A Distributed Computation Offloading Strategy for Edge Computing Based on Deep Reinforcement Learning

Hongyang Lai[1], Zhuocheng Yang[1], Jinhao Li[1], Celimuge Wu[2](✉), and Wugedele Bao[3]

[1] University of Electronic Science and Technology of China, Chengdu, China
[2] The University of Electro-Communications, Tokyo, Japan
celimuge@uec.ac.jp
[3] Hohhot Minzu College, Hohhot, China

Abstract. Mobile edge computing (MEC) has emerged as a new key technology to reduce time delay at the edge of wireless networks, which provides a new solution of distributed computing. But due to the heterogeneity and instability of wireless local area networks, how to obtain a generalized computing offloading strategy is still an unsolved problem. In this research, we deploy a real small-scale MEC system with one edge server and several smart mobile devices and propose a task offloading strategy for one subject device on optimizing time and energy consumption. We formulate the long-term offloading problem as an infinite Markov Decision Process (MDP). Then we use deep Q-learning algorithm to help the subject device to find its optimal offloading decision in the MDP model. Compared with a strategy with fixed parameters, our Q-learning agent shows better performance and higher robustness in a scenario with an unstable network condition.

Keywords: Mobile edge computing · Computation offloading · Markov Decision Process · Deep reinforcement learning

1 Introduction

With the development of Internet of Things (IoT) technology, the number of smart devices increases explosively. These IoT devices have the functions of sensing, computation, and communication which can be connected to the Internet and collaboratively implement various applications, such as home automation, health monitoring, automated industry, and smart transportation. By 2025, it is estimated there will be an installed base of 75.44 billion IoT connected devices worldwide [1].

Since IoT devices are usually limited by computing and storage capabilities. In traditional methods, most computing tasks are offloaded to the cloud, which is called mobile cloud computing (MCC) [7]. However, due to the relatively large

C. T. Calafate et al. (Eds.): MONAMI 2021, LNICST 418, pp. 73–86, 2022.
https://doi.org/10.1007/978-3-030-94763-7_6

distance between IoT devices and cloud servers, this leads to high transmission delays and affects delay-sensitive applications [14]. In the past decade, through the rapid development of artificial intelligence (AI) technologies such as computer vision and natural language processing, the number of delay-sensitive intelligent applications largely increases. But the latency of public cloud providers usually exceeds 100 milliseconds, which is unacceptable [12]. Mobile edge computing (MEC) has emerged as new architecture and a key technology for IoT networks to face this problem. MEC [9,16] can provide computing services at the edge of wireless networks with low latency. Therefore, in recent years, we have witnessed a paradigm shift from centralized MCC to MEC. Compared with MCC, MEC can provide lower latency and computational agility in computational offloading. For each smart mobile device (SMD), it can migrate its computing tasks to edge nodes or cloudlets [17]. However, computing power means high expenses and energy consumption, which is not economically friendly. In addition, computational offloading can cause greater interference in ultra-dense networks and unexpected transmission delays [4]. Therefore, offloading all the computing tasks to the MEC server is not always the optimal strategy.

In a multi-user MEC scenario, computing offloading involves three parts: application partitioning, task allocation, and task execution [13]. In general, an offloading decision is to make a viable task allocation decision, which may result in any of the three types of offloading strategies: local execution, full offloading, and partial offloading, which is a trade-off result between time and energy consumption. To minimize the weighted sum of tasks completion time and energy consumption, a SMD should determine not only whether and how much to offload, but also offloading target. Such a problem can be generally formulated as integer programming problems due to the existence of binary offloading variables.

Many related engineering models for offloading decision problems and resource allocation problems in MEC networks show attractive theoretical results. Lei [11] considered it as a dynamic programming problem and formulated it into a continuous-time Markov decision process model. Chen [3] proposed a game theory for multi-user situations and made a trade-off between energy consumption and time delay, trying to achieve a Nash equilibrium. Dinh [5] also modeled it with game theory and extended the problem to a practical scenario, where the number of processed CPU cycles is time-varying and unknown. In this case, he applied Q-learning to learn SMDs' long-term offloading strategies to maximize their long-term utilities.

