Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems

Ming Tang, *Member, IEEE* and Vincent W.S. Wong, *Fellow, IEEE*

Abstract—In mobile edge computing systems, an edge node may have a high load when a large number of mobile devices offload their tasks to it. Those offloaded tasks may experience large processing delay or even be dropped when their deadlines expire. Due to the uncertain load dynamics at the edge nodes, it is challenging for each device to determine its offloading decision (i.e., whether to offload or not, and which edge node it should offload its task to) in a decentralized manner. In this work, we consider non-divisible and delay-sensitive tasks as well as edge load dynamics, and formulate a task offloading problem to minimize the expected long-term cost. We propose a model-free deep reinforcement learning-based distributed algorithm, where each device can determine its offloading decision without knowing the task models and offloading decision of other devices. To improve the estimation of the long-term cost in the algorithm, we incorporate the long short-term memory (LSTM), dueling deep Q-network (DQN), and double-DQN techniques. Simulation results show that our proposed algorithm can better exploit the processing capacities of the edge nodes and significantly reduce the ratio of dropped tasks and average delay when compared with several existing algorithms.

Index Terms—Mobile edge computing, computation offloading, resource allocation, deep reinforcement learning, deep Q-learning

1 Introduction

1.1 Background and Motivation

Nowadays, mobile devices are responsible for processing many computational intensive tasks, such as data processing and artificial intelligence. Despite the development of mobile devices, these devices may not be able to process all their tasks locally with a low latency due to their limited computational resources. To facilitate efficient task processing, mobile edge computing (MEC) [1], also known as fog computing [2] and multi-access edge computing [3], is introduced. MEC facilitates mobile devices to offload their computational tasks to nearby edge nodes for processing in order to reduce the task processing delay. It can also reduce the ratio of dropped tasks for those delay-sensitive tasks.

In MEC, there are two main questions related to task offloading. The first question is whether a mobile device should offload its task to an edge node or not. The second question is that if a mobile device decides to perform offloading, then which edge node should the device offload its task to. To address these questions, some existing works have proposed task offloading algorithms. Wang *et al.* in [4] proposed an algorithm to determine the offloading decisions to maximize the network revenue. Bi *et al.* in [5] considered a wireless-powered MEC scenario and proposed an algorithm to optimize the offloading and power transfer decisions. In the works [4], [5], the processing capacity that each device obtained from an edge node is independent of the number of tasks offloaded to the edge node.

• *The authors are with the Department of Electrical and Computer Engineering, The University of British Columbia, Vancouver, BC V6T 1Z4, Canada. E-mail: {mingt, vincentw}@ece.ubc.ca.*

Manuscript received 28 Mar. 2020; revised 31 Oct. 2020; accepted 3 Nov. 2020. Date of publication 10 Nov. 2020; date of current version 5 May 2022. (Corresponding author: Vincent W.S. Wong) Digital Object Identifier no. 10.1109/TMC.2020.3036871

In practice, however, edge nodes may have limited processing capacities, so the processing capacity that an edge node allocates to a mobile device depends on the *load level* at the edge node (i.e., number of concurrent tasks offloaded to the edge node). When a large number of mobile devices offload their tasks to the same edge node, the load at that edge node can be high, and hence those offloaded tasks may experience large processing delay. Some of the tasks may even be dropped when their deadlines expire. Some existing works have addressed the load levels at the edge nodes and proposed centralized task offloading algorithms. Eshraghi *et al.* in [6] proposed an algorithm that optimizes the offloading decisions of mobile devices, considering their uncertain computational requirements. Lyu *et al.* in [7] focused on delay-sensitive tasks and proposed an algorithm to minimize the task offloading energy consumption subject to the task deadline constraint. In [8], Chen *et al.* designed a centralized algorithm for a software-defined ultra-dense network to minimize task delay. In [9], Poularakis *et al.* studied the joint optimization of task offloading and routing. The operation of these centralized algorithms in [6], [7], [8], [9], however, may require complete information of the system.

Other works have proposed distributed task offloading algorithms considering the load levels at the edge nodes, where each mobile device makes its offloading decision in a decentralized manner. Designing such a distributed algorithm is challenging. This is because when a device makes an offloading decision, the device does not know *a priori* the load levels at the edge nodes, since the load also depends on the offloading decisions of other mobile devices. In addition, the load levels at the edge nodes may change over time. To address these challenges, Lyu *et al.* in [10] focused on divisible tasks and proposed a Lyapunov-based algorithm to ensure the stability of the task queues. In [11], Li *et al.* considered the strategic offloading interaction among

mobile devices and proposed a price-based distributed algorithm. Shah-Mansouri *et al.* in [12] designed a potential game-based offloading algorithm to maximize the quality-of-experience of each device. Jošilo *et al.* in [13] designed a distributed algorithm based on a Stackelberg game. Yang *et al.* in [14] proposed a distributed offloading algorithm to address the wireless channel competition among mobile devices. Neto *et al.* in [15] proposed an estimation-based method, where each device makes its offloading decision based on the estimated processing and transmission capacities. Lee *et al.* in [16] proposed an algorithm based on online optimization techniques to minimize the maximum delay of the tasks.

In this work, we focus on the task offloading problem in an MEC system and propose a distributed algorithm that addresses the unknown load level dynamics at the edge nodes. Comparing with the aforementioned works [10], [11], [12], [13], [14], [15], [16], we consider a different and more realistic MEC scenario. First, the existing work [10] considered divisible tasks (i.e., tasks can be arbitrarily divided), which may not be realistic due to the dependency among the bits in a task. Although the works [11], [12], [13], [14], [15] considered non-divisible tasks, they do not consider the underlying queuing systems. As a result, the processing and transmission of each task should always be accomplished within one time slot (or before the arrival of the next task), which may not always be guaranteed in practice. Different from those works [10], [11], [12], [13], [14], [15], we consider non-divisible tasks together with queuing systems and take into account the practical scenario where the processing and transmission of a task can continue for multiple time slots. This scenario is challenging to deal with, because when a new task arrives, its delay can be affected by the decisions of the tasks of other devices arrived in the previous time slots. Second, different from the related works [10], [11], [12], [13], [14], [15], [16] considered delay-tolerant tasks, we focus on delay-sensitive tasks with processing deadlines. This is challenging to address since the deadlines can affect the load levels at the edge nodes and hence the delay of the offloaded tasks.

Under the aforementioned MEC system, it is difficult to apply traditional methods such as game theory and online optimization due to the complicated interaction among the tasks. To address the challenges, in this work, we propose to use deep Q-learning [17], which is a model-free deep reinforcement learning (DRL) technique. This approach enables the agents to make decisions based on local observations without the knowledge of the system modeling and dynamics. Some existing works such as [18], [19], [20] have proposed DRL-based algorithms for MEC systems, while they focused on centralized offloading algorithms. Zhao *et al.* in [21] proposed a DRL-based distributed offloading algorithm that addresses the wireless channel competition among mobile devices, while the algorithm at each mobile device requires the quality-of-service information of other mobile devices. Different from those works [18], [19], [20], [21], we aim to propose a DRL-based distributed algorithm that addresses the unknown load dynamics at edge nodes. It enables each mobile device to make its offloading decision without knowing the information (e.g., task models, offloading decisions) of other mobile devices.

1.2 Solution Approach and Contributions

In this work, we take into account the unknown load level dynamics at the edge nodes and propose a DRL-based distributed offloading algorithm for the MEC system. In the proposed algorithm, each mobile device can determine the offloading decision in a decentralized manner using its information observed locally, including the size of its task, the information of its queues, and the historical load levels at the edge nodes. The main contributions are as follows.

- *Task Offloading Problem for the MEC System:* We formulate a task offloading problem for non-divisible and delay-sensitive tasks. The problem takes into account the load level dynamics at the edge nodes and aims at minimizing the expected long-term cost of the tasks (considering the delay of the tasks and the penalties for those tasks being dropped).
- *DRL-based Task Offloading Algorithm:* To achieve the expected long-term cost minimization, we propose a model-free DRL-based distributed offloading algorithm that enables each mobile device to make its offloading decision without knowing the task models and offloading decisions of other mobile devices. To improve the estimation of the expected long-term cost in the proposed algorithm, we incorporate the long short-term memory (LSTM), dueling deep Q-network (DQN), and double-DQN techniques.
- *Performance Evaluation:* We perform simulations and show that when compared with the potential game based offloading algorithm (PGOA) in [14] and the user-level online offloading framework (ULOOF) in [15], our proposed algorithm can better exploit the processing capacities of the mobile devices and edge nodes, and it can significantly reduce the ratio of dropped tasks and the average delay.

The rest of this paper is organized as follows. The system model is presented in Section 2, and the problem formulation is given in Section 3. We present the DRL-based algorithm in Section 4 and evaluate its performance in Section 5. Conclusion is given in Section 6. For notation, we use $\mathbb{Z}_{++}$ to denote the set of positive integers.

2 System Model

We consider a set of edge nodes $\mathcal{N} = \{1, 2, \ldots, N\}$ and a set of mobile devices $\mathcal{M} = \{1, 2, \ldots, M\}$ in an MEC system. We focus on one episode that contains a set of time slots $\mathcal{T} = \{1, \ldots, T\}$, where each time slot has a duration of Δ seconds. In the following, we present the mobile device and edge node models, with an illustration given in Fig. 1.

2.1 Mobile Device Model

We focus on computational tasks of mobile devices, where each task is non-divisible such that it can either be processed locally or be offloaded to an edge node for processing. We assume that at the beginning of each time slot, a mobile device has a new task arrival with a certain probability. This assumption is consistent with some existing works (e.g., [22]). When a mobile device has a new task arrival, it first needs to decide whether to process the task locally or offload it to an edge node. If the mobile device decides to

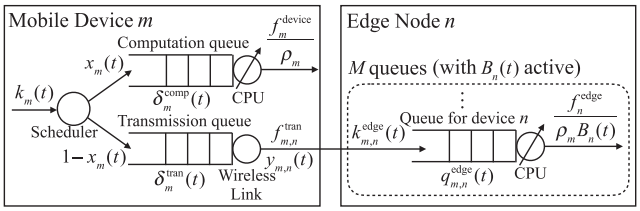


Fig. 1. An illustration of an MEC system with a mobile device $m \in \mathcal{M}$ and an edge node $n \in \mathcal{N}$.

process the task locally, then its scheduler (see Fig. 1) will place the task to the computation queue for processing. Otherwise, the mobile device needs to decide which edge node the task is offloaded to. The scheduler will then place the task to the transmission queue for offloading. The offloaded task will then be sent to the chosen edge node through a wireless link. For the computation (or transmission) queue, we assume that if the processing (or transmission) of a task is completed in a time slot, then the next task in the queue will be processed (or transmitted) at the beginning of the next time slot. This assumption is consistent with some existing works considering queuing dynamics in an MEC system (e.g., [22]).

