Dependent Task Offloading for Multiple Jobs in Edge Computing

Zhiqing Tang*‡, Jiong Lou*‡, Fuming Zhang*‡, and Weijia Jia†‡
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China 200240
Emails: {domain, lj1994, zhangfuming-alex}@sjtu.edu.cn
†BNU-UIC Joint AI Resrach Institute, Beijing Normal University & UIC (Zhuhai), Guangdong, China 519087
‡State Key Lab of IoT for Smart City, University of Macau, Macau, China 999078
Email: jiawj@um.edu.mo

***Abstract*—The dependent task offloading problem for one single job in edge computing (EC) has drawn attention widely. Unlike most existing approaches that only focus on a single job, we aim to solve the dependent task offloading problem for multiple jobs, which is more general in the real world. To solve this problem, we propose a deep reinforcement learning (DRL) based multi-job dependent task offloading algorithm. Specifically, 1) we model edge nodes, jobs, and tasks in a resource-limited EC scenario, where the dependent tasks of multiple jobs are offloaded to the nodes to be processed. Then we model the task offloading decision as a Markov decision process (MDP) problem to minimize the transmission cost and computation cost. 2) To represent the state space of MDP and to accelerate decision-making in EC, we propose a DRL-based algorithm with the aid of graph convolutional network (GCN) to extract the dependency information of different tasks and then improve the action selection process. 3) We conduct experiments with real-world trace, demonstrating our algorithm outperforms the baseline algorithms 13.78% on average in regarding to offloading cost.**

***Index Terms*—Dependent task offloading, deep reinforcement learning, multiple jobs, edge computing**

I. Introduction

With the development of mobile devices and the coming 5G, more and more applications can be processed on mobile devices [1]. Such mobile applications have high demands for computation resources, while existing mobile devices usually cannot meet their requirements. To obtain more computation resources, we can offload mobile tasks to the remote cloud data centers, which have massive computation resources but are far away from users [2]. The long transmission distance between the cloud and users will result in high delay, which makes it challenging to meet the tight end-to-end (i.e., motion-to-photon) delay of 20 - 25ms required for the delay-sensitive task, e.g., high-quality VR [3]. To better handle these tasks with lower delay, Edge Computing (EC) currently can play essential roles [4]. In EC, edge nodes with more computation resources are deployed closer to users, and the tasks from mobile users can be offloaded to these nodes.

The growing demand for computation resources in mobile applications leads to intensive resource consumption on the nodes. However, the computation capacity of each node is limited [5]. Meanwhile, the computation resources of many nodes are currently not effectively utilized due to the lack of efficient task scheduling [6]. Considering the limited computation resources of the nodes, how to make the mobile task offloading decisions more effective is an urgent issue to be solved. Besides, task requests from the same user can usually be split into a set of independent or dependent tasks, and these tasks are denoted as a job [7]. For example, a VR-type job consists of collecting sensor data, rendering new image frames, data compressing, transferring, and displaying, etc. [8]. These dependent tasks affiliated with the same job need to be executed sequentially according to the dependencies, i.e., subsequent tasks are required to wait for all of their predecessor tasks to complete execution. The dependencies between tasks are both constraints and factors that need to be considered when making offloading decisions [7], [9]. The offloading decisions bring about the computation cost of nodes. How to minimize the cost with dependencies among different tasks is also an issue worthy of attention.

Moreover, compared to a single dependent job, there is much more scheduling information for multiple dependent jobs. This leads to much larger problem sizes, including the state space (the information for decision-making) and action space (available choices for decision). Therefore, the challenge for dimension reduction is also much more severe. Besides, the randomness and heterogeneity of multiple jobs make the training process more difficult [10]. Finally, the scheduling of tasks of different jobs to the same edge node may cause mutual effects, such as resource preemption. Thus, how to deal with multiple dependent tasks in a large number of jobs to reduce the delay between users and nodes and the computation cost of nodes is another issue that needs to be solved urgently.

To solve these issues, this paper studies the dependent task offloading problem for multiple jobs under the resource-limited EC environment. There are many challenges in such an EC scenario. The first challenge is how to model the resource-limited EC reasonably. Zhang et al. [6] have studied the task scheduling problem under the resource-limited EC, but they did not consider the transmission cost and the dependencies between the tasks. In this paper, the service level agreement violation (SLAV) is used to indicate the degree of resource

Corresponding author: Weijia Jia.

978-1-7281-6607-0/20/$31.00 ©2020 IEEE

violation of each node [11]. Besides, the costs of tasks mainly include transmission cost and computation cost during the running period, and we aim to minimize the combination of these two costs [12].

Secondly, how to deal with the task dependencies in EC is also a challenge. Sundar et al. [7] studied the offloading problem of dependent tasks, but they only consider one single job which has limited applications in the real world. Mao et al. [10] considered multiple jobs with dependent tasks, but they did not consider the transmission cost between tasks, which is significant in EC. In this paper, the directed acyclic graph (DAG) is taken to describe the dependencies between tasks attached to each job. Each node of the DAG is a task, and each directed edge represents the order of task execution. To better represent the dependencies between tasks and the features of the tasks, graph convolutional network (GCN) [13] is adopted for embedding each job into a vector as a part of the information for decision-making. Besides, the entire environment is also embedded into a vector, including the features of all nodes, the running tasks, etc. The EC environment is then modeled based on the above information.

Finally, the offloading algorithm needs to support fast decision-making relying only on current and historical information in EC. Liu et al. [14] have proposed an online learning algorithm, but they do not consider the heterogeneity of EC and the dependencies of tasks. To this end, based on the EC environment described above, the task offloading problem is further modeled as a Markov decision process (MDP) problem. The action and reward of MDP are defined, and a deep reinforcement learning (DRL) [15] based task offloading algorithm which fulfills real-time decision-making is proposed. Furthermore, the action selection process is improved to make the algorithm converge faster.

To sum up, the contributions of our paper are as follows:

1) We model the resource-limited EC, which includes edge nodes, jobs, and tasks. The dependent tasks of multiple jobs are offloaded to the nodes to be processed. The task offloading decision process is modeled as an MDP problem to minimize the transmission cost and the computation cost.
2) To represent the state space of MDP and make it suitable for fast decision-making in EC, a DRL-based algorithm is proposed. In the algorithm, the GCN is used to extract the dependency information of different jobs to take the transmission between tasks into consideration. Besides, the action selection is improved for faster converge.
3) We conduct the experiments with real-world data-trace and compare the performance of our algorithm with some existing algorithms. The experimental results show that our algorithm can reduce 13.78% of the total offloading cost on average.