In most of the related works, the MEC system is stable and fixed, where it is uncomplicated to calculate the time consumption. However, a real MEC network is heterogeneous and flexible, whose structure may change at any time, and therefore it is impossible to reach a general solution. Hence, existing integer programming algorithms are not suitable for making real-time offloading decisions in a real MEC network. To improve the real-time performance, it is more practical to design a multi-user offloading mechanism, where SMDs can learn and adjust their offloading strategy at any time based on the reward and network information observed after each offloading action. In addition, most researches are based on simulations and analytical evaluations, which are not rigorous enough. The processing time for the same task varies from one SMD to another, which is not linear additive while offloading.

In this paper, we deploy a real small-scale multi-user MEC system, which has unique characteristics and topological network structure. All the SMDs have a certain ability of communication and computation and have the option to process computing tasks received from other SMDs. For instance, we use the Raspberry Pi as a SMD and apply YOLO algorithm to perform image analysis on multiple photos taken by the camera and offload the photos from SMD (Raspberry Pi) to MEC server connected to AP or other SMDs (other Raspberry Pi). To simulate the time variation and instability of a MEC system, we manually connect or disconnect SMDs at a random time. We measure the authentic processing time and calculate its power consumption according to its offloading strategy. Then, for the MEC system, the offloading process is formulated as a Markov decision process (MDP) model. Iterative methods such as Q-learning can be used to solve the above-mentioned MDP model. However, a challenge known as the curse of dimensionality [15] lies in the convergence of Q-table when simply applying Q-learning. To overcome the problem of dimensionality, we apply a better estimation alternative is to adopt a neural network which is known as Deep Q-Network (DQN). Eventually, the optimized offloading time and energy consumption can be obtained by employing the DQN to solve the MDP.

There are three main contributions of this article as follows.

1) The proposed offloading strategy not only includes local computing and MEC servers but also takes the computing ability of other SMDs into account, which contributes to practical optimization in industrial applications.
2) We consider both time delay and energy consumption and apply Deep Q-Network to solve the MDP model, giving a general offloading strategy for different scenarios.
3) Unlike most related works using simulation software to verify the offloading scheme, we build an actual MEC network system. In this case, the network structure and communication reliability may vary at any time. Besides, each offloading decision affects the whole MEC system, for example, CPU using rate of another SMD. In our work, we provide real data to evaluate its performance.

The remainder of this paper is organized as follows. In Sect. 1, the system model including network model and computation model will be introduced. Section 2 presents the offloading time and energy optimization according to MDP model and DQN strategy. Experiment settings and results are discussed in Sects. 3 and 4. Section 5 concludes the paper.

2 MEC System Model and Problem Formulation

2.1 MEC Network Model

We deploy a real MEC system for the IoT application, in which a MEC server connected to the access point (AP) provides service for N smart mobile devices (SMD) in an ad-hoc network as shown in Fig. 1, where the N is a flexible number because other SMDs may be offline at any time except for the subject smart mobile device (SSMD) in blue color. We choose Raspberry Pi as the smart mobile devices comprised of computing, communication, and storage modules. Considering a scenario of computer vision, the main task of this system is to analyze images using YOLO [2] in a real-time monitoring system. We suppose that the SSMD has several pictures to be processed as the original data, which is the basic task unit that cannot be divided anymore. However, Raspberry pi has a limited computing capability while a monitoring system is usually time-sensitive. Also, in the real-time monitoring system, since it is designed to operate constantly, the energy consumption cannot be ignored. To meet the demands for shorter delay and energy saving, the original data can be offloaded to the MEC server connected to AP or other SMDs for remote computing, to improve time efficiency and reduce energy consumption. The decision on computation process for executing tasks can be described as an action a.

