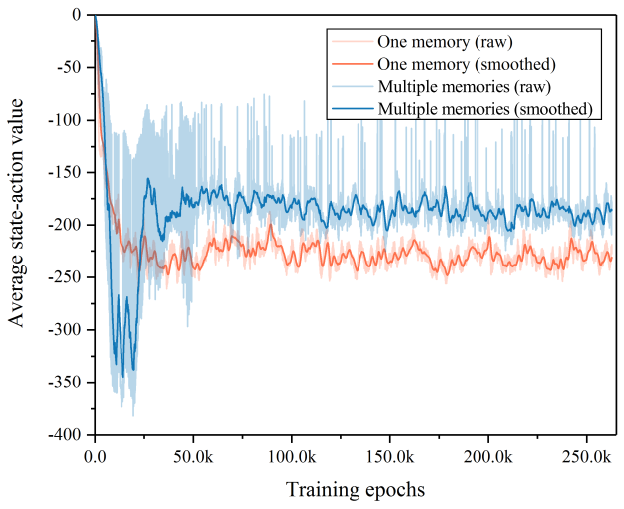
total time of transmission and process for the job forwarded to the remote IoT cloud d_0 is set to 20, which is large enough when compared to the time of processing the job locally in the MEC system. As we know, minimizing the completion time of jobs has the same importance as minimizing the number of requested computing units. Thus, the weight coefficient β is set to 0.5. Most of the environment parameters are summarized in Table III. It also gives the parameters used in Algorithm 1. There are $n_c + 1 = 11$ replay memories to store experiences, and the capacity of each replay memory is 1200. The parameters of ε-greedy rule are set as $\varepsilon_{\min} = 0.01$ and $\varepsilon_{\text{decay}} = 0.99$. The discount rate γ in (19) for estimating the target value is 0.99. The Q-network contains two fully connected layers. Each of them has 64 neurons, and applies the rectified linear units (ReLU) activation function. With the environment settings, the size of the input layer of the Q-network can be calculated by $(n_c \cdot n_\tau + 2 \cdot n_o + 3) = 73$. We can also obtain the size of the adjusting action subspace $\mathcal{A}_\psi$ according to (8), which is the same as the number of total computing resources $n_c = 10$. Moreover, based on the maximal size of job n_l, the size of the scheduling action subspace $\mathcal{A}_c$ is fixed to $(n_\tau \cdot n_l + 1) = 25$ according to (5). By combining these two subspaces together, the action space $\mathcal{A}$ has the size of 35, which is also the size of the output layer of the Q-network. The prediction network is updated by using the Adam algorithm with a learning rate of 0.001, and each update applies a batch of $|\tilde{D}| = 64$ samples. The weights of target network θ^- are copied from θ every $n_\theta = 16$ epochs.

A. Training Performances

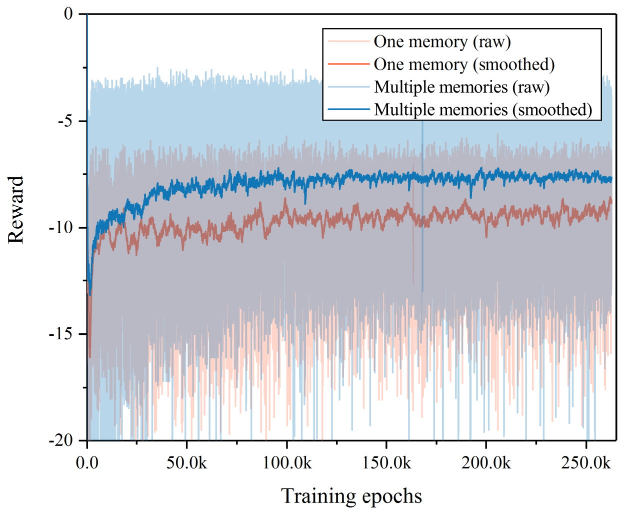
We first compare the training performance of the improved DQN algorithm with multiple replay memories (i.e., Algorithm



(a) Loss ($\text{L}(\theta)$) - Training epochs



(b) Average state-action value - Training epochs



(c) Reward ($r(s, a)$) - Training epochs

Fig. 6: Comparison of training performance between the improved DQN with multiple replay memories and the DQN with one replay memory.

