Hindawi
Wireless Communications and Mobile Computing
Volume 2023, Article ID 4665067, 12 pages
https://doi.org/10.1155/2023/4665067

Research Article

Dependent Task-Offloading Strategy Based on Deep Reinforcement Learning in Mobile Edge Computing

Bencan Gong and Xiaowei Jiang

College of Computer and Information Technology, China Three Gorges University, Yichang, 443000 Hubei, China

Correspondence should be addressed to Bencan Gong; 190026892@qq.com

Received 29 September 2022; Revised 5 November 2022; Accepted 18 November 2022; Published 4 January 2023

Academic Editor: Jun Cai

In mobile edge computing, there are usually relevant dependencies between different tasks, and traditional algorithms are inefficient in solving dependent task-offloading problems and neglect the impact of the dynamic change of the channel on the offloading strategy. To solve the offloading problem of dependent tasks in a dynamic network environment, this paper establishes the dependent task model as a directed acyclic graph. A Dependent Task-Offloading Strategy (DTOS) based on deep reinforcement learning is proposed with minimizing the weighted sum of delay and energy consumption of network services as the optimization objective. DTOS transforms the dependent task offloading into an optimal policy problem under Markov decision processes. Multiple parallel deep neural networks (DNNs) are used to generate offloading decisions, cache the optimal decisions for each round, and then optimize the DNN parameters using priority experience replay mechanism to extract valuable experiences. DTOS introduces a penalty mechanism to obtain the optimal task-offloading decisions, which is triggered if the service energy consumption or service delay exceeds the threshold. The experimental results show that the algorithm produces better offloading decisions than existing algorithms, can effectively reduce the delay and energy consumption of network services, and can self-adapt to the changing network environment.

1. Introduction

With the advent of the Internet era, smart devices are widely used in our lives. Due to the limited computing power of mobile devices, they sometimes cannot satisfy users' needs. In addition, mobile devices processing massive calculation tasks can lead to excessive energy consumption, which results in bad users' experience [1]. To solve these problems, mobile cloud computing has been brought up. Massive calculation tasks can be offloaded to the cloud by mobile devices, where the cloud server performs computing and returns the results to the terminal [2]. Because the cloud server is far from the terminal, the task transmission delay is high. At the same time, increasingly, terminal devices upload data to the cloud server, which also brings high pressure to the network. Therefore, mobile edge computing (MEC) came into being [3]. MEC offloads computing tasks from end devices to edge servers that are closer to them for computing to reduce network pressure, data transmission delay, and end device energy consumption [4, 5]. How to make offloading decisions is crucial, and the effectiveness of offloading decisions depends on key indicators, such as energy consumption and delay [6]. There are 2 forms of offloading, partial offloading and fully offloading [7].

In recent years, there have been many research results for the offloading of computing tasks for MEC. Fu and Ye [8] described the offloading problem as a delay minimization problem under energy consumption constraints. An improved firefly swarm optimization algorithm was proposed to generate the offloading decision, which significantly reduced the system cost. Zhu and Wen [9] defined the weighted sum of energy consumption and delay as an optimization function of total overhead. An offloading strategy based on an improved genetic algorithm was proposed, which achieved better results on delay and load balance, but not on energy consumption reduction. Wei et al. [10] proposed a maximum energy-saving priority algorithm to reduce the energy consumption, which used greedy

selection to solve the optimization problem, but only considered energy consumption and was not suitable for delay sensitive scenarios. Yang et al. [11] and Guo and Liu [12] proposed game theory-based offloading algorithm that described the overhead minimization problem as a policy game to reduce the energy consumption and delay of each mobile device through joint optimization.