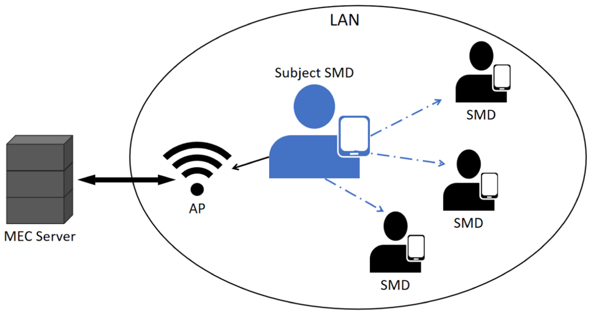


Fig. 1. Network model. (Color figure online)

$$a = \begin{cases} 0, & \text{computing locally} \\ 1, & \text{offloading to MEC server} \\ 2, & \text{offloading to another SMD} \\ 3, & \text{standby} \end{cases} \tag{1}$$

For example, $a = 1$ represents offloading the task to MEC server and $a = 2$ means offloading to another SMD. When offloading, the original data are transmitted and then the processed data will be sent back.

2.2 Computation Model

In the proposed network model, when SSMD has to complete i_{th} computation task $\tau = \{d_i, c_i\}$, where d_i is the size of data under processing and c_i represents the total number of CPU cycles required to complete the computing task. As for SSMD, it can choose either to execute all the tasks locally or to offload them to another SMD or the MEC server to execute remotely or remains in the standby state.

(1) Local Computing: We consider that the CPU in SSMD is operating at frequency f^L. While executing locally on SSMD, the computation time delay t^L is

$$t^L = \frac{c_i}{f^L} \tag{2}$$

The energy consumption of SSMD can be calculated as

$$e^L = \kappa (f^L)^3 t^L \tag{3}$$

where κ is a coefficient that depends mostly on the chip architecture and $\kappa = 10^{-26}$ [6,19].

(2) Edge Computing: In the transmission process of offloading, the achievable uplink transmission rate can be expressed as

$$r = log_2(1 + \frac{P^T * h}{\sigma^2}) \tag{4}$$

where P^T is the transmission power and h is the channel gain between SSMD and MEC server or SSMD and another SMD. σ^2 is the noise power. Let f^C denotes the number of CPU cycles frequency of the MEC server connected to AP or another SMD. In this case, the total time delay consists of transmission time and computation time, which can be expressed as

$$t^E = \frac{d_i}{r} + t^C \tag{5}$$

where $t^C = c_i/f^C$ means the computation time in MEC server or another SMD. The energy consumption of the MEC server or another SMD can be calculated as

$$e^E = P^T * \frac{d_i}{r} + P^C * t^C \tag{6}$$

where P^C is the computation power consumption of the MEC or another SMD.

(3) Standby: We consider there is a standby mode for CPU in SSMD, which means no offloading and computation actions will be performed. In this case, the total standby time is t^S and the energy consumption during standby state is

$$e^S = t^S * P^S \tag{7}$$

where we consider P^S is a small value shows the basic power consumption of idle CPU in SSMD. The introduction of the standby state is based on the instability and latency of the proposed network. When the last task is already offloaded, since the communication time is much lower than the computation time, the current calculation progress is unknown until SSMD obtains the returned result. So it may not be the optimal way to continue to perform task offloading or computation, then SSMD turns to the standby state to wait and reduce power consumption.

2.3 Problem Formulation

When making offloading decisions, both the total execution time of the task and the total power consumption need to be taken into consideration. To make a trade-off between those two factors, we define a weighting factor $w (w \in [0, 1])$ to indicate the degree of importance to time delay and power consumption. To meet the needs of specific scenarios, the weighting factor can be adjusted to emphasize a certain aspect [8].

