Deep Reinforcement Learning-based Task Scheduling Scheme in Mobile Edge Computing Network

Qi Zhao[a], Mingjie Feng[a], Li Li[a], Yi Li[a], Hang Liu[b] and Genshe Chen[a]
[a]Intelligent Fusion Technology, Inc., Germantown, MD 20876;
[b]The Catholic University of America, Washington, DC 20064;

ABSTRACT

Mobile edge computing is a new distributed computing paradigm which brings computation and data storage closer to the location where it is needed, to improve response time and save bandwidth in dynamic mobile network environments. Despite the improvements in network technologies, data centers cannot always guarantee acceptable transfer rates and response time, which could be critical requirements for many applications. The aim of mobile edge computing is to move the computation from data centers towards the edge of the network, exploiting smart objects, mobile phones, base stations and network gateways to perform tasks and provide services on behalf of the cloud. In this paper, we design a task offloading scheme in mobile edge networks to handle the task distribution, offloading and management by applying deep reinforcement learning. Specifically, we formulate the task offloading problem as a multi-agent reinforcement learning problem. The decision-making of each agent is modeled as a Markov decision process and a deep reinforcement learning approach is applied to deal with the large scale of dynamic states and actions. To evaluate the performance of our proposed scheme, we develop a simulation platform for the mobile edge computing scenarios. Our preliminary evaluation results indicate that the proposed solution can provide lower latency for the computational intensive tasks in mobile edge networks, and outperform baseline task offloading methods.

Keywords: Mobile edge computing, Distributed computing, Task scheduling and offloading, Markov decision process , Deep reinforcement learning.

1. INTRODUCTION

Edge computing is a distributed computing paradigm which brings computation and data storage closer to the location where it is needed, to improve response times and save bandwidth [1]. Despite the improvements in network technology, data centers cannot always guarantee acceptable transfer rates and response times, which could be a critical requirement for many applications. The aim of Edge Computing is to move the computation away from data centers towards the edge of the network, exploiting smart objects, mobile phones or network gateways to perform tasks and provide services on behalf of the cloud. By moving services to the edge, it is possible to provide content caching, service delivery, storage and device management resulting in better response times and transfer rates.

Mobile Edge Computing (MEC) is a brand new network paradigm that provides information technology services within the mobile access network of Mobile Units (MUs). European Telecommunications Standards Institute (ETSI) pointed out that MEC will provide a new ecosystem to migrate intensive computing tasks of MUs [2][3]. MEC is located close to MUs and it is deployed within the Radio Access Network (RAN). Therefore, it can provide higher bandwidth with lower latency to improve the Quality of Service (QoS). In the current 5G development trend, MEC also plays an important role and helps to meet the 5G high standards for delay [4].

However, the computation power of the edge devices are also limited due to the fast growing demand of the network users and load. When the tasks cannot be completely handled by the edge server, the edge server will offload a portion of the simulation task to the cloud data centers [5][6]. In this case, reducing the traffic and processing delay in the backhaul network becomes significant. Conventional task offloading algorithms [7][8][9][10][11], which select the cloud server according to some static rules (e.g., minimum distance), have two main shortages that can potentially cause huge traffic delays. First, the static offloading policy is not able to adapt to the time-varying network conditions. When the distribution of the traffic and computation load varies in the network. An offloading decision made without utilizing the network condition may keep forwarding heavy traffic to a node that encounters congestion, which further aggravates the congestion. It is critical for the offloading algorithm to sense the network conditions and actively avoid those congested or busy cloud servers to reduce the cost of offloading. Secondly, conventional offloading methods only offload the tasks to a single cloud

Sensors and Systems for Space Applications XIV, edited by Genshe Chen, Khanh D. Pham, Proc. of SPIE Vol. 11755, 117550K · © 2021 SPIE
CCC code: 0277-786X/21/$21 · doi: 10.1117/12.2589070

server. In this case, it could be extremely difficult to balance the load at cloud servers if some edge servers are trying to offload heavy tasks.

In this paper, we address the task offloading problem of unable to handle the computational intensive tasks in edge by proposing a deep reinforcement learning based method to help making the decision of the target of incoming tasks. Additionally, we validated and evaluated a simplified model to solve the task offloading problem to prove the feasibility of applying deep reinforcement learning to MEC task offloading scenarios. The proposed solution will meet the QoS requirements (i.e., latency) with minimal effort. Therefore, our main contributions in this work are:

1) Propose a deep reinforcement learning based method for task offloading in MEC environment;
2) Validate our proposed solution via a simplified simulation with using multi-armed bandit model;

In Section 2, we introduce the system model for the deep reinforcement learning based task offloading method. Section 3 describes the simplified simulation environment and the experiment detail, along with the validation and evaluation results and analysis. Section 4 introduces the conclustion for our work and the future plan.