In the following, we first present the task model and the task offloading decision, respectively. Then, we introduce the computation and transmission queues.

2.1.1 Task Model

At the beginning of time slot $t \in \mathcal{T}$, if mobile device $m \in \mathcal{M}$ has a newly arrived task, then we define a variable $k_m(t) \in \mathbb{Z}_{++}$ to denote the unique index of the task. If mobile device m does not have a new task arrival at the beginning of time slot t, then $k_m(t)$ is set to zero for presentation simplicity.

Let $\lambda_m(t)$ (in bits) denote the number of bits of the newly arrived task at the beginning of time slot $t \in \mathcal{T}$. If there exists a new task $k_m(t)$ at the beginning of time slot t, then $\lambda_m(t)$ is equal to the size of task $k_m(t)$. Otherwise, $\lambda_m(t)$ is set to zero. We set the size of task $k_m(t)$ to be from a discrete set $\Lambda \triangleq \{\lambda_1, \lambda_2, \ldots, \lambda_{|\Lambda|}\}$ with $|\Lambda|$ available values. Hence, $\lambda_m(t) \in \Lambda \cup \{0\}$. In addition, task $k_m(t)$ requires a processing density of ρ_m (in CPU cycles per bit), i.e., the number of CPU cycles required to process a unit of data. Task $k_m(t)$ has a deadline τ_m (in time slots). That is, if task $k_m(t)$ has not been completely processed by the end of time slot $t + \tau_m - 1$, then it will be dropped immediately.

2.1.2 Task Offloading Decision

If mobile device $m \in \mathcal{M}$ has a newly arrived task $k_m(t)$ at the beginning of time slot $t \in \mathcal{T}$, then it needs to make the offloading decision for task $k_m(t)$ as follows.

First, let binary variable $x_m(t) \in \{0, 1\}$ denote whether task $k_m(t)$ is to be processed locally or offloaded to an edge node. We set $x_m(t) = 1$ (or 0) if the task is to be processed locally (or to be offloaded to an edge node). At the beginning of time slot t, $\lambda_m(t)x_m(t)$ is the number of bits arrived at the computation queue of mobile device m, and $\lambda_m(t)(1 - x_m(t))$ is the number of bits arrived at the transmission queue of mobile device m.

Second, if task $k_m(t)$ is to be offloaded to an edge node, then let binary variable $y_{m,n}(t) \in \{0, 1\}$ denote whether task $k_m(t)$ is offloaded to edge node $n \in \mathcal{N}$ or not. We set $y_{m,n}(t) = 1$ if task $k_m(t)$ is offloaded to edge node n, and $y_{m,n}(t) = 0$ otherwise. Let $\boldsymbol{y}_m(t) = (y_{m,n}(t), n \in \mathcal{N})$. Note that each task can be offloaded to one edge node, i.e.,

$$\sum_{n \in \mathcal{N}} y_{m,n}(t) = \mathbb{1}(x_m(t) = 0), \ m \in \mathcal{M}, t \in \mathcal{T}, \tag{1}$$

where the indicator $\mathbb{1}(z \in \mathcal{Z}) = 1$ if $z \in \mathcal{Z}$, and is equal to zero otherwise.

2.1.3 Computation Queue

The computation queue operates in a first-in first-out (FIFO) manner. The arrivals are the tasks to be processed locally. We consider one CPU which processes the tasks in the computation queue. Let f_m^{device} (in CPU cycles per second) denote the processing capacity of the CPU of mobile device $m \in \mathcal{M}$. The value of f_m^{device} is a constant. At the beginning of time slot $t \in \mathcal{T}$, if task $k_m(t)$ is placed in the computation queue, then we define $l_m^{\text{comp}}(t) \in \mathcal{T}$ to denote the time slot when task $k_m(t)$ has either been processed or dropped. Without loss of generality, if either task $k_m(t)$ is not placed in the computation queue or $k_m(t) = 0$, then we set $l_m^{\text{comp}}(t) = 0$.

Let $\delta_m^{\text{comp}}(t)$ (in time slots) denote the number of time slots that task $k_m(t)$ will wait for processing if it is placed in the computation queue. Note that mobile device m will compute the value of $\delta_m^{\text{comp}}(t)$ before it decides the queue to place the task. Given $l_m^{\text{comp}}(t')$ for $t' < t$, the value of $\delta_m^{\text{comp}}(t)$ is computed as follows. For $m \in \mathcal{M}$ and $t \in \mathcal{T}$,

$$\delta_m^{\text{comp}}(t) = \left[\max_{t' \in \{0,1,\ldots,t-1\}} l_m^{\text{comp}}(t') - t + 1 \right]^+, \tag{2}$$

where the operator $[z]^+ = \max\{0, z\}$, and we set $l_m^{\text{comp}}(0) = 0$. Specifically, the term $\max_{t' \in \{0,1,2,\ldots,t-1\}} l_m^{\text{comp}}(t')$ determines the time slot when all the tasks placed in the computation queue before time slot t has either been processed or dropped. Hence, $\delta_m^{\text{comp}}(t)$ determines the number of time slots that task $k_m(t)$ should wait for processing. For example, suppose task $k_m(1)$ is placed in the computation queue, and its processing will be completed in time slot 5, i.e., $l_m^{\text{comp}}(1) = 5$. Meanwhile, suppose $k_m(2) = 0$, i.e., $l_m^{\text{comp}}(2) = 0$. At the beginning of time slot 3, if task $k_m(3)$ is placed in the computation queue, then its processing will start after time slot $l_m^{\text{comp}}(1) = 5$. Hence, it should wait for $\delta_m^{\text{comp}}(3) = [\max\{5, 0\} - 3 + 1]^+ = 3$ time slots.

If mobile device $m \in \mathcal{M}$ places task $k_m(t)$ in the computation queue at the beginning of time slot $t \in \mathcal{T}$ (i.e., $x_m(t) = 1$), then task $k_m(t)$ will have either been processed or dropped in time slot $l_m^{\text{comp}}(t)$:

$$l_m^{\text{comp}}(t) = \min\left\{ t + \delta_m^{\text{comp}}(t) + \left\lceil \frac{\lambda_m(t)}{f_m^{\text{device}} \Delta / \rho_m} \right\rceil - 1, \ t + \tau_m - 1 \right\}, \tag{3}$$

where $\lceil \cdot \rceil$ is the ceiling function. Specifically, the processing of task $k_m(t)$ will start at the beginning of time slot $t +$

$\delta_m^{\text{comp}}(t)$. The number of time slots required to process the task is $\lceil \lambda_m(t)/(f_m^{\text{device}}\Delta/\rho_m) \rceil$. Hence, the first term in the min operator is the time slot when the processing of task $k_m(t)$ will be completed without considering the deadline of the task. The second term is the time slot when task $k_m(t)$ will be dropped. As a result, $l_m^{\text{comp}}(t)$ determines the time slot when task $k_m(t)$ will either be processed or dropped.

2.1.4 Transmission Queue

The transmission queue operates in a FIFO manner. The arrivals are the tasks to be offloaded to the edge nodes. The wireless network link interface at the mobile device sends the tasks in the transmission queue to the chosen edge node. The wireless network model and the transmission rate from a mobile device to an edge node are as follows. We consider a wireless network model where mobile devices transmit on orthogonal channels. The wireless transmission from a mobile device to an edge node suffers from path loss and small-scale fading [11], [23]. Let $|h_{m,n}|^2$ denote the channel gain from mobile device $m \in \mathcal{M}$ to edge node $n \in \mathcal{N}$. Let P denote the transmission power of a mobile device. The transmission rate from mobile device m to edge node n, denoted by $r_{m,n}^{\text{tran}}$ (in bits per second), is computed as follows:

$$r_{m,n}^{\text{tran}} = W \log_2\left(1 + \frac{|h_{m,n}|^2 P}{\sigma^2}\right),\ m \in \mathcal{N}, n \in \mathcal{N}, \tag{4}$$

where W denotes the bandwidth allocated to a channel, and σ^2 denotes the received noise power at the edge node. The value of $r_{m,n}^{\text{tran}}$ is assumed to be a constant.

At the beginning of time slot $t \in \mathcal{T}$, if task $k_m(t)$ is placed in the transmission queue for offloading, then we define a variable $l_m^{\text{tran}}(t) \in \mathcal{T}$ to denote the time slot when task $k_m(t)$ has been either sent or dropped. Without loss of generality, if either task $k_m(t)$ is not placed in the transmission queue or $k_m(t) = 0$, then we set $l_m^{\text{tran}}(t) = 0$. Let $\delta_m^{\text{tran}}(t)$ (in time slots) denote the number of time slots that task $k_m(t)$ should wait for transmission if it is placed in the transmission queue. Note that mobile device m will compute the value of $\delta_m^{\text{tran}}(t)$ before it has decided on which queue to place the task. Given $l_m^{\text{tran}}(t')$ for $t' < t$, the value of $\delta_m^{\text{tran}}(t)$ is computed as follows. For $m \in \mathcal{M}$ and $t \in \mathcal{T}$,

$$\delta_m^{\text{tran}}(t) = \left[\max_{t' \in \{0,1,\ldots,t-1\}} l_m^{\text{tran}}(t') - t + 1\right]^+, \tag{5}$$

where we set $l_m^{\text{tran}}(0) = 0$ for presentation simplicity.