According to (2) and (3), the weighted sum G^L of time and energy consumption in the process of computing locally, can be defined as

$$G^L = wt^L + (1 - w)e^L \tag{8}$$

Similarly, according to (5) and (6) the energy consumption and time delay in the MEC server and in another SMD can be calculated respectively as

$$G_M^E = wt_M^E + (1 - w)e_M^E \tag{9}$$

$$G_A^E = wt_A^E + (1 - w)e_A^E \tag{10}$$

Lastly, according to (7), the weighted sum G^S of energy and time consumption in the standby state, can be defined as

$$G^S = wt^S + (1 - w)e^S \tag{11}$$

Therefore, the overhead of the subject SMD can be obtained by

$$\begin{aligned} G &= w(\sum^{s_1} t^L + \sum^{s_2} t_M^E + \sum^{s_3} t_A^E + \sum^{s_4} t^S) \\ &+ (1 - w)(\sum^{s_1} e^L + \sum^{s_2} e_M^E + \sum^{s_3} e_A^E + \sum^{s_4} e^S) \\ &= w(T^L + T_M^E + T_A^E + T^S) + (1 - w)(E^L + E_M^E + E_A^E + E^S) \\ &= wT + (1 - w)E \end{aligned} \tag{12}$$

where s_1, s_2, s_3, s_4 are the counters for recording the number of actions the trade-off has performed in order to complete all the tasks of SSMD and T, E represent the total sum of time and energy consumption in executing all the tasks of SSMD.

Then in the designated network model, to investigate the tradeoff between energy and time consumption for SSMD, we can formulate the optimization problem as follows.

$$\begin{aligned} \min \quad & w_1 \frac{T - T_{Exp}}{T_{Exp}} + w_2 \frac{E - E_{Exp}}{E_{Exp}} \\ \text{s.t.} \quad & C1 : 0 \leq T_{Exp} \leq T^{max} \\ & C2 : 0 \leq E_{Exp} \leq E^{max} \\ & C3 : w_1 + w_2 = 1 \end{aligned} \tag{13}$$

Here we introduce the expected minimum of total delay T_{Exp} and total energy consumption E_{Exp} to normalize the objective function. Constraint $C1$ and $C2$ limit the range of those manually giving value. Constraint $C3$ restricts the sum of weight factors to 1.

3 Offloading Time and Energy Optimization

In this section, we use MDP to represent the task execution and offloading process of the entire MEC system and use a deep reinforcement learning algorithm to solve the optimization problem.

3.1 MDP Model Formulation

In this section, we will analyze the stochastic process of the system state and formulate a Markov Decision Process (MDP) problem to minimize the normalized weighted sum of total time delay and power consumption in offloading process. When SSMD makes an offloading decision, it not only considers the current state but also considers the impact of the current decision on the future total reward. MDP considers the immediate and delayed rewards brought about by current decisions, and makes the expected optimal action under uncertain circumstances. There are five crucial elements in our MDP model: decision epoch, state, action, state transition probability, and reward function. Considering our MEC system model and offloading process, the details are as follows.

Decision Epochs: The period SSMD making offloading decision is called an epoch [18]. In the continuous decision epochs, time is described as sequence T, consisting of K discrete time slots, where K denotes the number of executing tasks.

$$T = \{1, 2, ..., t, ..., K\}, t \in T \tag{14}$$

States: A state includes the communication quality, the number of returned results, the number of remaining tasks, the number of tasks being processed, and the operating frequency of all the SMDs and the MEC server. The state vector S involves all possible states during the offloading process for the MEC system and is defined as follows.

$$S = N_{all} \times CQ \times RR \times RT \times PT \times f \tag{15}$$

where $\times$ represents the Cartesian product. N_{all} includes all the nodes of the proposed network, $N_{all} = \{N_{SSMD}, N_{MEC}, N_{ASMD}\}$. CQ denotes the communication quality by the possibility of channel failure. RR, RT and PT represent the numbers of returned results, tasks that remain and are under processing, respectively. f denotes the operating frequency of SMDs and the MEC server. All the states values can be normalized and range from $[0, 1]$. In this case, for any certain decision epoch t in (14), the current state can be described as s, a 3×5 dimension vector. An example state table is shown in Table 1.

Table 1. An example table of the current state.