1) against the original DQN with one replay memory. Fig. 6 shows the loss, reward, and average state-action value during the training processes of these two algorithms. Both of the training processes last $260k$ training epochs. For comparison purpose, an exponential moving average (EMA) algorithm is applied to smooth each raw data curve of these metrics. Thus, the data tendency can be revealed by the smoothed curves.

As shown in Fig. 6a, the loss curves of these two algorithms decrease with different rates as the training process goes on. The smoothed loss curve of multiple replay memories decreases rapidly to the value of approximately 0.7 during the first $50k$ training epochs, and continues to decrease gradually to the value about 0.3 at the end of training. On the other hand, the smoothed loss curve of one replay memory also decreases within the first $10k$ training epochs, and then reaches the value of approximately 1. However, it does not constantly decrease anymore after around the $10k$-th training epoch. Instead, it fluctuates around the value of 1. At the end of training, the DQN with multiple replay memories has much lower smoothed loss value than the DQN with one reply memory, i.e., a better convergence performance.

Fig. 6b shows the average state-action value at each training epoch, which is calculated by averaging all outputs of the prediction network, i.e., $\sum_{a\in\mathcal{A}} Q(s,a;\theta)/|\mathcal{A}|$. The reason of applying the average value is that the training process with higher average state-action value is more likely to outperform the others. Conversely, it is not necessary to record the state-action value of the executed action in each training epoch, since different state-action pairs corresponding to different state-action values, and different training processes may follow different state transition trajectories.

We firstly focus on the performances of the DQN with multiple replay memories. As shown in Fig. 6b, the average state-action values are around 0 in the initial several training epochs. It is because the kernels of each fully connected layers of Q-networks are initialized by the Glorot normal initializer, and the biases of each layers are initialized by zeros, which leads to the outputs of Q-networks in a small scale. Moreover, the average state-action values vary between 0 and -350 during the first $50k$ training epochs. It is due to the fact that the parameters ε_{i_m} of ε-greedy rule do not decay to the minimal value of $\varepsilon_{\min}$, and the agent takes more exploration in these training epochs. Thus, the Q-networks are not well trained, which is coincident with the above analysis, i.e, the loss values are not convergent in the first $50k$ training epochs as shown in Fig. 6a. On the other hand, Fig. 6c also reflects this phenomenon. Initially, the agent always takes exploratory moves (i.e., random action), which leads to a low immediate reward. The reward curve increases with the increasing number of the training epochs, because the agent takes more greedy actions according to state-action values with ε_{i_m} decaying. After around $50k$ training epochs, the smoothed average state-action value curve is convergent to the value about -180, and the smoothed reward curve converges to the value about -6.

The average state-action value and reward curves of the DQN with one replay memory are also shown in Fig. 6b and Fig. 6c, respectively. The reward curve has a similar tendency to that of the DQN with multiple replay memories. At the end of training, the smoothed reward converges to the value about -10. The smoothed average state-action value curve decreases with the increasing number of training epochs, and converges to the value about -240 at the end of training. It can be seen that both of the average state-action value and reward of the DQN with one replay memory are much lower than those of the DQN with multiple replay memories. In other words, the DQN with multiple replay memories can learn a better policy with the same training epochs.

Therefore, it can be found that the improved DQN with multiple replay memories has a better training performance than the original DQN algorithm.