If device $m \in \mathcal{M}$ places task $k_m(t)$ in the transmission queue in time slot $t \in \mathcal{T}$ (i.e., $x_m(t) = 0$), then task $k_m(t)$ will either be sent or dropped in time slot $l_m^{\text{tran}}(t)$:

$$l_m^{\text{tran}}(t) = \min\left\{t + \delta_m^{\text{tran}}(t) + \left\lceil \sum_{n \in \mathcal{N}} \frac{y_{m,n}(t)\lambda_m(t)}{r_{m,n}^{\text{tran}}\Delta} \right\rceil - 1,\ t + \tau_m - 1\right\}. \tag{6}$$

2.2 Edge Node Model

Each edge node $n \in \mathcal{N}$ maintains M queues, each queue corresponding to a mobile device in set $\mathcal{M}$. We assume that after an offloaded task is received by an edge node in a time slot, the task will be placed in the corresponding queue at the edge node at the beginning of the next time slot.

If a task of mobile device $m \in \mathcal{M}$ is placed in its queue at edge node $n \in \mathcal{N}$ at the beginning of time slot $t \in \mathcal{T}$, then we define a variable $k_{m,n}^{\text{edge}}(t) \in \mathbb{Z}_{++}$ to denote the unique index of the task. Specifically, if task $k_m(t')$ for $t' \in \{1, 2, \ldots, t-1\}$ is sent to edge node n in time slot $t-1$, then $k_{m,n}^{\text{edge}}(t) = k_m(t')$. Note that if there does not exist such a task, then we set $k_{m,n}^{\text{edge}}(t) = 0$. Let $\lambda_{m,n}^{\text{edge}}(t) \in \Lambda \cup \{0\}$ (in bits) denote the number of bits arrived in the queue of mobile device m at edge node n at the beginning of time slot t. If task $k_{m,n}^{\text{edge}}(t)$ is placed in the corresponding queue at the beginning of time slot t, then $\lambda_{m,n}^{\text{edge}}(t)$ is equal to the size of task $k_{m,n}^{\text{edge}}(t)$. Otherwise, $\lambda_{m,n}^{\text{edge}}(t) = 0$.

In the following, we first describe the queues. Then, we derive the task processing or dropping time.

2.2.1 Queues at Edge Nodes

The queue associated with a mobile device at an edge node operates in a FIFO manner. The arrivals of the queue are the tasks offloaded by the mobile device to that edge node. Let $q_{m,n}^{\text{edge}}(t)$ (in bits) denote the length of the queue of mobile device $m \in \mathcal{M}$ at edge node $n \in \mathcal{N}$ at the end of time slot $t \in \mathcal{T}$. Among those queues at edge node n, we refer to the queue of mobile device m as an *active queue* in time slot t if either there is a task of mobile device m arrived at the queue in time slot t (i.e., $\lambda_{m,n}^{\text{edge}}(t) > 0$) or the queue is non-empty at the end of time slot $t-1$ (i.e., $q_{m,n}^{\text{edge}}(t-1) > 0$). Let $\mathcal{B}_n(t)$ denote the set of active queues at edge node n in time slot t. That is, for $n \in \mathcal{N}$ and $t \in \mathcal{T}$,

$$\mathcal{B}_n(t) = \{m \mid \lambda_{m,n}^{\text{edge}}(t) > 0 \text{ or } q_{m,n}^{\text{edge}}(t-1) > 0,\ m \in \mathcal{M}\}. \tag{7}$$

Let $B_n(t)$ denote the number of active queues at edge node n in time slot t, i.e., $B_n(t) = |\mathcal{B}_n(t)|$.

We consider a scenario where the tasks of mobile devices have the same priority.[1] Each edge node has one CPU for processing the tasks in the queues. In each time slot $t \in \mathcal{T}$, the active queues at edge node $n \in \mathcal{N}$ (i.e., the queues in set $\mathcal{B}_n(t)$) equally share the processing capacity of the CPU at edge node n. This is the generalized processor sharing (GPS) model with equal processing capacity sharing [24]. Note that since the number of active queues $B_n(t)$ is time-varying and unknown *a priori*, the processing capacity allocated to each queue can vary across time. Meanwhile, the mobile devices and edge nodes may not have the information of this allocated processing capacity beforehand.

Let f_n^{edge} (in CPU cycles per second) denote the processing capacity of edge node n. We assume that mobile devices are aware of the value of f_n^{edge} for $n \in \mathcal{N}$. Let $e_{m,n}^{\text{edge}}(t)$ (in bits) denote the number of bits of the tasks dropped by the queue at the end of time slot $t \in \mathcal{T}$. Hence, the queue length is updated as follows. For $m \in \mathcal{M}$, $n \in \mathcal{N}$, and $t \in \mathcal{T}$,

1. This work can be extended to the scenario where the tasks of different mobile devices have different priorities. This can be achieved by setting different weights to the queues of different mobile devices and allocating the processing capacity based on the weights.

$$q_{m,n}^{\text{edge}}(t) = \Bigg[q_{m,n}^{\text{edge}}(t-1) + \lambda_{m,n}^{\text{edge}}(t) - \frac{f_n^{\text{edge}}\Delta}{\rho_m B_n(t)} \mathbb{1}(m \in \mathcal{B}_n(t)) - e_{m,n}^{\text{edge}}(t) \Bigg]^+. \tag{8}$$

2.2.2 Task Processing or Dropping

If task $k_{m,n}^{\text{edge}}(t)$ of mobile device $m \in \mathcal{M}$ is placed in the corresponding queue at edge node $n \in \mathcal{N}$ at the beginning of time slot $t \in \mathcal{T}$, then we define a variable $l_{m,n}^{\text{edge}}(t) \in \mathcal{T}$ to denote the time slot when this task has either been processed or dropped by edge node n. Due to the uncertain future load at edge node n, mobile device m and edge node n are unaware of the value of $l_{m,n}^{\text{edge}}(t)$ until the associated task $k_{m,n}^{\text{edge}}(t)$ has either been processed or dropped. Without loss of generality, if $k_{m,n}^{\text{edge}}(t) = 0$, then we set $l_{m,n}^{\text{edge}}(t) = 0$.

For the definition of variable $l_{m,n}^{\text{edge}}(t)$, let $\widehat{l}_{m,n}^{\text{edge}}(t)$ denote the time slot when the processing of task $k_{m,n}^{\text{edge}}(t)$ starts, i.e., for $m \in \mathcal{M}$, $n \in \mathcal{N}$, and $t \in \mathcal{T}$,

$$\widehat{l}_{m,n}^{\text{edge}}(t) = \max\Bigg\{ t, \max_{t' \in \{0,1,\ldots,t-1\}} l_{m,n}^{\text{edge}}(t') + 1 \Bigg\}, \tag{9}$$

where we set $l_{m,n}^{\text{edge}}(0) = 0$. Specifically, the time slot when the processing of task $k_{m,n}^{\text{edge}}(t)$ starts is no earlier than the time slot when the task arrives in the queue or when each task arrived earlier has been processed or dropped.

Given the realization of the load levels, $l_{m,n}^{\text{edge}}(t)$ satisfies the following constraints. For $m \in \mathcal{M}$, $n \in \mathcal{N}$, $t \in \mathcal{T}$,

$$\sum_{t'=\widehat{l}_{m,n}^{\text{edge}}(t)}^{l_{m,n}^{\text{edge}}(t)} \frac{f_n^{\text{edge}}\Delta}{\rho_m B_n(t')} \mathbb{1}(m \in \mathcal{B}_n(t')) \geq \lambda_{m,n}^{\text{edge}}(t), \tag{10}$$

$$\sum_{t'=\widehat{l}_{m,n}^{\text{edge}}(t)}^{l_{m,n}^{\text{edge}}(t)-1} \frac{f_n^{\text{edge}}\Delta}{\rho_m B_n(t')} \mathbb{1}(m \in \mathcal{B}_n(t')) < \lambda_{m,n}^{\text{edge}}(t). \tag{11}$$

Specifically, the size of task $k_{m,n}^{\text{edge}}(t)$ is no larger than the total processing capacity that edge node n allocated to mobile device m from time slot $\widehat{l}_{m,n}^{\text{edge}}(t)$ to $l_{m,n}^{\text{edge}}(t)$, and it is larger than that from time slot $\widehat{l}_{m,n}^{\text{edge}}(t)$ to $l_{m,n}^{\text{edge}}(t) - 1$.

3 Task Offloading Problem in MEC

At the beginning of each time slot, each mobile device observes its state (e.g., task size, queue information). If there is a new task to be processed, then the mobile device chooses an action for the task. The state and action will result in a cost (i.e., the delay of the task if the task is processed, or a penalty if it is dropped) for the mobile device. The objective of each device is to minimize its expected long-term cost by optimizing the policy mapping from states to actions.

3.1 State

Let matrix $\boldsymbol{H}(t)$ denote the history of the load level (i.e., the number of active queues) of each edge node within the previous T^{step} time slots (i.e., from time slot $t - T^{\text{step}}$ to time slot $t-1$). It is a matrix with size $T^{\text{step}} \times N$. Let $\{\boldsymbol{H}(t)\}_{i,j}$ denote the (i,j) element of $\boldsymbol{H}(t)$, which corresponds to the load level of edge node j in the ith time slot starting from time slot $t - T^{\text{step}}$ (i.e., time slot $t - T^{\text{step}} + i - 1$). That is,

$$\{\boldsymbol{H}(t)\}_{i,j} = B_j(t - T^{\text{step}} + i - 1). \tag{12}$$

To obtain $\boldsymbol{H}(t)$, we assume that each edge node broadcasts its number of active queues at the end of each time slot. Even when all M queues at an edge node are active, the number of active queues can be represented by $\lfloor \log_2 M \rfloor + 1$ bits. For example, if there are 1,000 mobile devices, then a maximum of 10 bits are required. Hence, the broadcast of the number of active queues only incurs a small signaling overhead.