	CQ	RR	RT	PT	f
SSMD	0.9	0.5	0.5	0.1	0.5
MEC	0.8	0.7	0.4	0.2	0.6
ASMD	0.7	0.4	0.2	0.2	0.7

Actions: For each epoch, SSMD needs to select a corresponding action in the action space (1), where $a_t = \{0, 1, 2, 3\}$. $a_t = 0$ denotes that for the i_{th} computation task it is executed locally in SSMD. $a_t = 1$ denotes offloading to the MEC server. $a_t = 2$ denotes offloading to another SMD (ASMD). $a_t = 3$ denotes that SSMD turns into standby state. The set of all actions selected to complete the task is defined as A.

$$A = \{a = (a_1, a_2, ..., a_t)\} \tag{16}$$

Transition Probability: The probability of Markov state may transfer from s to state s' by selecting action a. In our case, the state is 3×5 dimension vector. In order to simplify the model of transition probability, we take the MEC server node as the focus node from all the nodes and its operating frequency is also fixed. The transition probability of MEC server node is derived by

$$P(s'|s,a) = \begin{cases} P(RR'|RR)P(RT'|RT)P(PT'|PT), & CQ' = CQ \\ \rho, & CQ' \neq CQ \end{cases}$$

where $\rho \in [0, 1]$ represents the failure probability that the channel between MEC server and SSMD. P(RR′—RR) denotes the probability of the number of returned results in the next state. P(RR′—RR) is described by

$$P(RR'|RR) = \begin{cases} \eta, & RR' = RR \\ \xi, & RR' \neq RR \end{cases}$$

where η represents the probability of the number of returned results staying at the same number, $0 \leq \eta \leq 1$. Alternatively, ξ denotes the probability of the number of returned results changes to another number, and ξ is expressed as follows

$$\xi = (1 - \eta)/RT \tag{17}$$

P(RT′—RT) denotes the probability of the number of remaining tasks in the next state. P(RT′—RT) is described by

$$P(RT'|RT) = \begin{cases} \phi, & RT' = RT \\ \varphi, & RT' \neq RT \end{cases}$$

where ϕ represents the probability of the number of remaining tasks staying at the same number, $0 \leq \phi \leq 1$. Alternatively, φ denotes the probability of the number of remaining tasks changes to another number, and φ is expressed as follows

$$\varphi = (1 - \phi)/(RT - 1) \tag{18}$$

P(PT′—PT) denotes the probability of the number of processing tasks in the next state. P(PT′—PT) is described by

$$P(PT'|PT) = \begin{cases} \lambda, & PT' = PT \\ \mu, & PT' \neq PT \end{cases}$$

where λ represents the probability of the number of processing tasks staying at the same number, $0 \leq \lambda \leq 1$. Alternatively, μ denotes the probability of the number of processing tasks changes to another number, and μ is expressed as follows:

$$\mu = (1 - \lambda)/(RT + PT - 1) \tag{19}$$

Based on the above equations, the transition probability of the focus node and then all the nodes can be calculated and formulated as a state vector.

Reward Function: In order to evaluate the instant benefit of choosing an action in the current state, we introduce the reward function $r(s, a)$, which reflects the instant reward when SSMD makes an offloading decision in the current MEC system state. According to the objective optimization function (13) mentioned before, the reward function $r(s, a)$ is defined as follows

$$r(s, a) = \begin{cases} G^L, & a = 0 \\ G^E_M, & a = 1 \\ G^E_A, & a = 2 \\ G^S, & a = 3 \end{cases}$$

where $\{G^L, G^E_M, G^E_A, G^S\}$ are already defined by (8–11), respectively.

In our case, iterative methods such as Q-learning can be used to solve the above-mentioned MDP model. Therefore, we will discuss the use of Q-learning and Deep Q-Network in solving MDP model as follows.

3.2 Deep Q-Network

In the Q-learning algorithm, a Q-table is used to store Q-values of all state-action pairs [10]. According to Bellman equation, Q-learning $Q(s_t, a_t)$ can be expressed as

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma(\max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))] \tag{20}$$

where α is the learning rate, γ is the discount factor. In traditional Q-learning, we use a table to record Q-value $Q(s, a)$ of all the state-action pairs as shown in (20). However, there are too many latent states of our practical MEC system to fit in the Q-table, which is generally known as the curse of dimensionality [15]. Therefore, we can't store states through tables. It is necessary for us to compress the dimensions of the state. One solution is to approximate the value function approximation. Then a neural network is built for fitting and regression, approximate the Q-values function of different state-action pairs via prediction.