With the development of machine learning, great progress has been made in using machine-learning algorithms to solve the computing offloading problem [13]. Liang et al. [14] proposed a Distributed Deep Learning-based Offloading (DDLO) algorithm that used multiple parallel deep neural networks to generate offloading decisions. The network parameters were continuously updated using an experience replay mechanism. The algorithm generates near-optimal offloading decisions in a short time, but the authors only considered the static network scenario. Li et al. [15] proposed a deep reinforcement learning algorithm for the complex computing offloading problem in collaborative computing with heterogeneous edge computing servers. The algorithm optimized the offloading decision based on the real-time state of the network and the properties of the task to minimize the task delay, but the authors only considered the delay, not energy consumption. Zhou et al. [16] used deep reinforcement learning to study the joint optimization problem of computing offloading and resource allocation in dynamic multiuser MEC systems, using the DDQN (Double Deep Q Network) algorithm to dynamically generate offloading decisions. In reference [17], a DDQN-based trajectory and phase shift optimization method was proposed to maximize RIS-UAV network capacity. In reference [18], a new incentive-driven and deep Q network-based method (IDQNM) was proposed for designing mobile node incentive mechanisms and content-caching strategies in D2D offloading. The reference [19] proposed an incentive mechanism based on delay constraints and reverse auction. With the maximization of mobile network operators' revenue as the optimization objective, two optimization methods were proposed: the greedy winner selection method (GWSM) and the dynamic planning winner selection method (DPWSM). T. Yang and J. Yang [20] proposed a joint optimization method for offloading decisions and resource allocation, which improved the DQN algorithm, shortened the finish time of computing tasks, and reduced the terminal energy consumption. Zhu et al. [21] proposed a dynamic resource allocation strategy based on K-means, where resources were modeled as "fluids" and allocated using an auction algorithm. The throughput of the edge server was improved and the transmission delay was reduced. Above algorithms have not taken task dependency into account, while task dependency is truly existed in practical applications.

The representative research results on the offloading strategies of dependent tasks are as follows. Dong et al. [22] proposed a computational-offloading strategy based on genetic algorithms. This strategy encoded the offloading location and offloading order of tasks, used delay and energy consumption as evaluation criteria, and continuously optimized the offloading decision by variation and crossover operations. However, it did not consider the resource allocation of edge servers. The fine-grained offloading problem with multiple users and multiple servers was studied in [23]. The authors considered the fine-grained offloading of Internet of Things (IoT) devices as a multiconstrained objective optimization problem and proposed an improved Nondominated Sorting Genetic Algorithm (NSGA-II) with the objective of minimizing the average delay. Mao et al. [24] proposed a delay-acceptance-based offloading strategy for multiuser tasks. The strategy firstly used a nondominated genetic algorithm to solve the optimal solution in a single-user scenario for each user, then improved the convergence speed with a probabilistic selection mechanism and nondominated judging scheme, and finally proposed an adjustment strategy based on the idea of time delay acceptance in stable matching. It solved the multiuser-offloading problem with dependent task scenarios. Liu et al. [25] proposed an energy-efficient collaborative task-offloading algorithm based on semidefinite relaxation and stochastic mapping, which generated offloading decisions for dependent tasks in static network environments and reduced the total energy consumption of IoT devices.

Above literatures only consider the offloading of dependent tasks in static network environments, but in fact the network environment is dynamic. In this paper, we use deep reinforcement learning methods to generate dependent task-offloading decisions, cache the optimal decisions for each round, and then optimize DNN parameters using priority experience replay mechanism to extract valuable experiences. DTOS introduces a penalty mechanism, which is triggered if the service energy consumption or service delay exceeds the threshold. The algorithm can reduce delay and energy consumption. When the network transmission rate changes, the model can generate the corresponding optimal-offloading decision and adapt to the changing network environment by only obtaining the current network rate in real time. The main contributions of this paper are as follows:

(1) A dependent task-offloading model is built for the scenario of a dynamic network environment. The optimization objective of DTOS is derived by comprehensively considering the delay and energy consumption of the service

(2) Transform the dependent task offloading into an optimal policy problem under Markov decision processes. A deep reinforcement learning-based dependent task-offloading strategy is proposed, which can obtain optimal task-offloading decisions using priority experience replay mechanism and penalty mechanism

(3) The effectiveness of DTOS is verified through simulation experiments. According to the simulated experiment results, DTOS is effective and better than the other four algorithms