At the beginning of time slot $t \in \mathcal{T}$, each device $m \in \mathcal{M}$ observes its state information, including task size, queue information, and the load level history at edge nodes. Specifically, mobile device m observes the following state:[2]

$$\boldsymbol{s}_m(t) = \Big(\lambda_m(t), \delta_m^{\text{comp}}(t), \delta_m^{\text{tran}}(t), \boldsymbol{q}_m^{\text{edge}}(t-1), \boldsymbol{H}(t)\Big), \tag{13}$$

where vector $\boldsymbol{q}_m^{\text{edge}}(t-1) = (q_{m,n}^{\text{edge}}(t-1), n \in \mathcal{N})$. Let $\mathcal{S}$ denote the discrete and finite state space of each mobile device, i.e., $\mathcal{S} = \Lambda \times \{0,1,\ldots,T\}^2 \times \mathcal{Q}^N \times \{0,1,\ldots,M\}^{T^{\text{step}} \times N}$, where $\mathcal{Q}$ denotes the set of the available values of the queue length at an edge node within the T time slots. Note that mobile device $m \in \mathcal{M}$ can obtain state information $\lambda_m(t)$, $\delta_m^{\text{comp}}(t)$, and $\delta_m^{\text{tran}}(t)$ through local observation at the beginning of time slot t. Meanwhile, mobile device m can compute $\boldsymbol{q}_m^{\text{edge}}(t-1)$ locally according to (8). Specifically, mobile device m is aware of the number of bits that it has transmitted to an edge node in each time slot. In addition, it can compute the number of bits that have been processed or being dropped by an each edge node in each time slot.[3]

3.2 Action

At the beginning of time slot $t \in \mathcal{T}$, if mobile device $m \in \mathcal{M}$ has a new task arrival $k_m(t)$, then it will choose actions for task $k_m(t)$: (a) whether to process the task locally or offload it to an edge node, i.e., $x_m(t)$; (b) which edge node the task is offloaded to, i.e., $\boldsymbol{y}_m(t)$. Hence, the action of device m in time slot t is represented by the following action vector:

$$\boldsymbol{a}_m(t) = (x_m(t), \boldsymbol{y}_m(t)). \tag{14}$$

Let $\mathcal{A}$ denote the action space, i.e., $\mathcal{A} = \{0,1\}^{1+N}$.

3.3 Cost

If a task has been processed, then the delay of the task is the duration between the task arrival and the time when the

2. The queuing delay of a task at an edge node is unknown at the time when the offloading decision of the task is to be made, due to the unknown load dynamics at the edge node. Thus, we did not include it as state information. In addition, the operation of our proposed algorithm does not rely on such queuing delay at the edge node.

3. We have assumed that edge nodes send the number of active queues by broadcast in each time slot. Hence, a mobile device can compute the number of its bits processed by an edge node in each time slot. It can compute the number of bits from its tasks dropped by an edge node based on the deadline of those tasks.

task has been processed.[4] Let $\text{Delay}_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t))$ (in time slots) denote the delay of task $k_m(t)$, given state $\boldsymbol{s}_m(t)$ and action $\boldsymbol{a}_m(t)$. For $m \in \mathcal{M}$ and $t \in \mathcal{T}$, if $x_m(t) = 1$, then

$$\text{Delay}_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t)) = l_m^{\text{comp}}(t) - t + 1; \tag{15}$$

if $x_m(t) = 0$, then

$$\text{Delay}_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t)) = \sum_{n \in \mathcal{N}} \sum_{t'=t}^{T} \mathbb{1}(k_{m,n}^{\text{edge}}(t') = k_m(t)) l_{m,n}^{\text{edge}}(t') - t + 1. \tag{16}$$

Specifically, consider task $k_m(t)$ arrived at the beginning of time slot t. If task $k_m(t)$ is placed in the computation queue for local processing, then $l_m^{\text{comp}}(t)$ is the time slot when the task has been processed. If task $k_m(t)$ is placed in the transmission queue for offloading, then $\sum_{n \in \mathcal{N}} \sum_{t'=t}^{T} \mathbb{1}(k_{m,n}^{\text{edge}}(t') = k_m(t)) l_{m,n}^{\text{edge}}(t')$ is the time slot when the task has been processed. This is because $\mathbb{1}(k_{m,n}^{\text{edge}}(t') = k_m(t)) = 1$ indicates that task $k_m(t)$ has arrived at the queue of edge node $n \in \mathcal{N}$ at the beginning of time slot t', and $l_{m,n}^{\text{edge}}(t')$ is the time slot when the task of device m arrived at edge node n at the beginning of time slot t' has been processed.

There is a cost $c_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t))$ associated with task $k_m(t)$. If task $k_m(t)$ has been processed, then

$$c_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t)) = \text{Delay}_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t)). \tag{17}$$

On the other hand, if task $k_m(t)$ has been dropped, then

$$c_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t)) = C, \tag{18}$$

where $C > 0$ is a constant penalty. Without loss of generality, if task $k_m(t) = 0$, then we set $c_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t)) = 0$. In the remaining part of this work, we use the short form $c_m(t)$ to denote $c_m(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t))$. Note that in practical systems, there may be other kinds of costs, such as energy consumption and the subscription fee charged by the edge nodes. These costs can be incorporated into this work by including the corresponding terms in Equations (17) and (18). As the proposed DRL-based algorithm is a model-free approach, it will still be applicable to the extended scenario.

3.4 Problem Formulation

A policy of device $m \in \mathcal{M}$ is a mapping from its state to its action, i.e., $\pi_m : \mathcal{S} \rightarrow \mathcal{A}$. Let $\gamma \in (0, 1]$ denote the discount factor that characterizes the discounted cost in the future. We aim to find the optimal policy π_m^* for each device m such that its expected long-term cost is minimized, i.e.,

4. The delay of a task includes the queuing delay, processing delay, and transmission delay (if the task has been offloaded). Instead of computing these delays separately and then summing them up, the mobile device can determine the delay of the task by computing the duration between the task arrival and when the task has been processed. This is reasonable because in practical systems, when a task has been processed, the mobile device knows both time instances.

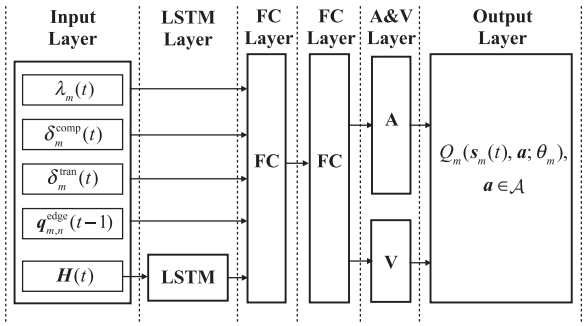


Fig. 2. The neural network of mobile device $m \in \mathcal{M}$ with parameter vector $\boldsymbol{\theta}_m$, which maps from state $\boldsymbol{s}_m(t) \in \mathcal{S}$ to the Q-value of each action $\boldsymbol{a} \in \mathcal{A}$.

$$\begin{aligned} \pi_m^* = \arg\underset{\pi_m}{\text{minimize}} \quad & \mathbb{E}\left[\sum_{t \in \mathcal{T}} \gamma^{t-1} c_m(t) \,\middle|\, \pi_m\right] \\ \text{subject to} \quad & \text{constraints } (1)-(6), (8)-(11), \\ & (15)-(18). \end{aligned} \tag{19}$$

The expectation $\mathbb{E}[\cdot]$ is with respect to the time-varying system parameters, e.g., the task arrivals and the computational requirements of the tasks of all mobile devices as well as the decisions of the mobile devices other than device m.

4 DRL-Based Offloading Algorithm

In this section, we propose a DRL-based offloading algorithm that enables the distributed offloading decision making of each mobile device. This algorithm is based on deep Q-learning [17]. As deep Q-learning is a model-free approach, the proposed algorithm can address the complicated system setting and interaction among the mobile devices without *a priori* knowledge of the system and interaction dynamics. Meanwhile, the proposed algorithm can handle the potentially large state space of the system.

In the proposed algorithm, each mobile device aims to learn a mapping from each state-action pair to a Q-value, which characterizes the expected long-term cost of the state-action pair. The mapping is determined by a neural network. Based on the mapping, each device can select the action inducing the minimum Q-value under its state to minimize its expected long-term cost. In the following, we present the neural network and the DRL-based algorithm, respectively.

4.1 Neural Network

The objective of the neural network is to find a mapping from each state to a set of Q-values of the actions. Fig. 2 shows an illustration of the neural network of mobile device $m \in \mathcal{M}$. Specifically, the state information is passed to the neural network through an input layer. Then, we use an LSTM layer to predict the load levels (at the edge nodes) in the near future based on the load level history. After that, the mapping from all the states (except the load level history) and the predicted load levels to the Q-values are learned through two fully-connected (FC) layers. Meanwhile, dueling DQN technique [25] is applied to improve the learning efficiency of the mapping from states to Q-values through an advantage and value (A&V) layer.

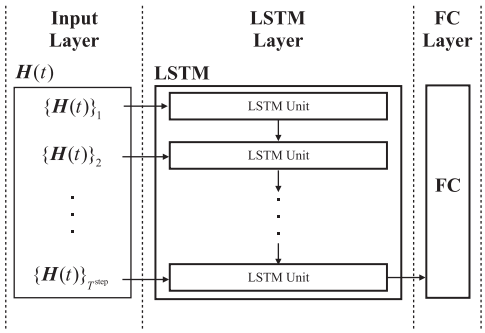


Fig. 3. An LSTM network with T^{step} LSTM units.

Finally, the Q-values of the actions are determined in the output layer. Let $\boldsymbol{\theta}_m$ denote the parameter vector of the neural network of device m, which includes the weights of all connections and the biases of all neurons from the input layer to the A&V layer. The details of each layer are as follows.

4.1.1 Input Layer

This layer is responsible for taking the state as input and passing them to the following layers. For mobile device $m \in \mathcal{M}$, the state information $\lambda_m(t)$, $\delta_m^{\text{comp}}(t)$, $\delta_m^{\text{tran}}(t)$, and $\boldsymbol{q}_m^{\text{edge}}(t-1)$ will be passed to the FC layer, and $\boldsymbol{H}(t)$ will be passed to the LSTM layer for load level prediction.