The remainder of this paper is organized as follows. In Section II, we model the EC and formulate the problem. Then, the dependent task offloading algorithm is devised in Section III. The experimental settings and results are described in Section IV. Finally, the paper is concluded in Section V.

II. System Model and Problem Formulation

In this section, first, the system model of EC is introduced in II-A, which includes the nodes, jobs, and tasks in EC. Then the transmission cost and computation cost of task offloading are defined in II-B. Finally, the task offloading problem is formulated in II-C to minimize these costs.

A. System Model

We consider a resource-limited EC environment. Different edge nodes offer different computation capacity and charge different computation costs. Mobile users generate jobs that can be offloaded to different nodes for real-time computation. A job contains one or more tasks with or without dependency. For dependent tasks, the following tasks need to wait for the completion of previous tasks. The main components are described as follows.

Node: A set of nodes $\mathbf{N} = \{N_1, N_2, ..., N_m\}$ are deployed in a specific area. Generally, the remote cloud can be considered as a node with near-infinite computation resources [7]. The features of each node N_i are described as follows:

- $N_i.l$ is the location.
- $N_i.c$ is the computation capacity, which is used to denote the CPU capacity [11].
- $N_i.p$ is the computation cost which indicates the cost of task execution per CPU resource per unit time.
- $N_i.b$ is the bandwidth.
- $N_i.c(t)$ is the available computation capacity at t.
- $N_i.a(t)$ is a set of running tasks on N_i at t.
- $N_i.v(t)$ is the SLAV, indicating the severity of resource violation at t.

Job: A job $J_i \in \mathbf{J}$ represents a collection of tasks generated by a mobile user to accomplish a specific goal. The features of job J_i are denoted as follows:

- $J_i.t$ is the arrival time when the job is sent to a node.
- $J_i.a$ is a set of tasks belonging to J_i, and the number of tasks belonging to J_i is denoted as $J_i.n$.
- $J_i.g$ is a DAG indicating the task dependencies of J_i.

Task: A task is executed for a minimal indivisible goal. For job J_i, the task set is denoted as $J_i.a = \{T_i^j | j \in [1, J_i.n]\}$. The features of task T_i^j is denoted as:

- $T_i^j.c$ is the requested CPU resource.
- $T_i^j.t$ is the arrival time.
- $T_i^j.e$ is the expected computation time.
- $T_i^j.z$ is the size of the data that needs to be transferred.
- $T_i^j.n$ is the assigned node.
- $T_i^j.l$ is the source location, which indicates the location of the user who generates T_i^j.
- $T_i^j.p$ and $T_i^j.s$ denote the set of predecessor tasks and successor tasks of T_i^j, respectively. If there is no dependency, then $T_i^j.p = T_i^j.s = \emptyset$.

Note that except for the main features introduced above, there are still some relevant features defined when they are used. With these definitions, the transmission cost and computation cost can be obtained for each task.

B. Cost Model

For each task, both the transmission cost and the computation cost are considered and unified by the cost charged. Task T_i^j with assigned node $T_i^j.n = N_k$ is analyzed as follows.

Transmission Cost: The transmission cost $C_{trans}^{i,j}$ of T_i^j includes the cost of transmitting T_i^j to the assigned node and the cost of transmitting data from all predecessor tasks to T_i^j. When a task T_i^j is offloaded, the data of this task need to be transmitted to the assigned node N_k to initialize the task. Let $d_{ijk} \propto |T_i^j.l - N_k.l|/N_k.b$ denote the transmission delay per unit size of data between T_i^j and N_k. Then the cost C_{init}^{ij} of transmitting T_i^j to N_k is denoted as:

$$C_{init}^{ij} = c \times T_i^j.z \times d_{ijk},$$

where c is the basic transmission cost per unit time.

Besides, let e_i^{jk} denote the data that need to be transferred from task T_i^j to task T_i^k. If the assigned nodes of T_i^j and T_i^k are the same, then $e_i^{jk} = 0$. Let d_{ij} be the delay in transmission of data per unit size between nodes N_i and N_j. Generally, $d_{ij} = d_{ji}$ and $d_{ij} = 0$ when $i = j$. Furthermore, if $T_i^j.n = N_p, T_i^k.n = N_q$, the transmission delay is $e_i^{jk} \times d_{pq}$, and the transmission cost is $c \times e_i^{jk} \times d_{pq}$. We use x_{jk} to indicate whether T_i^j and T_i^k are assigned to the same node, and if so, $x_{jk} = 1$, otherwise, $x_{jk} = 0$. For task T_i^j, the cost C_{data}^{ij} of its predecessor tasks to transfer data to it is defined as:

$$C_{data}^{ij} = \sum_{T_i^k \in T_i^j.p} c \times e_i^{jk} \times d_{pq} \times x_{jk}.$$

Then the transmission cost C_{trans}^{ij} of T_i^j is calculated as:

$$C_{trans}^{ij} = C_{init}^{ij} + C_{data}^{ij} \tag{1}$$

Computation Cost: For task T_i^j, its computation cost C_{comp}^{ij} is the sum of the expected computation cost and the additional computation cost due to SLAV of the assigned node [11]. The definition of SLAV is introduced as follows. Then the computation cost is defined based on SLAV.