B. Performance Evaluation

We compare the policy learned by Algorithm 1 with the following reference policies, i.e.,

- **Resource precedence policy**: When the job queue is not empty $\mathbf{q}_1 \neq \mathbf{0}$, this policy firstly selects the actions that can allocate the most available resources to the job, i.e., $\mathcal{A}_1(s) = \left\{(\delta_\tau, \delta_c) = \arg\max_{(\delta_\tau,\delta_c)\in\mathcal{A}_c(s)} \delta_c\right\}$. Then, the policy takes the action a with the least time slice offset from $\mathcal{A}_1(s)$, i.e., $a = \arg\min_{(\delta_\tau,\delta_c)\in\mathcal{A}_1(s)} \delta_\tau$.
- **Time precedence policy**: Similar to the previous policy, this policy firstly selects the actions with the least time slice offset, i.e., $\mathcal{A}_1(s) = \left\{(\delta_\tau, \delta_c) = \arg\min_{(\delta_\tau,\delta_c)\in\mathcal{A}_c(s)} \delta_\tau\right\}$. Then, the policy takes the action a that allocates the most resources from $\mathcal{A}_1(s)$, i.e., $a = \arg\max_{(\delta_\tau,\delta_c)\in\mathcal{A}_1(s)} \delta_c$.

Both of the reference policies are based on the fact that allocating the most available resources is aiming to reduce duration of processing the job $q_{1,1}/\delta_c$, and the least time slice offset means the minimal waiting time δ_τ. To ensure the sufficient computing resources for processing jobs locally, these two policies always request the most computing resources at each adjusting epoch, i.e, $a = n_c = \arg\max_{(-2,\delta_\psi)\in\mathcal{A}_\psi} \delta_\psi$, when $\mathbf{q}_1 = \mathbf{0}$.

The experiment of each policy runs $200k$ decision epochs. The policy of the improved DQN algorithm is obtained according to (16), which is based on the prediction network trained in the previous Section IV-A. Moreover, the weights of prediction network are not updated any more in the following experiments.

Fig. 7 shows the average values of reward, completion time of jobs, and the number of requested computing units over the decision epochs. For each policy, the average reward is equal to the negative value of the sum of average completion time of jobs and average number of requested computing units, which conforms to our expectations according to the design of reward function in (12). As can be seen, the average completion time of the policy learned by Algorithm 1 is equal to 4.03, which is longer than those of the other reference policies, i.e., 3.39 and 3.38, respectively. It implies that scheduling jobs with the learned policy may cost more time on average. However, on the other hand, the learned policy requests less computing units from the MEC system during the long run. From the results of average reward, we can see that the policy obtained by Algorithm 1 learns the tradeoff between

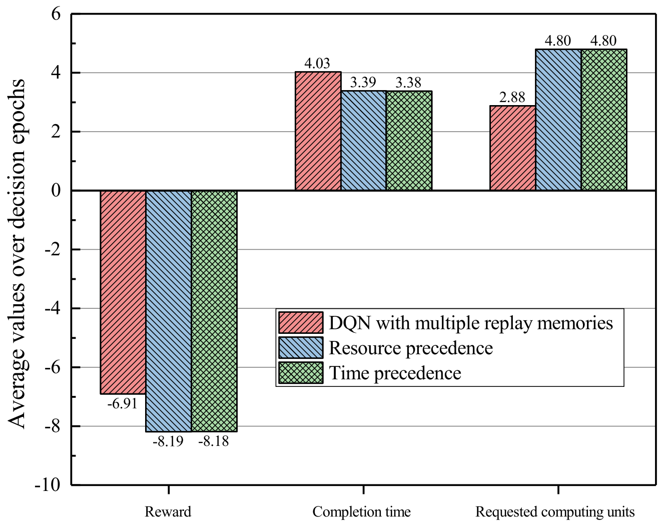
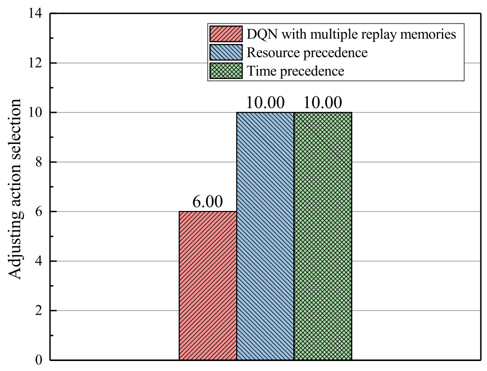


Fig. 7: The average values of reward, completion time of jobs, and the number of requested computing units over the decision epochs.