4.1.2 LSTM Layer

This layer is responsible for learning the dynamics of the load levels at edge nodes and predicting the load levels in the near future. This is achieved by including an LSTM network [26], which is a widely used approach for learning the temporal dependence of sequential observations and predicting the future variation of time series.[5]

Specifically, the LSTM network takes the matrix $\boldsymbol{H}(t)$ as input so as to learn the load level dynamics. Fig. 3 shows the structure of an LSTM network. The LSTM network contains T^{step} LSTM units, each of which contains a set of hidden neurons. Each LSTM unit takes one row of $\boldsymbol{H}(t)$ as input, we let $\{\boldsymbol{H}(t)\}_i$ denote the ith row of $\boldsymbol{H}(t)$ in Fig. 3. These LSTM units are connected in sequence so as to keep track of the variations of the sequences from $\{\boldsymbol{H}(t)\}_1$ to $\{\boldsymbol{H}(t)\}_{T^{\text{step}}}$, which can reveal the variations of the load levels at the edge nodes among time slots. The LSTM network will output the information that indicates the dynamics of the load levels in the future in the last LSTM unit, where the output will be passed to the next layer for further learning.

4.1.3 FC Layers

The two FC layers are responsible for learning the mapping from the state and the learned load level dynamics to the Q-values of the actions. Each FC layer contains a set of neurons with rectified linear unit (ReLU), which are connected with the neurons in the previous and following layers.

5. In this work, we use the conventional LSTM network. This work can be extended by applying variants of LSTM network (e.g., gated recurrent units), which may further enhance the algorithm performance and reduce the computational complexity.

4.1.4 A&V Layer and Output Layer

The A&V layer and the output layer implement the dueling-DQN technique [25] and determine the Q-value of each action as output. The main idea of the dueling-DQN is to first separately learn a *state-value* (i.e., the portion of the Q-value resulting from the state) and *action-advantage values* (i.e., the portion of the Q-value resulting from the actions). It then uses the state-value and action-advantage values to determine the Q-values of state-action pairs. This technique can improve the estimation of the Q-values through separately evaluating the expected long-term cost resulting from a state and an action.

The A&V layer contains two networks, denoted by network A and network V (see Fig. 2). The network A contains an FC network, and it is responsible for learning the action-advantage value of each action $\boldsymbol{a} \in \mathcal{A}$. For mobile device $m \in \mathcal{M}$, let $A_m(\boldsymbol{s}_m(t), \boldsymbol{a}; \boldsymbol{\theta}_m)$ denote the action-advantage value of action $\boldsymbol{a}$ under state $\boldsymbol{s}_m(t) \in \mathcal{S}$ with network parameter vector $\boldsymbol{\theta}_m$. The network V contains an FC network, and it is responsible for learning the state-value. For mobile device m, let $V_m(\boldsymbol{s}_m(t); \boldsymbol{\theta}_m)$ denote state-value of state $\boldsymbol{s}_m(t)$ with network parameter vector $\boldsymbol{\theta}_m$. The values of $A_m(\boldsymbol{s}_m(t), \boldsymbol{a}; \boldsymbol{\theta}_m)$ and $V_m(\boldsymbol{s}_m(t); \boldsymbol{\theta}_m)$ are determined by the parameter vector $\boldsymbol{\theta}_m$ and the neural network structure from the input layer to the A&V layer, where vector $\boldsymbol{\theta}_m$ is adjustable and will be trained in the DRL-based algorithm.

Based on the A&V layer, for mobile device $m \in \mathcal{M}$, the resulting Q-value of action $\boldsymbol{a} \in \mathcal{A}$ under state $\boldsymbol{s}_m(t) \in \mathcal{S}$ in the output layer is given as follows [25]:

$$\begin{aligned} Q_m(\boldsymbol{s}_m(t), \boldsymbol{a}; \boldsymbol{\theta}_m) =& V_m(\boldsymbol{s}_m(t); \boldsymbol{\theta}_m) + \Bigg(A_m(\boldsymbol{s}_m(t), \boldsymbol{a}; \boldsymbol{\theta}_m) \\ & - \frac{1}{|\mathcal{A}|} \sum_{\boldsymbol{a}' \in \mathcal{A}} A_m(\boldsymbol{s}_m(t), \boldsymbol{a}'; \boldsymbol{\theta}_m) \Bigg), \end{aligned} \tag{20}$$

which is the sum of the state-value under the corresponding state and the additional action-advantage value of the corresponding action (with respect to the average action-advantage value under the state over all actions).

4.2 DRL-Based Algorithm

In our proposed DRL-based algorithm, we let edge nodes help mobile devices to train the neural network to alleviate the computational loads at the mobile devices. Specifically, for each mobile device $m \in \mathcal{M}$, there is an edge node $n_m \in \mathcal{N}$ which helps device m with the training. This edge node n_m can be the edge node that has the maximum transmission capacity with mobile device m. For presentation convenience, let $\mathcal{M}_n \subset \mathcal{M}$ denote the set of mobile devices whose training is performed by edge node $n \in \mathcal{N}$, i.e., $M_n = \{\widetilde{m} | \widetilde{n}_m = n, m \in M\}$.

The DRL-based algorithms to be executed at mobile device $m \in \mathcal{M}$ and edge node $n \in \mathcal{N}$ are given in Algorithms 1 and 2, respectively. The key idea of the algorithms is to train the neural network using the experience (i.e., state, action, cost, and next state) of the mobile device to obtain the mapping from state-action pairs to Q-values, based on which the mobile device can select the action leading to the minimum Q-value under the observed state to minimize its expected long-term cost.

Algorithm 1. DRL-Based Algorithm at Device $m \in \mathcal{M}$

for episode = $1, 2, \ldots, E$ **do**
  Initialize $\boldsymbol{s}_m(1)$;
  for time slot $t \in \mathcal{T}$ **do**
    if device m has a new task arrival $k_m(t)$ **then**
      Send a parameter_request to edge node n_m;
      Receive network parameter vector $\boldsymbol{\theta}_m$;
      Select an action $\boldsymbol{a}_m(t)$ according to (22);
    end if
    Observe the next state $\boldsymbol{s}_m(t+1)$;
    Observe a set of costs $\{c_m(t'), t' \in \widetilde{\mathcal{T}}_{m,t}\}$;
    for each task $k_m(t')$ with $t' \in \widetilde{\mathcal{T}}_{m,t}$ **do**
      Send $(\boldsymbol{s}_m(t'), \boldsymbol{a}_m(t'), c_m(t'), \boldsymbol{s}_m(t'+1))$ to n_m;
    end for
  end for
end for

In the DRL-based algorithm, the edge node $n \in \mathcal{N}$ maintains a replay memory D_m for device $m \in \mathcal{M}_n$. The replay memory D_m stores the observed experience $(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ of mobile device m for some $t \in \mathcal{T}$, where we refer $(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ as experience t of mobile device m. Meanwhile, the edge node $n \in \mathcal{N}$ maintains two neural networks for device $m \in \mathcal{M}_n$, including an evaluation network, denoted by *Net*$_m$, and a target network, denoted by *Target_Net*$_m$. The evaluation network *Net*$_m$ is used for action selection. The target network *Target_Net*$_m$ is used for characterizing a target Q-value, which approximates the expected long-term cost of an action under the observed state. This target Q-value will be used for updating *Net*$_m$ by minimizing the difference between the Q-value under *Net*$_m$ and the target Q-value. Note that both *Net*$_m$ and *Target_Net*$_m$ have the same neural network structure, as presented in Section 4.1, while they have different network parameter vectors $\boldsymbol{\theta}_m$ and $\boldsymbol{\theta}_m^-$, respectively. Hence, the Q-values of *Net*$_m$ and *Target_Net*$_m$ are represented by $Q_m(\boldsymbol{s}_m\ (t), \boldsymbol{a}; \boldsymbol{\theta}_m)$ and $Q_m(\boldsymbol{s}_m(t), \boldsymbol{a}; \boldsymbol{\theta}_m^-)$ under observed state $\boldsymbol{s}_m(t) \in \mathcal{S}$ and action $\boldsymbol{a} \in \mathcal{A}$, respectively. The initialization of the replay memory D_m and two neural networks are given in steps 1−3 in Algorithm 2.

4.2.1 Algorithm 1 at Mobile Device $m \in \mathcal{M}$

We consider multiple episodes, where E denotes the number of episodes. At the beginning of each episode, mobile device $m \in \mathcal{M}$ initializes the state, i.e.,

$$\boldsymbol{s}_m(1) = (\lambda_m(1), \delta_m^{\text{comp}}(1), \delta_m^{\text{tran}}(1), \boldsymbol{q}_m^{\text{edge}}(0), \boldsymbol{H}(1)), \tag{21}$$

where we set $q_{m,n}^{\text{edge}}(0) = 0$ for all $n \in \mathcal{N}$, and $\boldsymbol{H}(1)$ is a zero matrix with size $T^{\text{step}} \times N$. At the beginning of time slot $t \in \mathcal{T}$, if mobile device m has a new task arrival $k_m(t)$, then it can send a parameter_request to edge node n_m in order to update the parameter vector $\boldsymbol{\theta}_m$ that it uses for making the task offloading decision. To reduce the communication overhead and address the potential scalability issue due to the neural network parameter transmission, the mobile device may not request the parameter vector in every time slot when it has a new task arrival. That is, steps 5-6 in Algorithm 1 can be omitted in some time slots. Intuitively, when the frequency that a mobile device requests the parameter vector is small, the communication overhead will also be small. However, it may reduce the rate of convergence of the proposed algorithm. We empirically evaluate how such a frequency affects the algorithm convergence in Section 5.1.