The discrete action mentioned above is Q-value function, where the value function Q here is not a specific value, but a set of vectors. In the neural network, the weight of the network is θ, which is represented by a Q-value function $Q(s, a, \theta)$, and the θ after the final neural network converges is the value function. Therefore, the core of the whole process determines the θ to approximate the value function. We adopt the most classic method, gradient descent, to minimize the loss function to continuously adjust the network weight θ. The loss function is defined as

$$L_i(\theta_t) = E_{(s,a,r,s^t)}[(r(s_t, a_t) + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2] \tag{21}$$

θ'_i denotes the target network parameter of the i_{th} iteration and θ_i denotes the network parameter of evaluation network. The next step is to find the gradient of θ, as follows

$$\frac{\partial L_t(\theta_t)}{\partial \theta_t} = E_{(s,a,r,s^t)}[(r(s_t, a_t) + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2] \tag{22}$$

In addition, during the learning process, trained quadruple is stored in a replay memory, where the training network structure is read by mini-batch during the learning process, which is called the experience replay. In this scenario, the reason we add experience replay is that the states are monitored consecutively under the MEC system, so the relevance of the samples is too large. Without experience replay, a problem that the gradient descent will be in the same direction with a continuous period occurs. In this condition, the gradient may not converge under the same step size. Therefore, experience replay is to randomly select some experience from a memory pool and then find the gradient, thus avoiding this underlying problem.

The target network and evaluation network share the same structure with different parameters, both of which consist of two fully connected layers. The structure of the DQN is shown in Fig. 2.

As shown in Algorithm 1, we first initialize the MEC system and all the network parameters. A few pre-train steps are set to get some experiences through random actions stored in the experience pool, and the training of the Q-evaluation network is completed in the pre-training stage. During training, it extracts minibatch experience from the experience pool every step to train the Q network. Eventually, update the parameters of the target network with the evaluation network's every C steps.

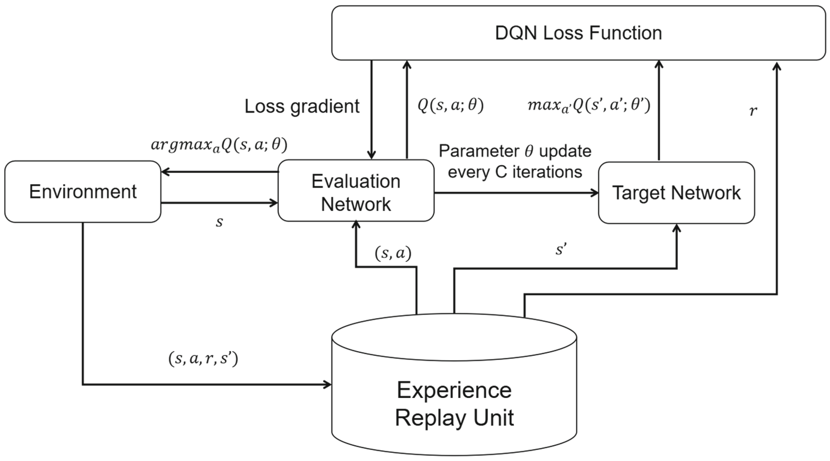


Fig. 2. DQN structure.