2. DEEP REINFORCEMENT LEARNING BASED TASK OFFLOADING

2.1 System Model

We consider a MEC system with M MEC servers indexed by $m \in \{1, \cdots, M\}$ and J cloud servers indexed by $j \in \{1, \cdots, J\}$, which collectively provide task offloading services to K MUs indexed by $k \in \{1, \cdots, K\}$ at a training site. At each time slot, MEC server m has N_m tasks to be executed indexed by $i \in \{1, \cdots, N_m\}$. Let $s_{m,i}$ be the size of input data (in bits) for task i at MEC server m (e.g., the size of a video clip taken from the training site) and let $z_{m,i}$ be its complexity, which is defined by the number of CPU cycles required to execute one bit of the task. Then, the number of CPU cycles required to complete tasks i is $s_{m,i}z_{m,i}$, and the number of CPU cycles required to complete all tasks at MEC server m is $\sum_{i=1}^{N_m} s_{m,i}z_{m,i}$.

To mitigate the computational load at the edge servers and guarantee timely processing of the simulation tasks, a proportion of the simulation tasks will be offloaded from an MEC server to one of the cloud servers. The associations between MEC and cloud servers are specified by the following binary variables.

$$x_{m,j} \in \begin{cases} 1, & \text{if MEC server } m \text{ offloads tasks to cloud server } j \\ 0, & \text{otherwise} \end{cases}, \; m \in 1,...,M; \; j \in 1,...,J. \tag{1}$$

Since we consider the scenario where each MEC server can only select at most one cloud server for task offloading, we have $\sum_{j=1}^{J} x_{m,j} \leq 1$. Suppose MEC server m is associated with cloud server j (i.e., $x_{m,j} = 1$), the task assignment between MEC and cloud servers can be denoted by the following binary variables.

$$a_{m,i} \in \begin{cases} 1, & \text{The } i \text{ th task is offloaded to the selected cloud server} \\ 0, & \text{The } i \text{ the task is executed by MEC server } m \end{cases} \quad m \in 1,...,M; \; i \in 1,...,N_m. \tag{2}$$

Then, the time required for locally executing the tasks at MEC server m is given by

$$t_{\text{comp},m}^{(\text{E})} = \frac{\sum_{i=1}^{N_m}(1 - a_{m,i})s_{m,i}z_{m,i}}{c_m^{(\text{E})}} \tag{3}$$

where $c_m^{(\text{E})}$ is the computational capability of MEC server m, which is measured in CPU cycles per second. Let $c_j^{(\text{C})}$ be the computational capability of cloud server j that is available for the MEC servers (at current stage, we assume that $c_j^{(\text{C})}$ is a constant for simplicity; for future analysis, we may consider $c_j^{(\text{C})}$ fluctuates according to a Markov process). We assume that $c_j^{(\text{C})}$ is allocated is a way such that all tasks offloaded from various MEC servers associated with cloud server j are completed at the same time. The design rationale behind this allocation is that the computational capability is fully utilized. In other words, the following scenario will not happen: when the tasks that require less CPU cycles are competed earlier, the released computational capability cannot be used by other tasks that are still being executed. Under the optimal computational capability allocation that all tasks are completed at the same time, the computational capability allocated to

a task is proportional to the number of CPU cycles required to execute the task. Note that the total number of CPU cycles for all tasks at cloud server j is $\sum_{m=1}^{M} x_{m,j} \sum_{i=1}^{N_m} a_{m,j} s_{m,j} z_{m,j}$, the computational capability allocated to the i th task of MEC server m by cloud server j is given by

$$c_{m,i}^{(\mathrm{C})} = \frac{a_{m,i} s_{m,i} z_{m,i}}{\sum_{m=1}^{M} x_{m,j} \sum_{i=1}^{N_m} a_{m,i} s_{m,i} z_{m,i}} c_j^{(\mathrm{C})} \tag{4}$$

Then, the execution time of all tasks offloaded to cloud server j is given by

$$t_{\mathrm{comp},j}^{(\mathrm{C})} = \frac{a_{m,i} s_{m,i} z_{m,i}}{c_{m,j}^{(C)}} = \frac{\sum_{m=1}^{\mathrm{M}} x_{m,j} \sum_{i=1}^{N_m} a_{m,i} s_{m,i} z_{m,i}}{c_j^{(\mathrm{C})}} \tag{5}$$