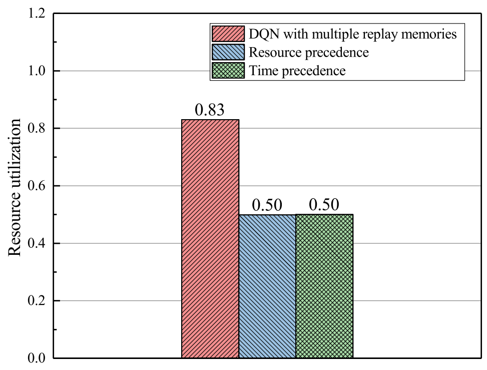
the completion time of jobs and the number of requested computing units. As a consequence, the average reward of the learned policy is -6.91, which is higher than those of the other reference policies, i.e., -8.19 and -8.18, respectively. The higher average reward means the smaller weighted sum of the average completion time of jobs and the average number of requested computing units. Therefore, the policy learned by Algorithm 1 outperforms the reference policies, and is much closer to optimization objective.

More experiments are carried out to show the tradeoff mentioned above. As shown in Fig. 8a, the policy learned by Algorithm 1 always requests 6 computing resources at each adjusting epoch, instead of the total number of computing resources $n_c = 10$. It is because the learned policy takes adjusting actions based on the observed states, which contains the information of the queue state, such as the number of jobs waiting in the queue, the number of computing units required by each waiting job, etc. After being well trained, the policy can estimate the reasonable number of computing resources requested in adjusting epochs to avoid the resource waste, which can be demonstrated by the resource utilization as shown in Fig. 8b. The resource utilization is calculated by dividing the number of computing units allocated to jobs by the total number of requested computing units. The resource utilization of the learned policy is 0.83, while both of the reference policies have the same resource utilization of 0.5. That is, about a half of the requested computing units are wasted when applying the reference policies. By contrast, the policy learned by Algorithm 1 makes more efficient use of the computing units through the more reasonable selection of adjusting actions.

Fig. 9a shows the percentage of jobs processed in the MEC system. It can be seen that all the policies allocate computing resources to jobs as far as possible, instead of forwarding jobs to the remote IoT cloud for further processing. Our proposed Algorithm 1 can learn this behavior, because the

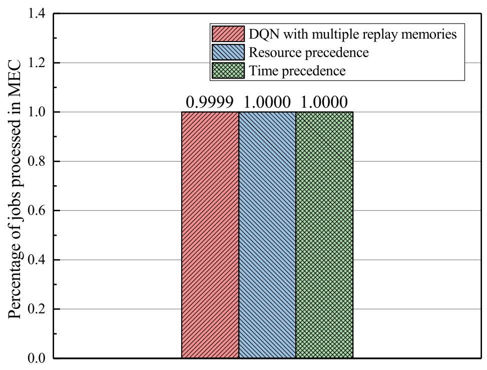


(a) Adjusting action selection

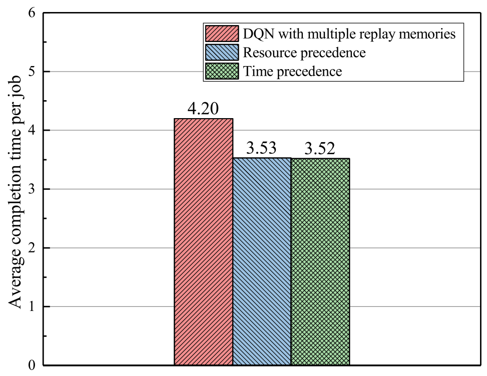


(b) Resource utilization

Fig. 8: Performance of adjusting actions.