4 Experiment

4.1 Experiment Settings

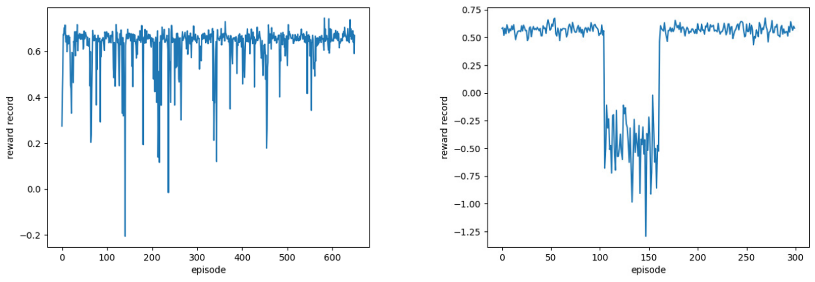
In this part, we deploy a real MEC system to conduct the experiment. The experimental network model consists of one SSMD, three other SMDs, and the MEC server. We use Raspberry Pi 4b (Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5 GHz) as smart mobile devices, and its power consumption is at 6.4w for regular operating and, at 2.7w for idle mode. The MEC server offers a GPU of Geforce GTX 1650 Ti. All the mobile devices are deployed in one ad-hoc based on IEEE 802.11ac in a scattered way in a radius of about 1.5 m. To simulate a real-time monitoring system, 16 unprocessed 256 * 256 images are generated in the SSMD per second as one episode. We have tested that all the SMDs and the MEC server satisfy the requirement of running YOLO algorithm to perform image analysis.

4.2 Experiment Results

To simulate a long-term unstable scenario, we set the episode number with 650, DQN memory batch size with 200, and keep the communication quality and computation stability of SMDs varying. The normalized reward value shows as Fig. 3(a). The figure illustrates a dynamic approach to the optimal strategy for this long-term dynamic MEC system, whose expected reward value increases monotonically over episodes and converges to about 0.67. Besides, it shows that the reward value declines quite frequently and with large amplitude in the early stage, indicating the learning and adapting period of DQN. To show its robustness, we set another comparing group with manual parameters that fix the current system well. It seems to be stable at first, but as the MEC system status