We then present the communication latency for offloading the tasks from MEC servers to cloud servers. The communication latency consists of two parts, one is the access latency and the other is data transmission latency. The access latency is the round-trip time for a packet to travel between MEC and cloud servers, which is normally measured with "Ping" messages. The data transmission latency is the time spent on transmitting all the packets, which is determined by the size of transmitted data and the data rate of backhaul connection. Due to the time-varying network dynamics, both the access latency and the data rate of backhaul link fluctuate over time, and such fluctuation is expected to be memoryless. Note that, access latency consists of the uplink component (from MEC server to cloud server) and the downlink component (from cloud server to MEC server) of the access latency. In the MEC-based simulation system, the two components are separated by the computing time at the cloud server, and the network dynamics may be changed during this period. Thus, the actual round-trip time could be different from the one measured by "Ping" messages. To obtain accurate round-trip time, the "Ping" message can be modified to record the total elapsed time from the beginning of task offloading to the completion of downloading outcome from cloud server. Then, the access latency can be calculated by subtracting the data transmission time and computing time from the total elapsed time. Let $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$ and $t_{\mathrm{acc},m,j}^{(\mathrm{DL})}$ be the time-varying uplink and downlink access latency of the backhaul link between MEC server m and cloud server j, respectively. Based on the memoryless property, we consider $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$ and $t_{\mathrm{acc},m,j}^{(\mathrm{DL})}$ $(m \in \{1, \cdots, M\},\ j \in \{1, \cdots, J\})$ follow a Markov process with finite number of states, each state corresponds to a certain range of $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$. Take $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$ as an example, we divide the range of the possible value of $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$ into multiple intervals with equal length of l, given by $\left[t_{\min}^{(\mathrm{UL})}, t_{\min}^{(\mathrm{UL})} + l\right], \left[t_{\min}^{(\mathrm{UL})} + l, t_{\min}^{(\mathrm{UL})} + 2l\right], \ldots, \left[t_{\max}^{(\mathrm{UL})} - l, t_{\max}^{(\mathrm{UL})}\right]$, where $t_{\min}^{(\mathrm{UL})}$ and $t_{\max}^{(\mathrm{UL})}$ are the predefined minimum and maximum possible values of $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$, respectively, which can be set according to historical data of $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$ Given the interval length l, the total number of intervals is $(t_{\max}^{(\mathrm{UL})} - t_{\min}^{(\mathrm{UL})})/l$, which is also the number of states. We define that $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$ is in state s if $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$ falls into the s th interval, i.e., $t_{\mathrm{acc},m,j}^{(\mathrm{UL})} \in \left[t_{\min}^{(\mathrm{B})} + (s-1)l, t_{\min}^{(\mathrm{B})} + sl\right]$. The Markov process for $t_{\mathrm{acc},m,j}^{(\mathrm{DL})}$ is defined in the same way as $t_{\mathrm{acc},m,j}^{(\mathrm{UL})}$. Note that the transition probabilities between different states are unknown to each MEC server.

Similarly, let $r_{m,j}^{(\mathrm{B})}$ be the data rate of the backhaul link between MEC server m and cloud server j, we assume that $r_{m,j}^{(\mathrm{B})}$ follows a Markov process with finite number of states, given by $\left[r_{\min}^{(\mathrm{B})}, r_{\min}^{(\mathrm{B})} + l'\right], \left[r_{\min}^{(\mathrm{B})} + l', r_{\min}^{(\mathrm{B})} + 2l'\right], \ldots, \left[r_{\max}^{(\mathrm{B})} - l', r_{\max}^{(\mathrm{B})}\right]$, where $r_{\min}^{(\mathrm{B})}$ and $r_{\max}^{(\mathrm{B})}$ are the predefined minimum and maximum possible values of $r_{m,j}^{(\mathrm{B})}$, respectively, and l' is the length of each interval. Same as $t_{\mathrm{acc},m,j}^{(\mathrm{B})}$, we define that $r_{m,j}^{(\mathrm{B})}$ is in state s' if $r_{m,j}^{(\mathrm{B})}$ falls into the s' th interval, i.e., $r_{m,j}^{(\mathrm{B})} \in \left[r_{\min}^{(\mathrm{B})} + (s'-1)l, r_{\min}^{(\mathrm{B})} + s'l\right]$.

With the data rate $r_{m,j}^{(\mathrm{B})}$ and the size of data to be transmitted $\sum_{i=1}^{N_m} a_{m,j} s_{m,j}$, the data transmission latency from MEC server m to cloud server j is $t_{\mathrm{trans},m,j}^{(\mathrm{B})} = \frac{\sum_{i=1}^{N_m} a_{m,j} s_{m,j}}{r_{m,j}^{(\mathrm{B})}}$. Considering the fact that the size of the output data of a task is small (e.g., a decision indicator), we neglect the latency for sending the outcome of a task back to an MEC server. In case such latency is non-negligible, the time for downloading the task outcome can be calculated in the same way as the uploading time.