2. DTOS Model

2.1. System Model. The model in this paper is built on the scenario of multiple IoT devices and a single-edge server,

the system model is shown in Figure 1. IoT devices can be smart devices, wireless sensors, and other network-connected devices. Unmanned Aerial Vehicle (UAV) is adopted as the edge server. Suppose that we need to execute an IoT service S, and that the service needs to be computed collaboratively by K IoT devices. Each IoT device has a certain amount of computing power that can execute computing tasks locally. Each IoT device connects to the edge server through a wireless network, and the wireless transmission rate from each IoT device to the edge server varies and is unstable because the UAV is moving at a low speed in the area. The K fine-grained computational tasks of the IoT service S are equally distributed in K different IoT devices. There are data dependencies between different computational tasks and these tasks are indivisible, each computational task is either computed locally or all uploaded to the edge server for computation, i.e., binary offloading. The offloading decision at time t is represented as a list $a_t = \{x_0, x_1, \cdots, x_{K-1}\}$ with length K. x_i determines whether the task i should be offloaded to the edge server for computation. When $x_i = 0$, it means that the task is executed locally. When $x_i = 1$, the task will be offloaded to the edge server for execution. In this paper, the optimization goal is to minimize the delay and energy consumption to generate the offloading decision to determine whether the task is executed at the local.

2.2. Communication Model. The wireless transmission rate from each IoT device to the edge server at time t is denoted as $R_t = \{v_0, v_1, \cdots, v_{K-1}\}$, where v_i denotes the wireless transmission rate from the IoT device i to the edge server, and the wireless transmission rates of all devices are different from each other. Assume that the flight altitude of the UAV is constant H, its position at time t is denoted as $\text{loc\_UAV}(t) = (x_t, y_t, H)$. The position of device i is denoted as $\text{loc\_UE}_i = (x_i, y_i, 0)$, thus the distance between the UAV and the IoT device i at time t is denoted as

$$D_{\text{it}} = \sqrt{H^2 + \text{loc\_UAV}(t) - \text{loc\_UE}_i^{\,2}}. \tag{1}$$

It is assumed that the UAV and IoT devices are modulated by Orthogonal Frequency Division Multiplexing (OFDM) and accessed by Time Division Multiple Access (TDMA). The wireless channels between UAV and IoT devices are line-of-sight channels. Therefore, the channel gain $h_i(t)$ between the UAV and the IoT device i at time t can be expressed as follows:

$$h_i(t) = \varphi_0 D_{\text{it}}^{-2} = \frac{\varphi_0}{H^2 + \text{loc\_UAV}(t) - \text{loc\_UE}_i^{\,2}}, \tag{2}$$

where φ_0 denotes the channel power gain at 1 m.

According to the Shannon formula, v_i is calculated as follows:

$$v_i = B \log_2\left(1 + \frac{P_i^{\text{send}} h_i(t)}{\sigma^2}\right), \tag{3}$$

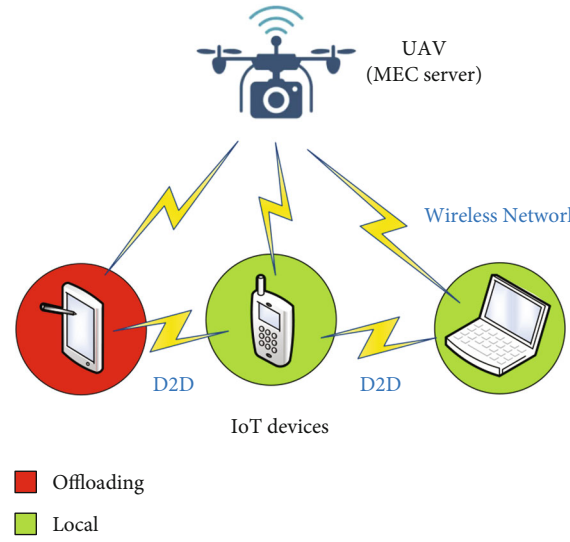


Figure 1: System model for dependent task offloading.

where B is the channel bandwidth of device i; P_i^{send} is the transmission power of device i; and σ^2 denotes the variance of additive Gaussian white noise.