SLAV: It is used to denote the utilization of the nodes. For node N_k, the resource request $N_k.r(t)$ at t is defined as:

$$N_k.r(t) = \sum_{T_i^j \in N_k.a(t)} T_i^j.c.$$

And the resource allocation $N_k.x(t)$ of N_k at t is defined as:

$$N_k.x(t) = \min\left(N_k.c, N_k.r(t)\right).$$

Then the SLAV $N_k.v(t)$ can be calculated as follows [16]:

$$N_k.v(t) = \frac{\sum_{T_i^j \in N_k.a(t)} T_i^j.c}{\min\left(N_k.c, \sum_{T_i^j \in N_k.a(t)} T_i^j.c\right)}.$$

With SLAV, the computation cost is then calculated as the sum of expected cost and additional cost. As task T_i^j is offloaded to node N_k, the computation time is $T_i^j.e$ and the computation amount is $N_k.p \times T_i^j.e$. Then the expected computation cost C_{exp}^{ij} of T_i^j is defined as follows:

$$C_{exp}^{ij} = N_k.p \times T_i^j.e \times T_i^j.c.$$

Moreover, the computation amount of the task is fixed, due to the possible SLAV of node N_k, the actual computation time of T_i^j may be longer than expected. For task T_i^j, the arrival time is $t_0 = T_i^j.t$, the expected finish time is $t_1 = T_i^j.t + T_i^j.e$. If SLAV occurs during time duration $[t_0, t_1]$, the actual allocated resource $T_i^j.x(t)$ can be calculated as:

$$T_i^j.x(t) = T_i^j.c \times \frac{1}{N_k.v(t)}.$$

Then the additional computation time can be calculated as:

$$\begin{aligned} \Delta t_{[t_0,t_1]}^{ij} &= \frac{(t_1 - t_0) \times T_i^j.c - \int_{t_0}^{t_1} T_i^j.x(t)\, dt}{T_i^j.c} \\ &= \int_{t_0}^{t_1} \left(1 - \frac{1}{N_k.v(t)}\right) dt. \end{aligned} \tag{2}$$

For time duration $[t_1, t_2]$, where $t_2 = t_1 + \Delta t_{[t_0,t_1]}^{ij}$, the additional time can be calculated with (2) if SLAV happens again. Thus, we can get $t_3 = t_2 + \Delta t_{[t_1,t_2]}^{ij}$, etc., until task T_i^j is finally finished. In short, the additional computation cost of task T_i^j can be denoted as:

$$\begin{aligned} C_{add}^{ij} &= N_k.p \times \left(\Delta t_{[t_0,t_1]}^{ij} + \Delta t_{[t_1,t_2]}^{ij} + \Delta t_{[t_2,t_3]}^{ij} + ...\right) \\ &= N_k.p \times \int_{t_0}^{t_{ij}} \left(1 - \frac{1}{N_k.v(t)}\right) dt, \end{aligned}$$

where $t_{ij} = t_1 + \Delta t_{[t_0,t_1]}^{ij} + \Delta t_{[t_1,t_2]}^{ij} + ...$ is the actual finish time of task T_i^j on node N_k. As t_{ij} is mainly influenced by SLAV, and the computation amount of task T_i^j is fixed, t_{ij} is finite and can be calculated finally. Then the computation cost of task T_i^j can be calculated as follows:

$$C_{comp}^{ij} = C_{exp}^{ij} + C_{add}^{ij} \tag{3}$$

C. Problem Formulation

In this subsection, first, some constraints of the task offloading process are introduced, then the task offloading problem is formulated.

Constraints: It is assumed that the scheduler is located at the central node [10], [16]. A job is initiated at a particular node and must end at the same node. To meet this requirement, two dummy tasks are inserted for a job, i.e., tasks T_i^0 and $T_i^{J_i.n+1}$ having zero execution time and zero communication cost. T_i^0 is inserted at the beginning of the task sequence to trigger the application to start execution, and $T_i^{J_i.n+1}$ is inserted to the end to retrieve all the results. The two dummy tasks inserted are required to be scheduled on a particular node:

$$T_i^0.n = T_i^{J_i.n+1}.n. \tag{4}$$

For task T_i^j, if $T_i^j.p \neq \emptyset$, which means there exists at least one task that must be finished before T_i^j can start. Then the

actual start time t_0^{ij} of task T_i^j must be after the end time of all its predecessor tasks, which can be denoted as:

$$t_0^{ij} \geq \max T_i^l.t_{il}, \quad T_i^l \in T_i^j.p, \tag{5}$$

where $T_i^l.t_{il}$ is the actual finish time of T_i^l. Besides, for each node, the resource constraint is defined as:

$$0 \leq \frac{N_k.a(t)}{N_k.c} \leq 1, \quad k = 1, 2, ..., m. \tag{6}$$

Problem Formulation: We aim to minimize the total cost of all task offloading decisions with these constraints. In short, the total cost C_{sum}^{ij} for T_i^j is the sum of transmission cost C_{trans}^{ij} and computation cost C_{comp}^{ij}. And the overall cost C of all tasks is the sum of C_{sum}^{ij}. The target is to find the best strategy which can minimize C while obeying the constraints in (4), (5), and (6). Therefore, the dynamic task offloading problem in EC is defined as follows:

Problem 1:

$$\min C = \sum_{T_i^j \in T} \left(C_{trans}^{ij} + C_{comp}^{ij} \right), \tag{7}$$
$$s.t. \quad (4) - (6) \quad .$$

Problem 1 is an advanced bin-packing problem, which is NP-hard and can only be solved heuristically. However, most of the existing heuristic algorithms are unstable in real EC environment and unable to achieve fast decision-making when facing large-scale problems. In this problem, the first-order transition probability of the tasks' resource demand is quasi-static for an extended period and not uniform distribution by adequately choosing the time slot duration [17]. Moreover, the arrival of tasks has the memoryless property [11]. Therefore, this problem can be modeled as an MDP and then solved by DRL based algorithms [15], as presented in the next section.

III. Dependent Task Offloading Algorithms

In this section, the dependent task offloading algorithms based on DRL and graph neural networks (GNN) are introduced. Before going into the details, reinforcement learning (RL) algorithms are briefly introduced as follows. In RL algorithms, at each time t, the RL agent collects system state S_t, and calculates the reward during last time slice R_{t-1}. Then, the agent selects action A_t according to a pre-defined strategy. After performing the action, the system would transit to the new state S_{t+1} in the next time slice. Similarly, the RL agent will repeat the above operations, i.e., calculating reward R_t and selecting new action A_{t+1} according to S_{t+1}.

Among different RL algorithms, the Q-learning algorithm has an advantage in fast decision-making [18]. Besides, for dependent tasks, the dependencies described as DAG need to be properly considered. Considering that GNN can easily handle graphical inputs, its representative GCN model is chosen to extract the information between vertices in the DAG.

The main components of our algorithms are introduced according to the components of RL algorithms. First, the GNN-based state information representation is introduced in III-A. Then the offloading action selection and the offloading algorithm are introduced in III-B and III-C, respectively.