Algorithm 2. DRL-Based Algorithm at Edge Node $n \in \mathcal{N}$

1: Initialize replay memory D_m for $m \in \mathcal{M}_n$ and Count := 0;
2: Initialize *Net*$_m$ with random $\boldsymbol{\theta}_m$ for $m \in \mathcal{M}_n$;
3: Initialize *Target_Net*$_m$ with random $\boldsymbol{\theta}_m^-$ for $m \in \mathcal{M}_n$;
4: **while** True **do**
5: **if** receive a parameter_request from $m \in \mathcal{M}_n$ **then**
6: Send $\boldsymbol{\theta}_m$ to device m;
7: **end if**
8: **if** receive an experience $(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ from $m \in \mathcal{M}_n$ and Converge_Indicator = 0 **then**
9: Store $(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ in D_m;
10: Sample a set of experiences (denoted by $\mathcal{I}$) from D_m;
11: **for** each experience $i \in \mathcal{I}$ **do**
12: Obtain experience $(\boldsymbol{s}_m(i), \boldsymbol{a}_m(i), c_m(i), \boldsymbol{s}_m(i+1))$;
13: Compute $\hat{Q}_{m,i}^{\text{Target}}$ according to (26);
14: **end for**
15: Set vector $\hat{\boldsymbol{Q}}_m^{\text{Target}} := (\hat{Q}_{m,i}^{\text{Target}}, i \in \mathcal{I})$;
16: Update $\boldsymbol{\theta}_m$ to minimize $L(\boldsymbol{\theta}_m, \hat{\boldsymbol{Q}}_m^{\text{Target}})$ in (24);
17: Count := Count + 1;
18: **if** mod(Count, Replace_Threshold) = 0 **then**
19: $\boldsymbol{\theta}_m^- := \boldsymbol{\theta}_m$;
20: **end if**
21: **end if**
22: **end while**

Based on the parameter vector $\boldsymbol{\theta}_m$ of *Net*$_m$, mobile device m will choose its action for task $k_m(t)$ as follows:

$$\boldsymbol{a}_m(t) = \begin{cases} \text{a random action from } \mathcal{A}, & \text{w.p. } \epsilon, \\ \arg\min_{\boldsymbol{a} \in \mathcal{A}} Q_m(\boldsymbol{s}_m(t), \boldsymbol{a}; \boldsymbol{\theta}_m), & \text{w.p. } 1-\epsilon, \end{cases} \tag{22}$$

where 'w.p.' is the short-form for "with probability", and ϵ is the probability of random exploration. Intuitively, with probability $1-\epsilon$, the device chooses the action that leads to the minimum Q-value under state $\boldsymbol{s}_m(t)$ based on *Net*$_m$.

At the beginning of the next time slot (i.e., time slot $t+1$), mobile device m observes the next state $\boldsymbol{s}_m(t+1)$. On the other hand, as the processing and the transmission of a task may continue for multiple time slots, the cost $c_m(t)$, which depends on the delay of task $k_m(t)$, may not be observed at the beginning of time slot $t+1$. Instead, mobile device m may observe a set of costs belonging to some tasks $k_m(t')$ with time slot $t' \leq t$. To address this, for device m, we define $\widetilde{\mathcal{T}}_{m,t} \subset \mathcal{T}$ as the set of time slots such that each task $k_m(t')$ associated with time slot $t' \in \widetilde{\mathcal{T}}_{m,t}$ has been processed or dropped in time slot t. Set $\widetilde{\mathcal{T}}_{m,t}$ is defined as follows:

$$\widetilde{\mathcal{T}}_{m,t} = \Bigg\{ t' \;\Bigg|\; t' = 1, 2, \ldots, t,\ \lambda_m(t') > 0,\ l_m^{\text{comp}}(t') = t \text{ or } \sum_{n \in \mathcal{N}} \sum_{i=t'}^{t} \mathbb{1}(k_{m,n}^{\text{edge}}(i) = k_m(t')) l_{m,n}^{\text{edge}}(i) = t \Bigg\}. \tag{23}$$

In (23), $\lambda_m(t') > 0$ implies that there is a newly arrived task $k_m(t')$ in time slot t'. Specifically, set $\widetilde{\mathcal{T}}_{m,t}$ contains a time slot $t' \in \{1, 2, \ldots, t\}$ if task $k_m(t')$ has been processed or dropped

in time slot t. Hence, at the beginning of time slot $t+1$, mobile device m can observe a set of costs $\{c_m(t'), t' \in \widetilde{\mathcal{T}}_{m,t}\}$, where set $\widetilde{\mathcal{T}}_{m,t}$ can be an empty set for some $m \in \mathcal{M}$ and $t \in \mathcal{T}$. Then, for each task $k_m(t')$ with $t' \in \widetilde{\mathcal{T}}_{m,t}$, device m sends its experience $(\boldsymbol{s}_m(t'), \boldsymbol{a}_m(t'), c_m(t'), \boldsymbol{s}_m(t'+1))$ to edge node n_m. Note that when device m transmits state information $\boldsymbol{s}_m(t')$ and $\boldsymbol{s}_m(t'+1)$, it does not need to send the load level history $\boldsymbol{H}(t')$ and $\boldsymbol{H}(t'+1)$. This is because the number of active queues at the edge nodes in each time slot has been sent by broadcast. Thus, each edge node knows the load level history at all edge nodes during the past time slots.[6] In this case, when an edge node receives experience $(\boldsymbol{s}_m(t'), \boldsymbol{a}_m(t'), c_m(t'), \boldsymbol{s}_m(t'+1))$, it can include $\boldsymbol{H}(t')$ and $\boldsymbol{H}(t'+1)$ to the experience for training.

4.2.2 Algorithm 2 at Edge Node $n \in \mathcal{N}$

After initializing the replay memory D_m as well as the neural networks Net_m and $Target\_Net_m$ for device $m \in \mathcal{M}_n$, edge node $n \in \mathcal{N}$ will wait for the request messages from the mobile devices in set $\mathcal{M}_n$. If edge node n receives a parameter_request from mobile device $m \in \mathcal{M}_n$, then it will send the current parameter vector $\boldsymbol{\theta}_m$ of Net_m to device m. On the other hand, if edge node n receives an experience $(\boldsymbol{s}_m(t), \boldsymbol{a}_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ from mobile device $m \in \mathcal{M}_n$, then it will store the experience in memory D_m. The reply of the parameter vector (steps 5-7 in Algorithm 2) and the training of the network (steps 8-21 in Algorithm 2) can be operated in parallel. That is, when an edge node receives the parameter_request from mobile device m, it will immediately send the current vector $\boldsymbol{\theta}_m$ to the device regardless of whether there is training in progress or not.

The edge node will train the neural network (in steps 10−20 in Algorithm 2) to update the parameter vector $\boldsymbol{\theta}_m$ of Net_m as follows. The edge node will randomly sample a set of experiences from the memory (in step 10), denoted by $\mathcal{I}$. Let $|\mathcal{I}|$ denote the number of experiences in set $\mathcal{I}$.[7] Based on these experience samples, the key idea of the update of Net_m is to minimize the difference between the Q-values under Net_m and the target Q-values computed based on the experience samples under $Target\_Net_m$. Specifically, for the experience samples in set $\mathcal{I}$, the edge node will compute $\hat{\boldsymbol{Q}}_m^{\text{Target}} = (\hat{Q}_{m,i}^{\text{Target}}, i \in \mathcal{I})$ and update $\boldsymbol{\theta}_m$ in Net_m by minimizing the following loss function:

$$L(\boldsymbol{\theta}_m, \hat{\boldsymbol{Q}}_m^{\text{Target}}) = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \left(Q_m(\boldsymbol{s}_m(i), \boldsymbol{a}_m(i); \boldsymbol{\theta}_m) - \hat{Q}_{m,i}^{\text{Target}} \right)^2. \tag{24}$$

Loss function (24) characterizes the gap between the Q-value of action $\boldsymbol{a}_m(i)$ given state $\boldsymbol{s}_m(i)$ under the current network parameter vector $\boldsymbol{\theta}_m$ and a target Q-value $\hat{Q}_{m,i}^{\text{Target}}$ for each experience $i \in \mathcal{I}$ (to be explained in the next paragraph). The minimization of the loss function is accomplished by performing backpropagation (see Section 6 in [27]) on the neural network using iterative optimization algorithms such as gradient descent algorithm.

The target Q-value $\hat{Q}_{m,i}^{\text{Target}}$ for experience $i \in \mathcal{I}$ is determined based on double-DQN technique [28], which can improve the estimation of the expected long-term cost when compared with the traditional method (e.g., [17]). To derive this target Q-value, let $\boldsymbol{a}_i^{\text{Next}}$ denote the action with the minimum Q-value given state $\boldsymbol{s}_m(i+1)$ under Net_m, i.e.,

$$\boldsymbol{a}_i^{\text{Next}} = \arg\min_{\boldsymbol{a} \in \mathcal{A}} Q_m(\boldsymbol{s}_m(i+1), \boldsymbol{a}; \boldsymbol{\theta}_m). \tag{25}$$

The value of $\hat{Q}_{m,i}^{\text{Target}}$ for experience i is derived as follows:

$$\hat{Q}_{m,i}^{\text{Target}} = c_m(i) + \gamma Q_m(\boldsymbol{s}_m(i+1), \boldsymbol{a}_i^{\text{Next}}; \boldsymbol{\theta}_m^-). \tag{26}$$

Intuitively, target-Q value $\hat{Q}_{m,i}^{\text{Target}}$ reveals the expected long-term cost of action $\boldsymbol{a}_m(i)$ given state $\boldsymbol{s}_m(i)$, i.e., the sum of the cost in experience i and a discounted Q-value of the action that is likely to be selected given the next state in experience i under network $Target\_Net_m$.

Let Replace_Threshold denote the number of training rounds after which $Target\_Net_m$ has to be updated. That is, for every Replace_Threshold training rounds, $Target\_Net_m$ has to be updated by copying the parameter vector of Net_m, where mod(·) is the modulo operator (in step 18 in Algorithm 2). The objective of this step is to keep the network parameter $\boldsymbol{\theta}_m^-$ in $Target\_Net_m$ up-to-date, so that it can better approximate the expected long-term cost in the computing of the target Q-values in (26).

4.2.3 Computational Complexity and Convergence

To determine the computational complexity, let L denote the number of multiplication operations in the neural network. The computational complexity of backpropagation for the training of one experience is $\mathcal{O}(L)$. Recall that $|\mathcal{I}|$ is the number of experiences sampled in each round of training. Let K denote the expected number of tasks in each episode. Since there are E episodes, the computational complexity of the proposed algorithm is $\mathcal{O}(LKE|\mathcal{I}|)$.

Regarding the convergence, as mentioned in many existing works (e.g., [29]), the convergence guarantee of a DRL-algorithm is still an open problem. Despite the fact that the convergence of a reinforcement learning algorithm can be proven, a DRL algorithm requires function approximation (e.g., the approximation of the Q-values in deep Q-learning algorithm) using neural networks, under which the convergence may no longer be guaranteed. In this work, we empirically evaluate the convergence performance of the proposed algorithm in Section 5.1.