(a) Percentage of jobs processed in MEC



(b) Average completion time per job

Fig. 9: Performance of scheduling actions.

action of forwarding the job generates a much lower reward than processing the job locally according to (10). Moreover, this behavior is not affected by the aforementioned adjusting action selection, i.e., requesting less computing units to keep a high resource utilization. It indicates that the learned policy still requests enough computing units for processing the jobs in the MEC system. However, the adjusting action selection has a side effect on the average completion time per job as shown in Fig. 9b. The average completion time per job of the learned policy is about 4.2, which is a little larger than those of the reference policies, i.e., 3.53 and 3.52, respectively. It is because the requested computing resources are just enough for allocating the jobs. Therefore, the jobs after scheduled may be queued at different time slice offsets of the computing resources, which leads to more processing time. Nevertheless, the slowdown caused by the side effect is not more than the duration of one timestep, which is acceptable to the most IoT edge applications.

V. Conclusions

In this paper, we have formulated the resource allocation problem in IoT edge computing system as a MDP model. To solve the problem, we propose an improved DQN algorithm where multiple replay memories are applied. Supported by simulations, the proposed algorithm can achieve the less loss value than the original DQN algorithm with the same number of training epochs. Moreover, the proposed algorithm can also achieve higher values of average state-action value and average reward. It demonstrates that our proposed algorithm has a better convergence performance. The simulation results also show that the corresponding policy learned by our proposed algorithm outperforms the other reference policies by a higher average reward over the long run. It means that the learned policy can achieve the smaller weighted sum of average completion time of jobs and average number of requested resources, which is coincident with our optimization objective.

Acknowledgment

This work is supported by China Natural Science Funding under grant 61731004, and BUPT Excellent Ph.D. Students Foundation under grant CX2016314.

References

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

[2] P. Mach and Z. Becvar, "Mobile Edge Computing: A Survey on Architecture and Computation Offloading," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[3] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile Edge Computing: A Survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, Feb. 2018.

[4] Y. Mehmood, N. Haider, M. Imran, A. Timm-Giel, and M. Guizani, "M2m Communications in 5g: State-of-the-Art Architecture, Recent Advances, and Research Challenges," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 194–201, 2017.

[5] J. Kim, S. Kim, T. Taleb, and S. Choi, "RAPID: Contention Resolution Based Random Access Using Context ID for IoT," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 7, pp. 7121–7135, Jul. 2019.

[6] "ETSI GS MEC-IEG 004: Service Scenarios," Nov. 2015.

[7] N. H. Motlagh, M. Bagaa, and T. Taleb, "Energy and Delay Aware Task Assignment Mechanism for UAV-Based IoT Platform," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6523–6536, Aug. 2019.

[8] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge Computing for the Internet of Things: A Case Study," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1275–1284, Apr. 2018.

[9] J. Pan and J. McElhannon, "Future Edge Cloud and Edge Computing for Internet of Things Applications," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, Feb. 2018.

[10] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on Multi-Access Edge Computing for Internet of Things Realization," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 2961–2991, 2018.

[11] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A Survey on the Edge Computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[12] 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*, vol. 5, no. 4, pp. 84–91, Oct. 2016.

[13] "ETSI GS MEC 003: Framework and Reference Architecture," Mar. 2016.

[14] J. An, W. Li, F. L. Gall, E. Kovac, J. Kim, T. Taleb, and J. Song, "EiF: Toward an Elastic IoT Fog Framework for AI Services," *IEEE Communications Magazine*, vol. 57, no. 5, pp. 28–33, May 2019.

[15] R. Morabito, I. Farris, A. Iera, and T. Taleb, "Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge," *IEEE Internet of Things Journal*, vol. 4, no. 4, pp. 1019–1030, Aug. 2017.

[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, ser. Adaptive computation and machine learning. Cambridge, Mass: MIT Press, 1998.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. 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," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[18] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," *arXiv:1502.05477 [cs]*, Feb. 2015, arXiv: 1502.05477.