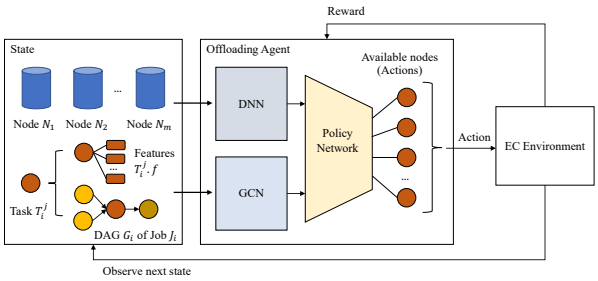


Fig. 1. Dependent Task Offloading Algorithm

A. State Embeddings

The system state is based on the tasks' transmission cost and computation cost, which are only related to the features and status of nodes, tasks, and jobs, according to (1) and (3). To extract the necessary information for offloading decisions, we do the node embedding, job-task embedding, respectively.

Node Embedding: For node N_k, the available CPU resource $N_k.c(t)$, SLAV $N_k.v(t)$, unit computation cost $N_k.p$, and bandwidth $N_k.b$ are all necessary information for decision-making. Besides, when a new task T_i^j arrives, in addition to get the current resource usage on these nodes, we also want to know the estimated resource usage in a limited future time to avoid the situation where some short tasks are offloaded to a node with high utilization, in which case the tasks have to wait for a relatively long time or the node will experience a severe SLAV. To extract this information, the expected remaining computation amount $N_k.r(t)$ is calculated as:

$$N_k.r(t) = \sum_{T_i^j \in N_k.a(t)} \left(t_0^{ij} + T_i^j.e - t \right) \times T_i^j.c.$$

Then the state of N_k can be denoted as $S_{N_k} = \{N_k.c(t), N_k.v(t), N_k.p, N_k.b, N_k.r(t)\}$. And the state of all nodes S_N can be denoted as $S_N = [S_{N_1}, S_{N_2}, ..., S_{N_m}]$.

To reduce the dimension of S_N and make it possible for further processing, S_N is encoded as a vector V_{S_N} to represent the status of all nodes after several nonlinear transformations, and the details are introduced in III-C.

Job-Task Embedding: For task T_i^j, the features can be denoted as $T_i^j.f = \{T_i^j.c, T_i^j.e, T_i^j.z\}$. Due to the dependencies among tasks belonging to the same job J_i, we use a GNN, which embeds the corresponding job information (DAG structure) into a set of vectors to better represent the necessary information for decision-making of task T_i^j. The network is designed based on GCN [13], and the details are as follows.

A DAG $G_i = (V_i, E_i)$ is used to denote all the tasks and dependencies of job J_i, where $V_i = J_i.a$, and E_i represents the dependencies. For each task of J_i, there is a feature description $T_i^j.f$. The features of job J_i can be summarized as:

$$X_i = [T_i^1.f; T_i^2.f; ...T_i^{J_i.n}.f], \quad X_i \in \mathbb{R}^{J_i.n \times D},$$

where $D = 3$ is the number of task features. An adjacency matrix M_i can be used to represent the graph structure of G_i. The state of T_i^j then can be denoted as $S_{T_i^j} = \{T_i^j.f, M_i\}$.

We aim to produce a task-level output $Z_i \in \mathbb{R}^{J_i.n \times F}$, where F is the number of output features per task, taking the X_i and M_i as the input. Each row Z_i^k of Z_i represents the embedding of task T_i^k of job J_i. With Z_i, we can get the overall information of all tasks of J_i. Each network layer of our GCN can be denoted as:

$$H_i^{(l+1)} = f\left(H_i^{(l)}, M_i\right) = \sigma\left(M_i H_i^{(l)} W^{(l)}\right), \tag{8}$$

where $H_i^{(0)} = X_i$, $Z_i = H_i^{(L)}$, and L is the number of layers. $W^{(l)}$ is a weight matrix for the l-th network layer and $\sigma(\cdot)$ is a nonlinear activation function, e.g., ReLU [19]. However, (8) means that for every task, all the features of neighboring nodes are summed up except itself. Besides, M_i is not normalized and (8) will change the scale of the feature vectors. To solve these problems, a new propagation rule is adopted [13]:

$$H_i^{(l+1)} = \sigma\left(\hat{D_i}^{-\frac{1}{2}}\left(M_i + I\right)\hat{D_i}^{-\frac{1}{2}} H_i^{(l)} W^{(l)}\right), \tag{9}$$

where I is the identity matrix. $\hat{D}_i$ is the diagonal node degree of $\hat{M}_i = M_i + I$. With (9), the output $Z_i = H_i^{(L)}$ of job J_i can be obtained. Then the embedding of T_i^j is calculated as:

$$V_{S_{T_i^j}} = g\left(\sum_{T_i^k \in T_i^j.s} Z_i^k\right),$$

where $Z_i^k \in \mathbb{R}^{1 \times F}$ is the embedding feature of T_i^k extracted from Z_i, and $g(\cdot)$ is a nonlinear activation function.

The process of embedding job information with GCN is demonstrated in Fig. 2. As shown in Fig. 2, given the task dependencies M_i of the job J_i and the features X_i of these tasks, the information of each task propagates along the dependency path. To get the embedding vector $V_{S_{T_i^1}}$ of task T_i^1, the information of task T_i^6 is first transferred to task T_i^3, T_i^4 and T_i^5, and the information of task T_i^5 is transferred to task T_i^2 and T_i^4. Then, task T_i^2, T_i^3 and T_i^4, which aggregate the information of previous tasks, propagate the information to task T_i^1. After integrating its own information, we finally get the embedding vector $V_{S_{T_i^1}}$ of task T_i^1. The embedding vectors of the other tasks are also obtained in a similar way. In short, the state of EC at t can be denoted as:

$$S_t = [S_N, S_{T_i^j}] \in \mathbb{S}, \tag{10}$$

where $\mathbb{S}$ is the set of all states. The state embeddings can be denoted as $V_{S_t} = [V_{S_N}, V_{S_{T_i^j}}]$. After state embeddings, the offloading action selections can be made accordingly.

B. Offloading Action Selection

In this subsection, the offloading action set and action selection algorithm are introduced. The action set is the set of all available nodes. Besides, the action is selected based on ϵ-greedy algorithm with the estimated cost.