5 Performance Evaluation

We consider a scenario with 50 mobile devices and five edge nodes. The parameter settings are given in Table 1. The neural network settings are as follows. The batch size is set to 16. The learning rate is equal to 0.001, and the discount factor is equal to 0.9. The probability of random exploration is gradually decreasing from 1 to 0.01. Meanwhile, we use RMSProp optimizer. In these simulations, we focus on a scenario with stationary environment, i.e., the transition function (from the

6. As we focus on addressing the load level dynamics at the edge nodes, we consider a scenario where the edge nodes are located in neighboring areas. We assume that any two edge nodes are within one hop from each other. In the scenario where this assumption does not hold, an edge node can send the number of active queues to other edge nodes through backhaul links.

7. When all other factors are fixed, a smaller batch size (i.e., the number of experiences sampled in each round) incurs a shorter time for one round of training.

TABLE 1
Parameter Settings

Parameter	Value
Δ	0.1 second
$f_m^{\text{device}}, m \in \mathcal{M}$	2.5 GHz [15]
$f_n^{\text{edge}}, n \in \mathcal{N}$	41.8 GHz [15]
$r_{m,n}^{\text{tran}}, m \in \mathcal{M}, n \in \mathcal{N}$	14 Mbps [30]
$\lambda_m(t), m \in \mathcal{M}, t \in \mathcal{T}$	{2.0, 2.1, ..., 5.0} Mbits [4]
$\rho_m, m \in \mathcal{M}$	0.297 gigacycles per Mbits [4]
$\tau_m, m \in \mathcal{M}$	10 time slots (i.e., 1 second)
Task arrival probability	0.3

state and action to the next state) and the cost function (from the state and action to cost) do not vary across time. Under a non-stationary environment, if the environment has changed, then the proposed algorithm can adapt to it by resetting the probability of random exploration to be one so as to enable the random exploration again.

5.1 Performance and Convergence

The neural network in the proposed algorithm is trained online, where the real-time collected experience is used to train the neural network and update the task offloading decision. We evaluate the convergence of the proposed algorithm under different neural network hyperparameters and algorithm settings. We consider 1,000 episodes, where each episode has 100 time slots. The simulation results are shown in Fig. 4. In the subfigures, the x-axis shows the episode, and the y-axis shows the average cost among the mobile devices and the time slots in each episode. We plot the performance of the proposed algorithm under different settings and the random policy (denoted by "Rand."), where the actions are randomly selected.

Fig. 4a shows the convergence of the proposed algorithm under different values of learning rate (denoted "lr"), where the learning rate is the step size in each iteration for moving towards the minimum of the loss function. In Fig. 4a, lr $= 10^{-3}$ leads to a relatively fast convergence and small converged cost. When the learning rate is small (i.e., 10^{-4}), the convergence is slow. When the learning rate is large (i.e., 10^{-2}, 10^{-1}), the converged cost increases, which may be even higher than that of the random policy.

Fig. 4b shows the algorithm performance under different batch sizes, i.e., the number of experiences sampled in each training round. As the batch size increases from 2 to 8, the convergence speed increases. As it further increases from 8 to 32, the performance of the proposed algorithm does not have a significant improvement in terms of the convergence speed and the converged result. Thus, we can choose a small batch size (e.g., 8) to reduce the time for one round of training without significantly reducing the performance of the proposed algorithm.

Fig. 4c shows the algorithm performance under different optimizers, including gradient descent (denoted by "GD"), RMSProp, and adaptive moment estimation (Adam) optimizers. These optimizers provide different approaches to update the neural network for minimizing the loss function in (24). As shown in Fig. 4c, RMSProp and Adam optimizers lead to similar convergence speed and result.

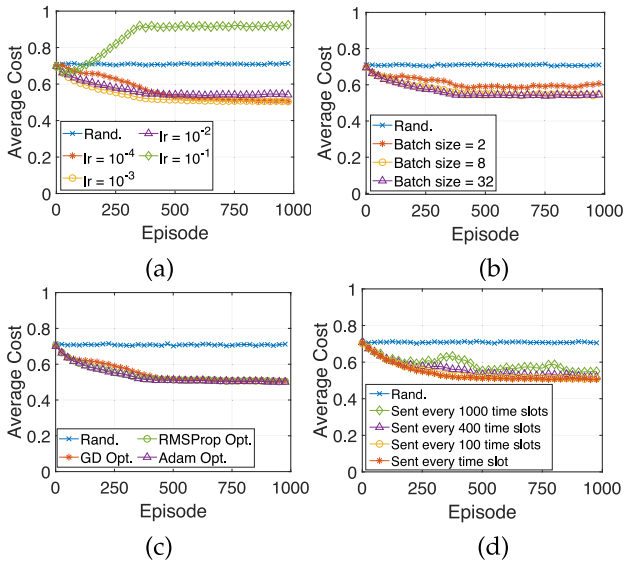


Fig. 4. Convergence of the proposed algorithm under different: (a) learning rate; (b) batch size; (c) optimizer; (d) the frequency that a mobile device sends a parameter_request.

In Fig. 4d, we consider a setting where a mobile device sends the parameter_request to update its network parameter every fixed number of time slots (instead of every time slot when the mobile device has a task arrival). As shown in the figure, sending the request every 100 time slots does not have much impact on the algorithm performance when compared with sending it every time slot. This is because the training of the neural network in the proposed algorithm is based on randomly sampled experiences from experience replay rather than newly obtained experience. Hence, the algorithm can tolerate a certain degree of delay in terms of the update of the neural network for action selection. As a result, in order to reduce the communication overhead, we can reduce the frequency that the parameter_request is sent without significantly affecting the algorithm performance.

5.2 Method Comparison

We compare our proposed DRL-based method with several benchmark methods, including no offloading (denoted by No Offl.), random offloading (denoted by R. Offl.), PGOA in [14], and ULOOF in [15]. The PGOA is designed based on the best response algorithm for the potential game, which considers the strategic interaction among mobile devices.[8] The ULOOF is designed based on the capacity estimation according to historical observations. We choose PGOA [14] and ULOOF [15] for comparison because similar to our work, those schemes considered non-divisible tasks and multiple edge nodes, and they did not consider the involvement of any centralized entity. We consider two

8. PGOA operates under the assumption that the processing of each task can be finished within each time slot. Hence, the offloading decision of each task can be made based on the feedback (e.g., delay) of the tasks arrived in the previous time slots. In our work, we do not impose this assumption. To evaluate the performance, we consider the offloading decision of each task is made based on the feedback of the tasks that have been processed in the previous time slots.

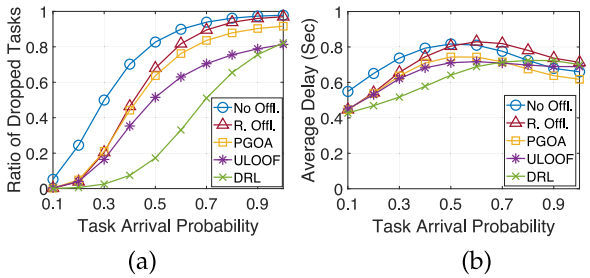


Fig. 5. Performance evaluation under different task arrival probabilities: (a) ratio of dropped tasks; (b) average delay.

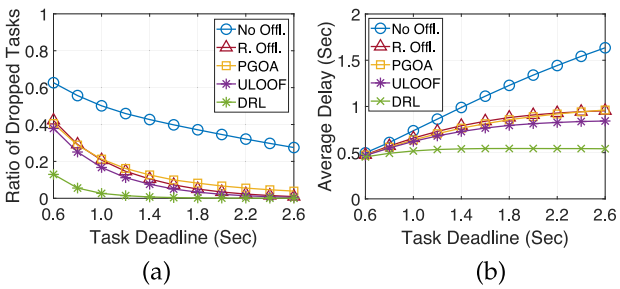


Fig. 6. Performance under different task deadlines: (a) ratio of dropped tasks; (b) average delay.

performance metrics: the ratio of dropped tasks (i.e., the ratio of the number of dropped tasks to the number of total task arrivals) and the average delay (i.e., the average delay of the tasks which have been processed).

In Fig. 5a, as the task arrival probability increases, the proposed DRL-based algorithm can always maintain a lower ratio of dropped tasks when compared with the benchmark methods. When the task arrival probability is small (i.e., 0.1), most of the methods can achieve a ratio of dropped tasks of around zero. As the task arrival probability increases from 0.1 to 0.5, the ratio of dropped tasks of the proposed algorithm remains less than 0.2, while those of the benchmark methods increase to more than 0.5. In Fig. 5b, as the task arrival probability increases from 0.1 to 0.4, the average delay of our proposed DRL-based algorithm increases by 26.1 percent, while those of the benchmark methods increase by at least 34.5 percent. This implies that as the load of the system increases, the average delay of the proposed algorithm increases less dramatically than those of the benchmark methods. As the task arrival probability increases to 0.6, the average delay of some of the methods decrease, because an increasing number of tasks are dropped and hence are not accounted in the average delay. For the same reason, when the load of the system is high, the proposed algorithm may have a larger average delay than the other methods, as it has less tasks dropped.

As shown in Fig. 6a, the proposed algorithm always achieves a lower ratio of dropped tasks than the benchmark methods, especially when the deadline is small. When the task deadline is 0.6 second, the proposed algorithm reduces the ratio of dropped tasks by 65.8-79.3 percent when compared with the benchmark methods. In Fig. 6b, as the task deadline increases, the average delay of each method increases and gradually converges. This is because when the deadline is larger, the tasks requiring longer processing (and transmission) time can be processed and are accounted in the average delay. When the deadline is large enough, no task is dropped, so further increasing the deadline makes no difference. As shown in Fig. 6b, the average delay of the proposed algorithm converges (i.e., achieves a marginal increase of less than 0.05) after the deadline increases to 1.4 seconds, and the converged average delay is around 0.54 second. In comparison, the converged average delay of the other methods are larger than 0.84 second.