Given the access latency, data transmission latency, and the computational latency, the total elapsed time for offloading the tasks of MEC server m to cloud server j and completing these tasks is $t_{\mathrm{off},m,j}^{(\mathrm{B})} = t_{\mathrm{acc},m,j}^{(\mathrm{UL})} + t_{\mathrm{trans},m,j}^{(\mathrm{B})} + t_{\mathrm{comp},m,j}^{(\mathrm{C})} +$

$t^{(\mathrm{DL})}_{\mathrm{acc},m,j}$. Recall that the time for executing the proportion of tasks at MEC server m is $t^{(\mathrm{E})}_{\mathrm{comp},m} = \frac{\sum_{i=1}^{N_m}(1-a_{m,i})s_{m,i}z_{m,i}}{c_m^{(\mathrm{E})}}$. Let $t_{m,j}$ be the latency for completing all the tasks at MEC server m when associated with cloud server j (i.e., $x_{m,j} = 1$). Given that a proportion of the tasks are offloaded to cloud server j and executed there, $t_{m,j}$ is impacted by the dependency of the tasks. Specifically, if all the tasks are independent of each other, they can be executed in parallel. Then, $t_{m,j}$ is determined by the latest set of tasks completed between MEC server m and cloud server j, which is given by

$$t_{m,j} = \max\{t^{(\mathrm{B})}_{\mathrm{off},m,j}, t^{(\mathrm{E})}_{\mathrm{comp},m}\} \tag{6}$$

If the tasks are inter-dependent, they have to be executed following certain orders, making the calculation of $t_{m,j}$ more complicated. For example, the tasks of a training simulation system may include user device localization, map downloading and updating, image/video processing, trajectory prediction, and training outcome generation. Obviously, the last task must rely on the outcomes of previous tasks. At the current stage, we consider a simple dependency pattern in which the tasks need to be executed sequentially. With such a dependency pattern, the output of one task is an input to the subsequent task. Based on the required order of execution, the set of tasks to be executed first (with proportion $\sum_{i=1}^{N_m}(1-a_{m,i})s_{m,i}z_{m,i}$) are assigned to be executed by MEC server m, and the remaining tasks (with proportion $\sum_{i=1}^{N_m} a_{m,i}s_{m,i}z_{m,i}$) are assigned to cloud server j. Then, the output of the tasks executed by MEC server m is used as the input to start executing the tasks offloaded to cloud server. This way, the computing at MEC server m and the offloading from MEC server m to cloud server j can be performed in parallel, resulting in reduced latency. Under this setting, the time elapsed before the execution at cloud server j is determined by the slower one between the computing at MEC server m and the offloading from MEC server m to cloud server j, which is given by $\max\left\{t^{(\mathrm{E})}_{\mathrm{comp},m}, t^{(\mathrm{UL})}_{\mathrm{acc},m,j} + t^{(\mathrm{B})}_{\mathrm{trans},m,j}\right\}$. Finally, $t_{m,j}$ is calculated by

$$t_{m,j} = \max\left\{t^{(\mathrm{E})}_{\mathrm{comp},m}, t^{(\mathrm{UL})}_{\mathrm{acc},m,j} + t^{(\mathrm{B})}_{\mathrm{trans},m,j}\right\} + t^{(\mathrm{DL})}_{\mathrm{acc},m,j} + t^{(\mathrm{C})}_{\mathrm{comp},m,j} \tag{7}$$

Note that the optimal task partitioning is achieved when $t^{(\mathrm{B})}_{\mathrm{off},m,j} = t^{(\mathrm{E})}_{\mathrm{comp},m}$ for independent tasks and $t^{(\mathrm{E})}_{\mathrm{comp},m} = t^{(\mathrm{UL})}_{\mathrm{acc},m,j} + t^{(\mathrm{B})}_{\mathrm{trans},m,j}$ for sequentially dependent tasks.

2.2 Problem Formulation

We formulate the task offloading of MEC servers as a multi-agent reinforcement learning problem. Each MEC server acts as an agent who makes a series of decisions on task offloading (specifically, cloud server selection) over time. The decision-making process of each agent is modeled as a Markov Decision Process (MDP). For MEC server m, the system state is defined by the observed UL and DL access latencies $t^{(\mathrm{UL})}_{\mathrm{acc},m,j}$ and $t^{(\mathrm{DL})}_{\mathrm{acc},m,j}$, and the backhaul data date $r^{(\mathrm{B})}_{m,j}$ observed at the current time slot. The system state observed by MEC server m at time t is specified by a $1\times 3J$ vector $S_m(t) = [s^{(\mathrm{UL})}_{m,1}(t), \dots, s^{(\mathrm{UL})}_{m,J}(t), s^{(\mathrm{DL})}_{m,1}(t), \dots, s^{(\mathrm{DL})}_{m,J}(t), s'_{m,1}(t), \dots, s'_{m,J}(t)]$, where $\{s^{(\mathrm{UL})}_{m,j}(t)\}$, $\{s^{(\mathrm{DL})}_{m,j}(t)\}$, and $\{s'_{m,j}(t)\}$ ($j \in \{1, \cdots, J\}$) are the states of $t^{(\mathrm{UL})}_{\mathrm{acc},m,j}$, $t^{(\mathrm{DL})}_{\mathrm{acc},m,j}$, and $r^{(\mathrm{B})}_{m,j}$ at time t, respectively. The action space of MEC server m is specified by a combination of two variables $\{A_m(t), B_m(t)\}$, which corresponds to the strategies of cloud server selection and task partitioning ratio of MEC server m. Recall that the association between MEC m and various cloud servers is specified by a set of binary variables $[x_{m,1}(t), \dots x_{m,J}(t)]$, where $x_{m,j} = 1$ indicates that MEC server m offloads a proportion of tasks to cloud server j and $x_{m,j} = 0$ indicates otherwise. Given the constraint that at most one cloud server will be selected (i.e., $\sum_{j=1}^{J} x_{m,j} \le 1$), the number of possible values of $A_m(t)$ is J+1 and we index them by

$$A_m(t) \in \begin{cases} j & \text{if cloud server } j \text{ is selected} \\ 0 & \text{if no cloud server is selected} \end{cases}, m \in 1,\dots,M; j \in 1,\dots J. \tag{8}$$