Action Set: For any arriving task, the decision goal is to find a suitable node to offload the task, therefore the range of actions is the set of all the nodes, which is defined as:

$$A_t \in \{N_1, N_2, ..., N_m\} = \mathbb{A},$$

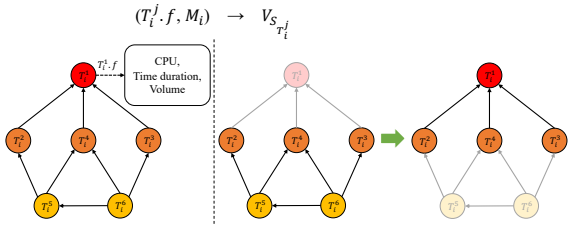


Fig. 2. Job Embedding

where A_t is the action at time t, and $\mathbb{A}$ is the set of all actions.

Action Selection: The action is selected based on ϵ-greedy algorithm, which consists of exploration and exploitation [20]. To select a best action in exploitation, a deep Q-learning based algorithm [15], [18] is adopted which has an advantage in fast computation for fast decision-making in EC. In such algorithm, the quality of each state-action pair is indicated by Q-value $Q(S_t, A_t)$. The best action is the action with maximum Q-value, which is selected as:

$$A_t = arg\max_{A_t} Q(S_t, A_t). \tag{11}$$

However, in exploration, randomly selected actions could be terrible decisions. For example, offloading a task to an over-utilized node can lead to more severe SLAV and extended computation time for running tasks. Besides, offloading a dependent task to a node far from the node of its predecessor task can lead to high transmission costs. To avoid these problems, some constraints are added as follows.

Firstly, to select a better random action, the estimated cost E_{ij}^k is calculated for placing the task T_i^j to each node N_k, which is defined as:

$$E_{ij}^k = C_{trans}^{ij} + C_{exp}^{ij} + \Delta E_{ij}^k,$$

where ΔE_{ij}^k is the estimated additional cost due to SLAV change of node N_k, which is defined as:

$$\Delta E_{ij}^k = \left(\frac{1}{N_k.v(t)} - \frac{1}{E_{slav}^k}\right) \times T_i^j.e \times \sum_{T_l^m \in N_k.a(t)} T_l^m.c,$$

where E_{slav}^k is the estimated SLAV if task T_i^j is offloaded to node N_k, which can be calculated as:

$$E_{slav}^k = \frac{N_k.r(t) + T_i^j.c}{\min(N_k.a(t) + T_i^j.c, N_k.c)}.$$

We denote $E_{ij} = \{E_{ij}^k | k \in [1, n]\}$ the estimated cost of all nodes. Then the estimated best action can be denoted as:

$$A_E = arg\min_k E_{ij}^k. \tag{12}$$

Secondly, to further avoid high transmission costs, a threshold is added for each job. Both the distributions of the job durations and the number of tasks from Alibaba cluster trace [21] follow the long-tail distribution. In this case, we can distinguish the relative short jobs and reduce the transmission cost between dependent tasks of these jobs by assigning these tasks to the same node.

Algorithm 1 Action Selection

Input: $S(t), T_i^j$
Output: A_t
1: **if** $T_i^j.n = \emptyset$ **then**
2: **if** $\phi > \epsilon_1$ **then**
3: Select A_t by (11)
4: **else if** $\phi < \epsilon_2$ **then**
5: Select $A_t = A_E$ by (12)
6: **else**
7: $A_t = A_{rand}$
8: **end if**
9: **if** $J_i.d < th_1$ and $J_i.n < th_2$ **then**
10: Handle $J_i.a \setminus \{T_i^j\}$
11: **end if**
12: **else**
13: Handle T_i^j
14: **end if**
15: **end**

Finally, the action selection algorithm is shown in Algorithm 1. ϵ_1, ϵ_2 ($\epsilon_1 \geq \epsilon_2$), th_1, and th_2 are set in advance. If $T_i^j.n = \emptyset$, the action is selected according to the generated random number ϕ as shown in lines 2 - 8. If $\phi > \epsilon_1$, then the best action is selected according to (11). If $\phi < \epsilon_2$, then the best estimated action is selected according to (12). Otherwise, a random action A_{rand} is selected. After that, we calculate the job duration $J_i.d = \sum_{T_i^j \in J_i} T_i^j.e$, and check if the job duration $J_i.d$ and task number $J_i.n$ satisfy the thresholds th_1 and th_2, respectively. If so, the tasks of J_i except T_i^j is handled, i.e., assign the tasks to the offloaded node of T_i^j and push the tasks in a priority queue waiting for process. The details of the priority queue will be explained in the next subsection. At last, as shown in line 13, if $T_i^j.n \neq \emptyset$, it means that the task satisfy the threshold policy and has been pre-assigned to a node, this task is assigned to the specific node and the environment is updated accordingly.

C. Offloading Algorithm

In this subsection, the reward of our algorithm is introduced, which is divided into instant reward and delayed reward. Then, the training process is designed based on the DQN [15] with modified reward, action selection, and replay memory.

Reward: As shown in Problem 1, we aim to minimize the transmission cost and computation cost of tasks. For each task T_i^j, the transmission cost C_{trans}^{ij} is an instant cost. However, the computation cost C_{comp}^{ij} contains two parts: instant cost C_{exp}^{ij} and delayed cost C_{add}^{ij}. Based on this, the reward R_t is separated into instant reward R_t^i and delayed reward R_t^d.

Firstly, the instant reward is calculated upon the offloading decision is made, which is defined as:

$$R_t^i = C_{trans}^{ij} + C_{exp}^{ij} \tag{13}$$

Secondly, the delayed reward is caused by SLAV and can not be calculated instantly. The additional cost C_{add}^{ij} denotes the delayed reward of T_i^j. However, the SLAV affects not only the task T_i^j, but also the other tasks running on node $T_i^j.n$. The total delayed reward R_t^d can be calculated as:

$$R_t^d = N_k.p \times \int_{T_i^j.t}^{t_{ij}} \left(1 - \frac{1}{N_k.v(t)}\right) dt + \int_{T_i^j.t}^{T_i^j.t+T_i^j.e} \left(\frac{1}{N_k.v(t)} - \frac{1}{E_{slav}^k}\right) \times \left(\sum_{T_i^m \in J_i.a \setminus \{T_i^j\}} T_i^m.c\right) dt,$$

where t_{ij} is the actual finish time of task T_i^j as defined in II-B. The delayed reward R_t^d is obtained when the task finishes.