Algorithm 1. Deep Q-learning algorithm

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $Q'$ with weights $\theta' = \theta$
for episode=1,$M$ do
  Initialize the MEC system with all the images unprocessed
  while tasks are not fully processed do
    Observe the system state $s_t$
    With probability $\epsilon$ select a random action $a_t$
    Otherwise select $a_t = arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ as an offloading decision and observe reward $r_t$
    Store transition $(s_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $\phi_j, a_j, r_j, \phi_{j+1}$ from $D$
      Set $y_j = \begin{cases} r_j, & \text{if episode terminates at } j+1 \\ r_j + \gamma \max_{a'} Q'(\phi_{j+1}, a'; \theta'), & \text{otherwise} \end{cases}$
    Perform a gradient descent step on the loss $Q(s_i, a_i; \theta_t) - y_i)^2$ with respect to
    the network parameter $\theta$
    Reset $Q' = Q$ every $r(s_t, a_t)C$ steps
  end while
end for
```



(a) DQN algorithm (b) Manual parameter strategy

Fig. 3. Performance comparison.

change goes beyond its scope at around 100 episodes, the whole strategy crashed. Due to its fixed parameter, the agent cannot make any proper decision until we set new parameters at around episode 160. Without preset parameters, our DQN agent can choose appropriate offloading actions corresponding to MEC system states after sufficient working episodes. With the prolonged working episodes under the same system, the decisions made are more precise and less likely to be interfered by other random interferences.

5 Conclusion

In this paper, we design a practical MEC system with computation time and energy taken into account and propose an offloading strategy based on MDP model. We formulate a minimization problem that minimizes the weighted sum of the average delay and power consumption of the subject smart mobile device. We apply Deep Q-learning algorithm to learn from the offloading experience of a certain past period to improve its offloading action and predict viable decisions under unknown circumstances through regression. Compared with other traditional computation strategies, our method shows higher robustness. Especially even though the MEC system is flexible, the algorithm can make adjustments and offloading decisions in time. In our future work, we may design a larger-scale network with multiple MEC servers and take the resource allocation problem into consideration.

Acknowledgement. This research was supported in part by the National Natural Science Foundation of China under Grant No. 62062031, in part by Inner Mongolia natural science foundation grant number 2019MS06035, and Inner Mongolia Science and Technology Major Project, China, in part by ROIS NII Open Collaborative Research 21S0601, and was supported in part by JSPS KAKENHI grant numbers 18KK0279, 19H04093, 20H00592, and 21H03424.

References

1. Ahmad, I.: Discover Internet of Things editorial, inaugural issue. Disc. Internet Things **1**(1), 1–4 (2021). https://doi.org/10.1007/s43926-021-00007-6
2. Bochkovskiy, A., Wang, C.Y., Liao, H.Y.M.: YOLOv4: optimal speed and accuracy of object detection. arXiv preprint arXiv:2004.10934 (2020)
3. Chen, X., Jiao, L., Li, W., Fu, X.: Efficient multi-user computation offloading for mobile-edge cloud computing. IEEE/ACM Trans. Netw. **24**(5), 2795–2808 (2015)
4. Deng, M., Tian, H., Lyu, X.: Adaptive sequential offloading game for multi-cell mobile edge computing. In: 2016 23rd International Conference on Telecommunications (ICT), pp. 1–5 (2016). https://doi.org/10.1109/ICT.2016.7500395
5. Dinh, T.Q., La, Q.D., Quek, T.Q., Shin, H.: Learning for computation offloading in mobile edge computing. IEEE Trans. Commun. **66**(12), 6353–6367 (2018)
6. Guo, S., Liu, J., Yang, Y., Xiao, B., Li, Z.: Energy-efficient dynamic computation offloading and cooperative task scheduling in mobile cloud computing. IEEE Trans. Mob. Comput. **18**(2), 319–333 (2018)
7. Guo, S., Xiao, B., Yang, Y., Yang, Y.: Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In: IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications, pp. 1–9 (2016). https://doi.org/10.1109/INFOCOM.2016.7524497
8. Guo, S., Xiao, B., Yang, Y., Yang, Y.: Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In: IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, pp. 1–9. IEEE (2016)
9. Hu, Y.C., Patel, M., Sabella, D., Sprecher, N., Young, V.: Mobile edge computing-a key technology towards 5g. ETSI White Paper **11**(11), 1–16 (2015)

10. Iqbal, A., Tham, M.L., Chang, Y.C.: Double deep q-network-based energy-efficient resource allocation in cloud radio access network. IEEE Access **9**, 20440–20449 (2021)
11. Lei, L., Xu, H., Xiong, X., Zheng, K., Xiang, W.: Joint computation offloading and multiuser scheduling using approximate dynamic programming in NB-IoT edge computing system. IEEE Internet Things J. **6**(3), 5345–5362 (2019)
12. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp: comparing public cloud providers. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC 2010, pp. 1–14. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1879141.1879143
13. Lin, L., Liao, X., Jin, H., Li, P.: Computation offloading toward edge computing. Proc. IEEE **107**(8), 1584–1607 (2019). https://doi.org/10.1109/JPROC.2019.2922285
14. Pan, J., McElhannon, J.: Future edge cloud and edge computing for internet of things applications. IEEE Internet Things J. **5**(1), 439–449 (2018). https://doi.org/10.1109/JIOT.2017.2767608
15. Peng, S.: Stochastic Hamilton-Jacobi-Bellman equations. SIAM J. Control. Optim. **30**(2), 284–304 (1992)
16. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1), 30–39 (2017). https://doi.org/10.1109/MC.2017.9
17. Satyanarayanan, M., Chen, Z., Ha, K., Hu, W., Richter, W., Pillai, P.: Cloudlets: at the leading edge of mobile-cloud convergence. In: 6th International Conference on Mobile Computing, Applications and Services, pp. 1–9. IEEE (2014)
18. Yang, G., Hou, L., He, X., He, D., Chan, S., Guizani, M.: Offloading time optimization via Markov decision process in mobile-edge computing. IEEE Internet Things J. **8**(4), 2483–2493 (2020)
19. Zhang, W., Wen, Y., Guan, K., Kilper, D., Luo, H., Wu, D.O.: Energy-optimal mobile cloud computing under stochastic wireless channel. IEEE Trans. Wirel. Commun. **12**(9), 4569–4581 (2013). https://doi.org/10.1109/TWC.2013.072513.121842