The proportion of tasks (measured in total required CPU cycles) to be offloaded to cloud server, given by $\frac{\sum_{i=1}^{N_m} a_{m,i}s_{m,i}z_{m,i}}{\sum_{i=1}^{N_m} s_{m,i}z_{m,i}}$. Considering that MEC server m may receive different kinds of tasks over time, $\frac{\sum_{i=1}^{N_m} a_{m,i}s_{m,i}z_{m,i}}{\sum_{i=1}^{N_m} s_{m,i}z_{m,i}}$ may take any value between

0 and 1. Thus, if we directly set the value of $\frac{\sum_{i=1}^{N_m} a_{m,i} s_{m,i} z_{m,i}}{\sum_{i=1}^{N_m} s_{m,i} z_{m,i}}$ as the variable for task partitioning in the action space (i.e., $B_m(t) = \frac{\sum_{i=1}^{N_m} a_{m,i} s_{m,i} z_{m,i}}{\sum_{i=1}^{N_m} s_{m,i} z_{m,i}}$), the dimension of action space would be extremely high. To reduce the complexity of training, we set $B_m(t)$ to take a relatively small number of discrete values between 0 and 1, e.g., {0,0.1,0.2,…,0.9,1}. Once a certain value of $B_m(t)$ is selected, the task assignment is set such that $\frac{\sum_{i=1}^{N_m} a_{m,i} s_{m,i} z_{m,i}}{\sum_{i=1}^{N_m} s_{m,i} z_{m,i}}$ is the closest to the selected value of $B_m(t)$, i.e., $|B_m(t) - \frac{\sum_{i=1}^{N_m} a_{m,i} s_{m,i} z_{m,i}}{\sum_{i=1}^{N_m} s_{m,i} z_{m,i}}|$ is minimized. Obviously, there is a tradeoff between performance and complexity when different resolution of $B_m(t)$ are selected. The reward of MEC server m at each time slot is set to be $-t_{m,j}$. The objective is to find the optimal policy that maximizes the expected long-term accumulated discounted reward.

2.3 Deep Q-Learning Solution

A reinforcement learning (RL) agent aims to learn from the environment and take action to maximize its long-term cumulative reward. The environment is modeled an MDP with state space $\mathcal{S}$ and a RL agent can take actions from space $\mathcal{A}$. The agent interacts with the environment by taking actions, observing the reward and system state transition, and updating its knowledge about the environment. The objective of a RL algorithm is to find the optimal policy, which determines the strategy of taking actions under certain system states. A policy π is specified by $\pi(s|a) = P(A_t = a|S_t = s)$. In general, a policy is in a stochastic form to enable exploration over different actions. To find the optimal policy, the key component is to determine the value of each state-action function, also known as Q-function, which is defined by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{9}$$

Therefore, in large scale systems with large numbers of states and actions, the traditional Q-learning approach becomes infeasible since a table is required to store all the Q-values. In addition, traditional Q-learning needs to visit and evaluate every state-action pair, resulting in huge complexity and slow convergence. An effective approach to deal with such a challenge is to use a neural network (NN) to approximate the Q-values, given by $Q(s, a, \boldsymbol{w}) \approx Q_\pi(s, a)$, where $\boldsymbol{w}$ are the weights of the NN. By training a NN with sampled data, the NN can map the inputs of state-action pairs to their corresponding Q-values.

In our problem, each MEC server has a DQN to be trained and the DQN is used to generate task offloading decisions once the training is completed. The input layer of the DQN is the current system state, which consists of $2J$ neurons, each corresponds to an element of $S_m(t)$. The output layer is set to generate the values of Q-functions when taking all the J+1 actions, given by $Q(S_m(t), 1), Q(S_m(t), 2), \ldots, Q(S_m(t), J+1)$.

The direct application of NN in Q-learning may be unstable or even diverge due to the correlations between observations and the correlations between Q-values and target values. To deal with this challenge, we apply a mechanism called experience replay for the training of the DQN [12].