[19] M. Wiering and M. v. Otterlo, Eds., *Reinforcement learning: state-of-the-art*, ser. Adaptation, learning, and optimization. Heidelberg ; New York: Springer, 2012.

[20] K. Zheng, H. Meng, P. Chatzimisios, L. Lei, and X. Shen, "An SMDP-Based Resource Allocation in Vehicular Cloud Computing Systems," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 12, pp. 7920–7928, Dec. 2015.

[21] Q. Zheng, K. Zheng, H. Zhang, and V. C. M. Leung, "Delay-Optimal Virtualized Radio Resource Scheduling in Software-Defined Vehicular Networks via Stochastic Learning," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 10, pp. 7857–7867, Oct. 2016.

[22] Q. Ye, J. Li, K. Qu, W. Zhuang, X. S. Shen, and X. Li, "End-to-end quality of service in 5g networks: Examining the effectiveness of a network slicing framework," *IEEE Vehicular Technology Magazine*, vol. 13, no. 2, pp. 65–74.

[23] Q. Ye, W. Zhuang, S. Zhang, A.-L. Jin, X. Shen, and X. Li, "Dynamic radio resource slicing for a two-tier heterogeneous wireless network," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 10, pp. 9896–9910.

[24] H. Peng, Q. Ye, and X. S. Shen, "SDN-based resource management for autonomous vehicular networks: A multi-access edge computing approach," *IEEE Wireless Communications*, vol. 26, no. 4, pp. 156–162.

[25] L. Lei, Y. Tan, S. Liu, K. Zhuang, K. Zheng, and X. Shen, "Deep Reinforcement Learning for Autonomous Internet of Things: Model, Applications and Challenges," *arXiv:1907.09059 [cs, stat]*, Jul. 2019, arXiv: 1907.09059 version: 1. [Online]. Available: http://arxiv.org/abs/1907.09059

[26] M. Bagaa, A. Ksentini, T. Taleb, R. Jantti, A. Chelli, and I. Balasingham, "An efficient D2D-based strategies for machine type communications in 5G mobile systems," in *2016 IEEE Wireless Communications and Networking Conference*, Apr. 2016, pp. 1–6, iSSN: 1558-2612.

[27] A. T. Nassar and Y. Yilmaz, "Reinforcement Learning-based Resource Allocation in Fog RAN for IoT with Heterogeneous Latency Requirements," *arXiv:1806.04582 [cs]*, Jan. 2019, arXiv: 1806.04582. [Online]. Available: http://arxiv.org/abs/1806.04582

[28] 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*, p. 6, 2018.

[29] X. Liu, Z. Qin, and Y. Gao, "Resource allocation for edge computing in IoT networks via reinforcement learning," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. IEEE, pp. 1–6.

[30] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Performance Optimization in Mobile-Edge Computing via Deep Reinforcement Learning," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, Aug. 2018, pp. 1–6, iSSN: 1090-3038.

[31] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digital Communications and Networks*, Oct. 2018.

[32] Y. He, N. Zhao, and H. Yin, "Integrated Networking, Caching, and Computing for Connected Vehicles: A Deep Reinforcement Learning Approach," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 1, pp. 44–55, Jan. 2018.

[33] L. Lei, H. Xu, X. Xiong, K. Zheng, W. Xiang, and X. Wang, "Multiuser Resource Control With Deep Reinforcement Learning in IoT Edge Computing," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 119–10 133, Dec. 2019.

[34] Y. Wei, F. R. Yu, M. Song, and Z. Han, "Joint Optimization of Caching, Computing, and Radio Resources for Fog-Enabled IoT Using Natural Actor–Critic Deep Reinforcement Learning," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2061–2073, Apr. 2019.

[35] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource Management with Deep Reinforcement Learning." ACM Press, 2016, pp. 50–56.

[36] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[37] 3GPP, "Cellular system support for ultra-low complexity and low throughput Internet of Things (CIoT)," 3rd Generation Partnership Project (3GPP), Technical Report (TR) 45.820, 11 2015, version 13.1.0.