Finally, the total reward for A_t of T_i^j can be denoted as:

$$R_t = R_t^i + R_t^d. \tag{14}$$

Training: The dependent task offloading (DTO) algorithm is proposed based on DQN as shown in Algorithm 2. The

Algorithm 2 Dependent Task Offloading

Input: $E, \mathbf{J}, \theta, \theta' = \theta$
Output: A_t
1: Init T_{PQ} by (16)
2: **while** $T_{PQ} \neq \emptyset$ **do**
3: Get T_i^j from T_{PQ}
4: **if** $T_i^j.s = start$ **then**
5: Get state S_t by (10)
6: Select action A_t by Algorithm 1
7: Get R_t^i according to (13)
8: Update E, store task memory $T_i^j.m = (S_t, A_t, R_t^i)$
9: Push $(T_i^j.t, end, T_i^j)$ to T_{PQ}
10: **for** (S_τ, A_τ, R_τ) in D_{tmp} **do**
11: Push $(S_\tau, A_\tau, R_\tau, S_t)$ to D
12: **end for**
13: Set $D_{tmp} = \emptyset$, and sample $D_\tau \subset D$
14: **for** $\left(S_{\tau-1}^{(j)}, A_{\tau-1}^{(j)}, R_{\tau-1}^{(j)}, S_\tau^{(j)}\right)$ in D_τ **do**
15: Calculate $Q(S_\tau^{(j)}, A_\tau^{(j)}; \theta')$
16: Calculate $y(\tau)$ by (15)
17: **end for**
18: Train policy network $\theta = arg\min_\theta L(\theta)$
19: From time to time set $\theta' = \theta$
20: **else**
21: Update $T_i^j.\Delta t$
22: **if** $T_i^j.\Delta t = 0$ **then**
23: Add $T_i^m \in T_i^j.s$ to T_{PQ}
24: Get (S_τ, A_τ, R_τ) from $T_i^j.m$
25: Calculate reward R_t by (14)
26: Push (S_τ, A_τ, R_t) to D_{tmp}
27: **else**
28: Push $(T_i^j.ts + T_i^j.\Delta t, end, T_i^j)$ to T_{PQ}
29: Set $T_i^j.\Delta t = 0$
30: **end if**
31: **end if**
32: **end while**
33: **end**

objective of the training process is to minimize the loss $L(\theta)$, which is defined as:

$$L(\theta) = E[(y(t) - Q(S_t, A_t; \theta))^2],$$

where $Q(S_t, A_t; \theta)$ is the output of the policy network with weights θ. The policy network is used to store and calculate Q-values. $y(t)$ is the target Q-value and defined as:

$$\begin{aligned} y(t) = E[(1-\alpha)Q(S_{t-1}, A_{t-1}; \theta') + \alpha\,(R_{t-1} \\ +\gamma \max Q(S_t, A_t; \theta'))\,|S_{t-1}, A_{t-1}], \end{aligned} \tag{15}$$

where α is the learning rate and γ is the discount parameter. θ' is the weights of the target network. These two networks share the same network structure, but the weights of the target network θ' are copied from the policy network θ at regular intervals, in order to provide more stable training results.

In Algorithm 2, the input is the environment E, the job set $\mathbf{J}$, and the policy network with weights θ. The output is the offloading decision A_t at each time t. To handle the tasks and obtain the delayed reward, a priority queue T_{PQ} is used to store all pending and running tasks, which is defined as:

$$T_{PQ} = \{(T_i^j.ts, T_i^j.s, T_i^j) | T_i^j \in T \text{ and } T_i^j.p = \emptyset\},$$

where $T_i^j.ts$ is the timestamp of T_i^j, and $T_i^j.s \in \{start, end\}$ is the status of T_i^j. If $T_i^j.s = end$, then $T_i^j.ts$ is the expected finish time and the task T_i^j will be finished at $T_i^j.ts$. Otherwise, $T_i^j.ts$ is the start time of T_i^j and $T_i^j.ts = T_i^j.t$. At the beginning of DTO, the T_{PQ} is initialized as follows:

$$T_{PQ} = \{(T_i^0.t, start, T_i^0) | i \in [1, J_i.n]\}, \tag{16}$$

where T_i^0 is the inserted task 0 for job J_i as described in II-C. After that, while T_{PQ} is not empty, the task T_i^j is obtained with minimal timestamp from T_{PQ}. If $T_i^j.s = start$, as shown in lines 4 - 19, the DRL agent first gets the state S_t, selects an action A_t, and gets the instant reward R_t^i. The environment E is then updated. The instant reward R_t^i along with current state S_t and action A_t as a tuple (S_t, A_t, R_t^i) is stored in the task memory $T_i^j.m$. Then this task is pushed to T_{PQ}.

Moreover, as in lines 10 - 12, D_{tmp} is a temporary memory storing all (S_τ, A_τ, R_τ) from last task arrival time, i.e., the state is only observed when a task arrives. The reason is that other tasks may finish before a task starts and affect the state information. For each (S_τ, A_τ, R_τ) in D_{tmp}, the current state S_t is added to it as the next state of S_τ, and then $(S_\tau, A_\tau, R_\tau, S_t)$ is pushed to experience replay memory D [15]. After that, D_{tmp} is cleared and a subset D_τ is sampled from D. As shown in lines 13 - 19, for each entry in D_τ, the corresponding $Q(S_\tau^{(j)}, A_\tau^{(j)}; \theta')$ and $y_d(\tau)$ are calculated. Finally, a gradient descent step is performed in the training process, and the weights θ of the policy network is copied to the target network from time to time.

Otherwise, if the task status is end, as in lines 20 - 31, the additional running time $T_i^j.\Delta t = \Delta t^{ij}_{[t', T_i^j.ts]}$ is updated due to SLAV according to (2), where t' is the timestamp of last time which can be easily recorded in experiments. If $T_i^j.\Delta t = 0$, there is no additional running time. Then all successor tasks of T_i^j to T_{PQ} are added, and the task memory $T_i^j.m$ is read. The final reward is calculated based on (14) and the entry (S_τ, A_τ, R_t) is stored in D_{tmp} for further getting the next state. Else, if $T_i^j.\Delta t \neq 0$, the expected finish time is updated and T_i^j is pushed back to T_{PQ} for further processing.