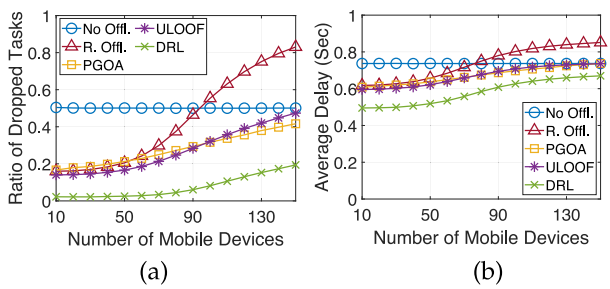


Fig. 7. Performance under different number of mobile devices: (a) ratio of dropped tasks; (b) average delay.

In Fig. 7a, the proposed algorithm achieves a lower ratio of dropped tasks than the other methods, especially when the number of devices is large. This is because the proposed algorithm can effectively address the unknown load dynamics at the edge nodes. When the number of mobile devices increases to 80, the proposed algorithm maintains a ratio of dropped tasks of less than 0.05. In Fig. 7b, as the number of mobile devices increases, the average delay of each method (except no offloading) increases due to the potentially increasing load at the edge nodes. Since the proposed algorithm can effectively deal with the unknown edge load dynamics, when the number of mobile devices increases to 150, it achieves an average delay of 9.0 percent lower than those of PGOA and ULOOF.

6 Conclusion

In this work, we studied the computational task offloading problem with non-divisible and delay-sensitive tasks in the MEC system. We designed a distributed offloading algorithm that enables mobile devices to make their offloading decisions in a decentralized manner, which can address the unknown load level dynamics at the edge nodes. Simulation results showed that when compared with several benchmark methods, our proposed algorithm can reduce the ratio of dropped tasks and average delay. The benefit is especially significant when the tasks are delay-sensitive or the load levels at the edge nodes are high.

There are several directions to extend this work. First, it is interesting to extend the simplified wireless network model to incorporate the transmission error and the interference among mobile devices. Second, the algorithm performance can be evaluated in a demo system, under which many practical issues (e.g., real computational tasks) should be addressed. Third, the computational complexity of the proposed algorithm can be reduced. This can be achieved by incorporating deep compression [31] to reduce the number of multiplication operations in the neural network and incorporating transfer learning [32] to accelerate the

convergence. In addition, as we let edge nodes help mobile devices to train the neural networks, the communication overhead and scalability issue due to the neural network parameter transmission may be a concern when the neural network is large. To extend this work, deep compression [31] can be applied to reduce the number of weights of the neural network and the number of bits required to represent each weight. Furthermore, to enhance the performance of the proposed algorithm under non-stationary environment, it is interesting to incorporate techniques such as the reinforcement learning for non-stationary environment (e.g., [33]) and lifelong reinforcement learning [34]. Last but not least, game-theoretic and multi-agent reinforcement learning techniques can be applied to further understand the strategic interactions among mobile devices. Those techniques can be incorporated into the proposed algorithm to further address the load level dynamics at the edge nodes.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, Fourthquarter 2017.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st Ed. MCC Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.

[3] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for Internet of Things realization," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 2961–2991, Fourthquarter 2018.

[4] C. Wang, C. Liang, F. R. Yu, Q. Chen, and L. Tang, "Computation offloading and resource allocation in wireless cellular networks with mobile edge computing," *IEEE Trans. Wireless Commun.*, vol. 16, no. 8, pp. 4924–4938, Aug. 2017.

[5] S. Bi and Y. J. Zhang, "Computation rate maximization for wireless powered mobile-edge computing with binary computation offloading," *IEEE Trans. Wireless Commun.*, vol. 17, no. 6, pp. 4177–4190, Jun. 2018.

[6] N. Eshraghi and B. Liang, "Joint offloading decision and resource allocation with uncertain task computing requirement," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1414–1422.

[7] X. Lyu, H. Tian, W. Ni, Y. Zhang, P. Zhang, and R. P. Liu, "Energy-efficient admission of delay-sensitive tasks for mobile edge computing," *IEEE Trans. Commun.*, vol. 66, no. 6, pp. 2603–2616, Jun. 2018.

[8] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 587–597, Mar. 2018.

[9] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 10–18.

[10] X. Lyu *et al.*, "Distributed online optimization of fog computing for selfish devices with out-of-date information," *IEEE Trans. Wireless Commun.*, vol. 17, no. 11, pp. 7704–7717, Nov. 2018.

[11] L. Li, T. Q. S. Quek, J. Ren, H. H. Yang, Z. Chen, and Y. Zhang, "An incentive-aware job offloading control framework for multi-access edge computing," *IEEE Trans. Mobile Comput.*, early access, Sep. 17, 2019, doi: 10.1109/TMC.2019.2941934.

[12] H. Shah-Mansouri and V. W. S. Wong, "Hierarchical fog-cloud computing for IoT systems: A computation offloading game," *IEEE Internet Things J.*, vol. 5, no. 4, pp. 3246–3257, Aug. 2018.

[13] S. Jošilo and G. Dán, "Wireless and computing resource allocation for selfish computation offloading in edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 2467–2475.

[14] L. Yang, H. Zhang, X. Li, H. Ji, and V. Leung, "A distributed computation offloading strategy in small-cell networks integrated with mobile edge computing," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2762–2773, Dec. 2018.

[15] J. L. D. Neto, S.-Y. Yu, D. F. Macedo, M. S. Nogueira, R. Langar, and S. Secci, "ULOOF: A user level online offloading framework for mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 17, no. 11, pp. 2660–2674, Nov. 2018.

[16] G. Lee, W. Saad, and M. Bennis, "An online optimization framework for distributed fog network formation with minimal latency," *IEEE Trans. Wireless Commun.*, vol. 18, no. 4, pp. 2244–2258, Apr. 2019.

[17] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

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

[19] J. Luo, F. R. Yu, Q. Chen, and L. Tang, "Adaptive video streaming with edge caching and video transcoding over software-defined mobile networks: A deep reinforcement learning approach," *IEEE Trans. Wireless Commun.*, vol. 19, no. 3, pp. 1577–1592, Mar. 2020.

[20] Y. Liu, H. Yu, S. Xie, and Y. Zhang, "Deep reinforcement learning for offloading and resource allocation in vehicle edge computing and networks," *IEEE Trans. Veh. Technol.*, vol. 68, no. 11, pp. 11 158–11 168, Nov. 2019.

[21] N. Zhao, Y.-C. Liang, D. Niyato, Y. Pei, M. Wu, and Y. Jiang, "Deep reinforcement learning for user association and resource allocation in heterogeneous cellular networks," *IEEE Trans. Wireless Commun.*, vol. 18, no. 11, pp. 5141–5152, Nov. 2019.

[22] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *Proc. IEEE Int. Symp. Inf. Theory*, 2016, pp. 1451–1455.

[23] Y. Mao, J. Zhang, S. Song, and K. B. Letaief, "Stochastic joint radio and computational resource management for multi-user mobile-edge computing systems," *IEEE Trans. Wireless Commun.*, vol. 16, no. 9, pp. 5994–6009, Sep. 2017.

[24] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, no. 3, pp. 344–357, Jun. 1993.

[25] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1995–2003.

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

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[28] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 2094–2100.

[29] H. Van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, "Deep reinforcement learning and the deadly triad," 2018, *arXiv:1812.02648v1*[cs.AI].

[30] Speedtest intelligence, "Speedtest market reports: Canada average mobile upload speed based on Q2-Q3 2019 data," Accessed: Aug. 5, 2020. [Online]. Available: https://www.speedtest.net/reports/canada/

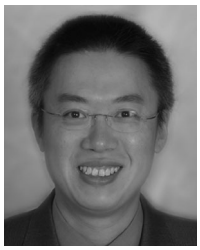
[31] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2016, *arXiv:1510.00149v5*[cs.CV].

[32] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *J. Big Data*, vol. 3, no. 1, pp. 1345–1359, May 2016.

[33] E. Lecarpentier and E. Rachelson, "Non-stationary markov decision processes, a worst-case approach using model-based reinforcement learning," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 7214–7223.

[34] D. Abel, Y. Jinnai, S. Y. Guo, G. Konidaris, and M. Littman, "Policy and value transfer in lifelong reinforcement learning," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 20–29.

Ming Tang (Member, IEEE) received the PhD degree from the Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong, China, in 2018. She is currently a postdoctoral research fellow with the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada. Her research interests include mobile networking and network economics.



Vincent W.S. Wong (Fellow, IEEE) received the BSc degree from the University of Manitoba, Winnipeg, MB, Canada, in 1994, the MASc degree from the University of Waterloo, Waterloo, ON, Canada, in 1996, and the PhD degree from the University of British Columbia (UBC), Vancouver, BC, Canada, in 2000. From 2000 to 2001, he worked as a systems engineer at PMC-Sierra Inc. (now Microchip Technology Inc.). He joined the Department of Electrical and Computer Engineering, University of British Columbia, Canada, in 2002 and is currently a professor. His research areas include protocol design, optimization, and resource management of communication networks, with applications to wireless networks, smart grid, mobile edge computing, and Internet of Things. Currently, he is an executive editorial committee member of the *IEEE Transactions on Wireless Communications*, an area editor of the *IEEE Transactions on Communications* and the *IEEE Open Journal of the Communications Society*, and an associate editor of the *IEEE Transactions on Mobile Computing*. He is a technical program co-chair of the *IEEE* 92*nd Vehicular Technology Conference (VTC*2020-*Fall)*. He has served as a guest editor of the *IEEE Journal on Selected Areas in Communications* and the *IEEE Wireless Communications*. He has also served on the editorial boards of the *IEEE Transactions on Vehicular Technology* and the *Journal of Communications and Networks*. He was a tutorial co-chair of *IEEE Globecom*'18, a technical program co-chair of *IEEE SmartGridComm*'14, as well as a symposium co-chair of *IEEE ICC*'18, *IEEE SmartGridComm* ('13, '17) and *IEEE Globecom*'13. He is the chair of the IEEE Vancouver Joint Communications Chapter and has served as the chair of the IEEE Communications Society Emerging Technical Sub-Committee on Smart Grid Communications. He is an IEEE Communications Society distinguished lecturer (2019 - 2020).