As mentioned in the problem formulation, due to the interaction between MEC servers that share the computational resource of the same cloud server, the task offloading strategies of different MEC servers are coupled, resulting in a multi-agent RL problem. A simple solution to such a problem is independent Q-learning (IQL) [13], where each agent (MEC server) regards the activities of other agents as part of the environment and learns the optimal policy based on the received rewards over time. In our problem, the MEC servers can learn to optimize their task offloading policies via the history of observed latency, hence gradually learn to select the proper cloud server. However, with IQL, all MEC servers are learning and adjusting their policies simultaneously, the environment from the perspective of each MEC server is non-stationary and there may be Ping-Pong effect. For example, two MEC servers may select the same cloud server at a time slot and experience high computational latency. Then, they select another same cloud server at the next time slot and observe high computational latency again. Hence, the system may take an extremely long time to converge.

To break such correlations, experience replay-based deep Q-network (DQN) is considered. The idea is to "frozen" the agent's experience for a certain time and use it to train the DQN later. Specifically, the agent first explores the environment by randomly taking actions and stores the experience, $e_t = (s_t, a_t, r_t, s_{t+1})$, in a target network. With the samples randomly drawn from the target network, the weights of the DQN is updated by minimizing the loss function given by

$$L_i(w_i) = \mathrm{E}_{(s,a,r,s')\in U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a', w_i^-) - Q(s, a, w_i)\right)^2\right] \quad (10)$$

where $\boldsymbol{w}_i$ and $\boldsymbol{w}_i^-$ are the weights of DQN and target network at iteration i, respectively. The loss function is the mean square error between DQN and target network, which can be minimized through stochastic gradient descent. To reduce the correlation between DQN and target network, the target network is updated less frequently. After the training of DQN, the agent then takes action based on the estimated Q-values. The framework of the experience replay based DQN training is shown in **Figure 1**.

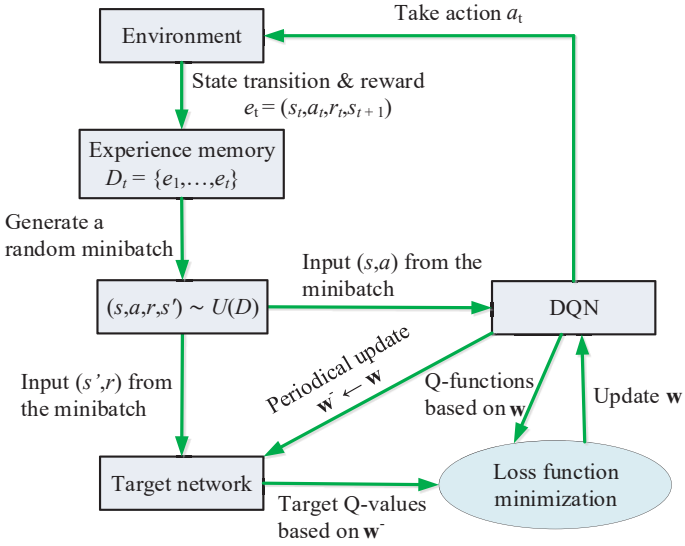


Figure 1 Experience replay-based DQN training.

With experience replay-based training that "frozen" each agent's experience, the MEC servers are not interacting with each other at the same pace and unable to learn from instantaneous feedback. As a result, the experience replay may be unstable, causing the system fail to convergence. To stablize training and accelerate convergence, a key observation is that the environment observed by an SU can be made stationary conditioned on the policies of other SUs. However, given that samples generated by experience replay are obsolete (i.e., cannot reflect current system dynamics), the MEC servers may not be able disambiguate the age of the sampled data from the replay memory. Importance sampling with off-environment training is an effective approach to tackle this issue [14][15]. The idea of importance sampling is assigning an importance ratio to each sample in a minibatch.

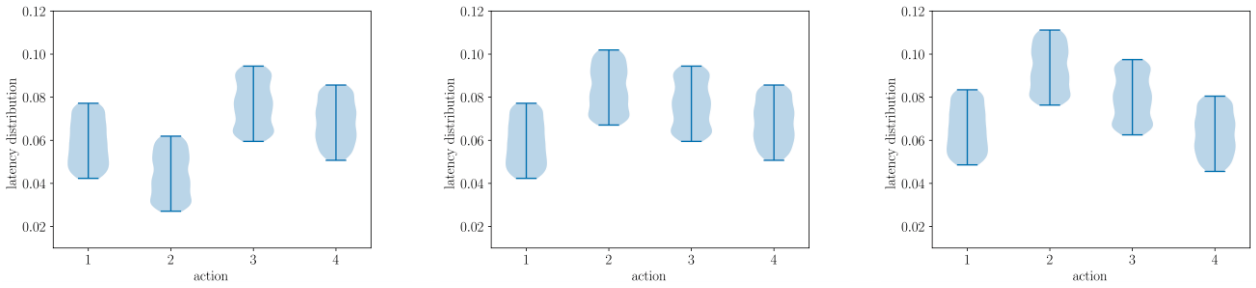
3. PRELIMINARY EVALUATION RESULTS

Due to the time limitation, we conducted a simple evaluation with using a simplified but efficient model instead of the deep Q-learning to prove the effectiveness of our proposed system. The model we used for evaluation is called Multi-armed bandit model [16][17][18], which is a classical but effective algorithm in RL. The objective of the model is to maximize the expected total reward over some time period. In a k-armed bandit problem, each of the k actions that has an expected or mean reward is given; the goal of the agent is to select a proper action form that maximizes the long-term expectation of the reward. In our scenario, the objective turns to choose a proper targeted server that minimize the latency for each arrived task. To conduct the evaluation experiment, we listed several assumptions:

1) The edge server and cloud server are undistinguishable, i.e., the task offloading controller considers the edge sever and cloud servers has the same distribution of the latency;
2) The offloading of a task would not affect the distribution of the latency. i.e., the decision of the task off-loading controller would not affect the environment;
3) The latencies of the servers are uniformly distributed, as well as the randomness of latency for each server;

4) As the RL algorithms maximize the reward, the application pursuits the minimum latency. The relation between the latency and the reward is $R = e^{-T}$, where R is the immediate reward, and T is the latency in second (s).

To simulate the scenario where the distribution of the latency changes, two perturbations are added at t=300 and t=600, where, at t=300 we assume that a congestion occurs on the link between the edge server and the cloud server which is the best initial off-loading target (or the computational resource in the edge server is exhausting, and the latency increases). At t=600, we assume that a small random perturbation in the whole system occurs; the best action may or may not change at this time. One example for this process of latency distribution change in a four-sever scenario is shown in Figure 2.



(a) The original distribution of the latencies. The best action $a = 2$ in this case.

(b) $t = 300$, a congestion occurs at server 2, the best action changes to $a = 1$.

(c) $t = 600$, a small random perturbation is injected into the whole system, the best action changes to 4.

Figure 2 An example of latency distribution change in a four-server scenario.

Therefore, we conducted a simulation experiment with four servers, and the results are shown in Figure 3. Three algorithms are tested in this group of experiments. The blue line represents the constant step-size ε-greedy algorithm, the yellow line represents the sample average ε-greedy algorithm, and the green line is a naïve algorithm. The subfigure on the top shows the average reward of each algorithm in 2000 runs. The middle subfigure shows the average latency, and the last subfigure shows the ratio between the times of the best action chosen by the algorithm to the real best action (2000).

The strategy of the naïve algorithms is simply testing every action at the beginning of each run and chooses the sever which as the shortest latency, then it uses this server forever. This strategy performs good at the beginning of the simulations but fails when the distribution of the latency changes because it lacks flexibility. One can see that both sample average and constant step-size ε-greedy algorithms are able to seek the suboptimal sever when the best original link congested or seek new optimal sever when a small random perturbation occurs in the system. However, the constant step-size ε-greedy algorithm is more resilience to both perturbations. It consists with the conclusion that the constant step-size ε-greedy algorithm is adaptive to the nonstationary environment.

According to the experiments, the constant step-size ε-greedy algorithm shows good performance on adaptively choosing the best servers for the arriving tasks even the distribution of the latency in the system changes. We also prove that the RL method works for the task offloading in the MEC environment. However, at this stage, we assume the choice of the task offloading does not affect the latency of future tasks, which is simplified for this preliminary evaluation specifically. We are currently working on replacing the multi-armed bandit model with deep Q-learning to explore whether the assumptions can be removed and further performance improvement will be achieved or not.