IV. Experiment

In this section, first the data preprocessing and parameter settings are introduced in IV-A and IV-B, respectively. Then the experimental results are shown in IV-C.

A. Data Preprocessing

In our experiment, the real-world cluster trace *cluster-trace-v2018* from Alibaba [21] is used, which includes about 4000 servers in a period of 8 days. Although this is a data set of cloud computing, it is also widely adopted to represent EC scenarios after proper preprocessing [22].

The first 1×10^6 records are intercepted, and the records which contain missing values are removed. Then, the validity of the start and end time of tasks is checked, and the duration of each task is calculated. The consistency of the records is reviewed, and the records with incomplete task dependencies are deleted. After that, the data set contains about 8.9×10^5 tasks and 2.6×10^5 jobs.

From the preprocessed data set, the task dependencies of a job are extracted as $J_i.g$. The task duration is taken as the expected computation time $T_i^j.e$. Moreover, the requested CPU resource is taken as $T_i.c$. Other necessary features are randomly generated, as described in the following subsection.

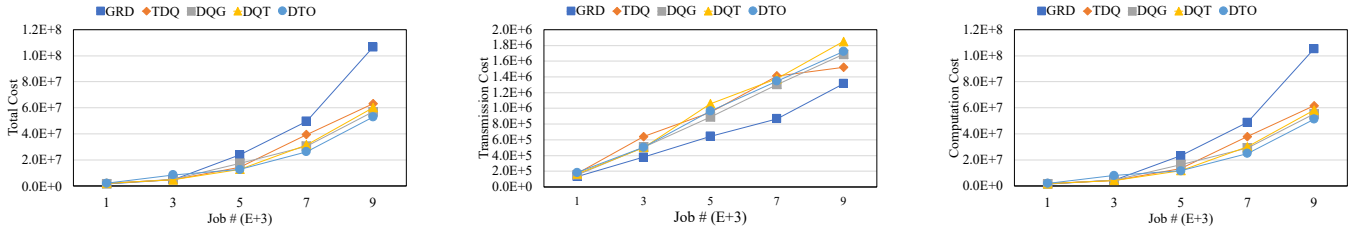
B. Parameter Settings

Node: For each node N_k, its location $N_k.l$ is represented by two-dimensional coordinates between $[0, 100]$. The computation capacities of each node $N_k.c$ and bandwidth $N.b$ are set differently in the range of $[50, 100]$ and $[12, 250]$, respectively. The default computation cost $N.p$ is set to 1 and it is changed during the experiments.

Job and Task: The arrival time of each job $J.t$ is set to a random number between $[0, 1000]$. The volume $T_i^j.z$ of task T_i^j is a random integer between $[1, 100]$. Besides, the source location $T_i^j.l$ is initialized in the same way as that of $N_k.l$.

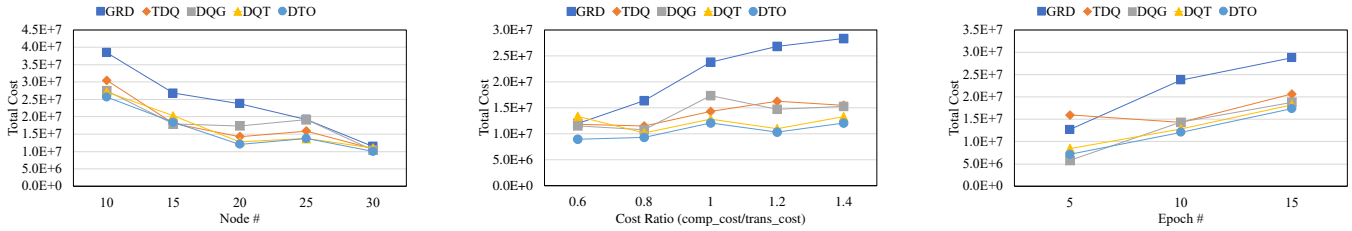
Algorithm: In Algorithm 1, the value of ϵ_1 and ϵ_2 are set to $\epsilon_1 = \epsilon_2 = 0.9$. Besides, the thresholds th_1 and th_2 for threshold policy are set to $th_1 = 10$ and $th_2 = 2$, respectively. In Algorithm 2, both the learning rate α and the discount parameter γ are set to 1. Besides, the training interval of the policy network and the update interval of the target network is set to 10 and 100, respectively. Two fully connected layers are used for node embedding, one GCN layer is used for job-task embedding, and then two fully connected layers are used to combine all features.

Furthermore, the number of jobs, nodes, training epochs, and the value of cost ratio w are changed. The cost ratio w is used to control the proportion of transmission cost c and computation cost $N_k.p$ in the total cost, i.e., it equals to change the total cost to $\sum_{T_i^j \in T} \left(C^{ij}_{trans} + w \times C^{ij}_{comp}\right)$ to investigate the effect of difference cost ratio.



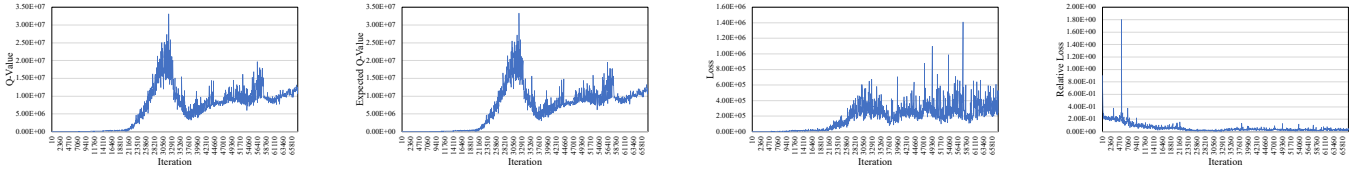
(a) Total Cost (b) Transmission Cost (c) Computation Cost

Fig. 3. Performance with Different Job Number



(a) Different Node Number (b) Different Cost Ratio (c) Different Epoch Number

Fig. 4. Performance with Different Node Number, Cost Ratio, and Epoch Number



(a) Q-Value (b) Expected Q-Value (c) Loss (d) Relative Loss

Fig. 5. Training Process of DTO

C. Experimental Results

To show the effectiveness of our DTO algorithm, the following baselines are adopted: Greedy (GRD) [7], Traditional Deep Q-learning (TDQ) [15], Deep Q-learning with GCN only (DQG), and Deep Q-learning with Threshold policy only (DQT). The experimental results are described as follows.

Job Number: The performance with different job numbers is shown in Fig. 3. When the job number increases, the advantage of the total cost in DTO becomes more obvious, as shown in Fig. 3(a). This is because the shortage of resources is greater when the job number grows, while the GRD algorithm does not consider SLAV. From Fig. 3(a) it can be concluded that in most cases, the DTO has less total cost than baselines. Compared with the DQT algorithm, the performance of the DTO algorithm is better since it can take the dependencies between tasks into consideration. Besides, DTO is better than DQG since it improves the action selection to avoid poor random offloading decisions.

Fig. 3(b) and Fig. 3(c) demonstrates the transmission cost and computation cost, respectively. As illustrated in (1) and (3), when the job number increases, the request CPU and SLAV also increase, then the transmission cost and computation cost increases accordingly. As shown in Fig. 3(b), the performance of transmission cost is almost the same for all algorithms. The reason is that both DRL-based algorithms and GRD algorithms take it into account. However, the GRD algorithm cannot take additional computation cost, caused by SLAV into consideration. Thus as shown in Fig. 3(c), the computation cost of GRD is greater than other algorithms. Furthermore, with the GCN and threshold policy, the performance is further improved.

Node Number: Fig. 4 demonstrates the performance of total cost with different node numbers, cost ratios, and epoch numbers. As shown in Fig. 4(a), with the increase in the number of nodes, the total amount of computation resources in the system also increases. Fig. 4(a) shows our DTO algorithm is more outstanding in the case of resource shortages when the node number is small, i.e., our DTO algorithm can offload tasks effectively in a resource-limited EC scenario.

Cost: In Fig. 4(b), the performance of different cost ratios is shown. From Fig. 4(b) it can be seen that in most cases, the total cost of DTO is smaller than other algorithms.

Epoch Number: Epoch refers to the number of training rounds. The epoch number is changed to show the effective-

ness of our algorithm, and the results are shown in Fig. 4(c). It can be concluded that with enough training rounds, our DTO algorithm performs better than other algorithms.

In short, DTO has an overall best performance than all baselines and outperforms the baseline approaches 60.55% maximum and 13.78% on average regarding total offloading cost. In addition to the performance with different parameter settings, we also investigate the performance of our algorithm in learning speed and convergence as follows.

Learning Speed: The Q-value, expected Q-value, loss, and relative loss during the training process are shown in Fig. 5. These values are sampled every 50 iterations. As shown in Fig. 5(a) and 5(b), the Q-value predicted by neural network and the calculated expected Q-value is very similar. At the beginning of training, the Q-value is relatively small, because the system can handle all the tasks that arrive at this time. Subsequently, as the number of arriving tasks increases, the Q-value also increases. The reason is that the Q-value is updated according to the reward, and the reward is affected by the number of tasks. After a period of time, as the system stabilizes, the Q-value also decreases and stabilizes.

Furthermore, as shown in Fig. 5(c) and 5(d), when the Q-value and expected Q-value grows, the loss also grows. In Fig. 5(d), the relative loss is defined as the quotient of the loss divided by the expected Q-value. It is clear to see that the relative loss tends to be stable after about 25000 iterations, which is indicative of the convergence of the algorithm. Besides, the results show that the learning speed of our DTO algorithm is fast enough to deal with task scheduling in EC.

V. Conclusion

In this paper, we first model the resource-limited EC, which includes the edge nodes, jobs, and tasks. The dependent tasks are offloaded to different nodes to be processed. Then the task offloading decision process is modeled as an MDP problem, and a DRL-based algorithm is proposed. The GCN and threshold policy are used to improve the state information representations and action selections, respectively. Finally, the experiments are conducted with real-world data-trace, and the results show that the DTO algorithm outperforms the baseline algorithms 13.78% on average in regarding total offloading cost. Future work will consider the joint task offloading and service placement problem to maximize the profit of nodes.

Acknowledgment

This work is supported by Chinese National Research Fund (NSFC) Key Project No. 61532013 and No. 61872239; also partially funded by The Science and Technology Development Fund, Macau SAR, FDCT.0007/2018/A1, FDCT.0060/2019/A1, and by University of Macau (File no. MYRG2018-00237-FST, CPG2020-00015-IOTSC and SRG2018-00111-FST).

References

[1] 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, 2017.

[2] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[3] K. Boos, D. Chu, and E. Cuervo, "Flashback: Immersive virtual reality on mobile devices via rendering memoization," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2016, pp. 291–304.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[5] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE, 2015, pp. 73–78.

[6] F. Zhang, Z. Tang, M. Chen, X. Zhou, and W. Jia, "A dynamic resource overbooking mechanism in fog computing," in *2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2018, pp. 89–97.

[7] S. Sundar and B. Liang, "Offloading dependent tasks with communication delay and deadline constraint," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 37–45.

[8] L. Liu, R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser, "Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 68–80.

[9] J. Zhu, X. Li, R. Ruiz, and X. Xu, "Scheduling stochastic multi-stage jobs to elastic hybrid cloud resources," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1401–1415, 2018.

[10] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," *arXiv preprint arXiv:1810.01963*, 2018.

[11] X. Zhou, K. Wang, W. Jia, and M. Guo, "Reinforcement learning-based adaptive resource management of differentiated services in geo-distributed data centers," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 2017, pp. 1–6.

[12] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 207–215.

[13] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[14] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 372–382.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[16] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, 2018.

[17] Z. Han, H. Tan, G. Chen, R. Wang, Y. Chen, and F. C. Lau, "Dynamic virtual machine management via approximate markov decision process," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[18] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[20] M. Tokic and G. Palm, "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," in *Annual Conference on Artificial Intelligence*. Springer, 2011, pp. 335–346.

[21] Alibaba. (2019, Mar.) Alibaba cluster trace program. [Online]. Available: https://github.com/alibaba/clusterdata

[22] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *Proceedings of the International Symposium on Quality of Service*. ACM, 2019, p. 20.