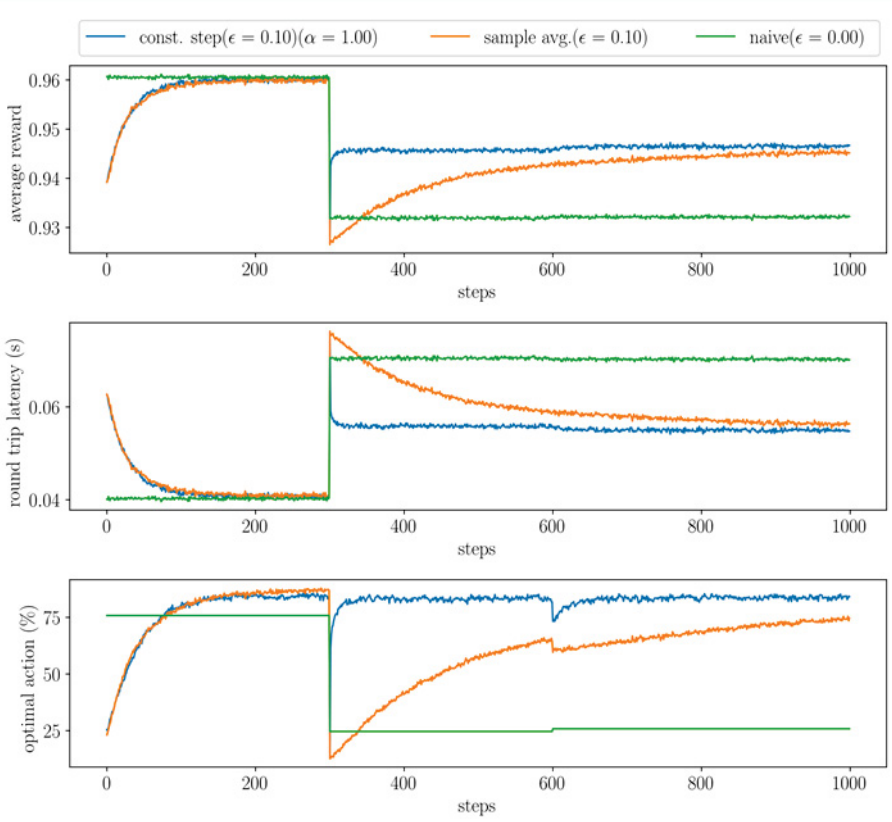


Figure 3 The simulation results of the four-server scenario.

4. CONCLUSION

In this work, we proposed a deep reinforcement learning based task offloading method for the computational intensive tasks within the MEC environment. We provided the detailed design and the model description with mathmetical analysis and modeling. We also built a simulation environment to conduct a simplified validation and evaluation experiment to explore the feasibility of our proposed solution. Preliminary experiment results showed that our solution outperforms than the naïve task offloading strategy. We are currently working on two main tasks. The first one is comparing our simplified model with other conventional taks offloading algorithms. The second one is replacing the simplified multi-armed bandit model with deep Q-learning model for the task offloading decision making. More comprehensive results will be carried out in our next paper.

REFERENCES

[1] Shi, Weisong, et al. "Edge computing: Vision and challenges." IEEE internet of things journal 3.5 (2016): 637-646.
[2] Peng, K., Leung, V., Xu, X., Zheng, L., Wang, J. and Huang, Q., 2018. A survey on mobile edge computing: Focusing on service adoption and provision. Wireless Communications and Mobile Computing, 2018.
[3] Zhao, T., Zhou, S., Guo, X., Zhao, Y. and Niu, Z., 2015, December. A cooperative scheduling scheme of local cloud and internet cloud for delay-aware mobile cloud computing. In 2015 IEEE Globecom Workshops (GC Wkshps) (pp. 1-6). IEEE.
[4] Hu, Y.C., Patel, M., Sabella, D., Sprecher, N. and Young, V., 2015. Mobile edge computing—A key technology towards 5G. ETSI white paper, 11(11), pp.1-16.
[5] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari, “Deep reinforcement learning for dynamic multichannel access in wireless networks,” IEEE Trans. on Cognitive Communications and Networking, vol. 4, no. 2, pp. 257-265, June 2018.

[6] Chen, M. and Hao, Y., 2018. Task offloading for mobile edge computing in software defined ultra-dense network. IEEE Journal on Selected Areas in Communications, 36(3), pp.587-597.
[7] Xia, Q., Liang, W., Xu, Z. and Zhou, B., 2014, December. Online algorithms for location-aware task offloading in two-tiered mobile cloud environments. In 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (pp. 109-116). IEEE.
[8] Jia, M., Cao, J. and Yang, L., 2014, April. Heuristic offloading of concurrent tasks for computation-intensive applications in mobile cloud computing. In 2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS) (pp. 352-357). IEEE.
[9] Imai, S. and Varela, C.A., 2011, November. Light-weight adaptive task offloading from smartphones to nearby computational resources. In Proceedings of the 2011 ACM symposium on research in applied computation (pp. 146-152).
[10] Cao, Y., Jiang, T. and Wang, C., 2014. Optimal radio resource allocation for mobile task offloading in cellular networks. IEEE Network, 28(5), pp.68-73.
[11] Geng, Y., Hu, W., Yang, Y., Gao, W. and Cao, G., 2015, November. Energy-efficient computation offloading in cellular networks. In 2015 ieee 23rd international conference on network protocols (ICNP) (pp. 145-155). IEEE.
[12] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529-533, Feb. 2015.
[13] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in Proc. ICML'93, pp. 330-337, 1993.
[14] J. Foerster, N. Nardelli, G. Farquhar, T. Afouras, P. H. S. Torr, P. Kohli, and S. Whiteson, "Stabilising experience replay for deep multi-agent reinforcement learning," in Proc. ICML'17, pp. 1146-1155, 2017.
[15] K. A. Ciosek and S. Whiteson, "Offer: Off-environment reinforcement learning," in Proc. AAAI'17, pp. 1819-1825, 2017.
[16] Auer, P., Cesa-Bianchi, N., Freund, Y. and Schapire, R.E., 2002. The nonstochastic multiarmed bandit problem. SIAM journal on computing, 32(1), pp.48-77.
[17] Vermorel, J. and Mohri, M., 2005, October. Multi-armed bandit algorithms and empirical evaluation. In European conference on machine learning (pp. 437-448). Springer, Berlin, Heidelberg.
[18] Kuleshov, V. and Precup, D., 2014. Algorithms for multi-armed bandit problems. arXiv preprint arXiv:1402.6028.