If the task i is offloaded to the edge server for calculation, the transmission delay is calculated as follows:

$$T_i^{\text{send}} = \frac{d_i}{v_i}, \tag{4}$$

where d_i denotes the data size of task i. After the task is finished computing at the edge server, the result is transmitted back to the IoT device.

The energy consumption of device i during the task transmission to the edge server is calculated as follows:

$$E_i^{\text{send}} = P_i^{\text{send}} T_i^{\text{send}} \tag{5}$$

2.3. Computing Model

2.3.1. Local-Computing Model. If f_i^{local} is defined as the computing resource of device i, the local computational delay of task i is as follows:

$$T_i^{\text{local}} = \frac{d_i w}{f_i^{\text{local}}}, \tag{6}$$

where w is the computation complexity parameter of the task.

Defining P_i^{local} as the local-computing power, the local-computing energy consumption of device i is as follows:

$$E_i^{\text{local}} = T_i^{\text{local}} P_i^{\text{local}} \tag{7}$$

2.3.2. Edge-Computing Model. Define f^{edge} as the total computing resources of the edge server and f_i^{edge} as the

computing resources allocated to task i by the edge server, so the edge-computing delay of task i is as follows:

$$T_i^{\text{edge}} = \frac{d_i w}{f_i^{\text{edge}}} \tag{8}$$

When the task is offloaded to the edge server for calculation, the device i is in an idle state. During this period, the energy consumption generated by device i is as follows:

$$E_i^{\text{edge}} = T_i^{\text{edge}} P_i^{\text{free}}, \tag{9}$$

where P_i^{free} is the power of device i when it is idle.

2.4. Task Dependency Model. The task dependency is considered as input dependency in this paper, i.e., the execution of some tasks requires the execution results of other tasks as input. The task dependency can be modeled as a directed acyclic graph, where each node represents a computational task. A single task can have multiple precursor and successor tasks. If the task has no successor, it is the final task, and the entire service is executed when its execution is finished.

Each successor task needs to wait all its precursor tasks to end, which are in parallel, so the execution time of successor task equals the maximum of the finish time of all precursor tasks. Denote the time when all tasks start to execute as $\text{RT} = \{\text{rt}_0, \text{rt}_1, \cdots, \text{rt}_{K-1}\}$, and the beginning node executes at 0. Denote the finish time of all tasks as $\text{FT} = \{\text{ft}_0, \text{ft}_1, \cdots, \text{ft}_{K-1}\}$, then the finish time of task i equals the following formula:

$$\text{ft}_i = \text{rt}_i + T_i, \tag{10}$$

where T_i is the execution delay of task i. It is calculated as follows:

$$T_i = (1 - x_i) T_i^{\text{local}} + x_i \left(T_i^{\text{edge}} + T_i^{\text{send}} \right). \tag{11}$$

The start execution time of task i is as follows:

$$\text{rt}_i = \begin{cases} \max\ ft_j & P(i) \neq \phi, j \in P(i), \\ 0 & P(i) = \phi, \end{cases} \tag{12}$$

where $P(i)$ is the set of direct predecessor tasks of task i. If $P(i)$ is the empty set, the task i is the starting task and its start execution time is 0.

2.5. Problem Description. The optimization objective of the offloading strategy is to minimize the weighted sum of the delay and energy consumption of the service. From the above model, the finish time of the entire service S can be deduced as follows:

$$T_s = \max\ \text{ft}_i, \text{ft}_i \in \text{FT}. \tag{13}$$

The energy consumption generated by task i is as follows:

$$E_i = (1 - x_i) E_i^{\text{local}} + x_i \left(E_i^{\text{edge}} + E_i^{\text{send}} \right). \tag{14}$$

The sum of the energy consumption generated by all tasks is as follows:

$$E_s = \sum_{i=0}^{k-1} E_i. \tag{15}$$

The algorithm introduces a penalty mechanism, which is triggered if the service energy consumption or service delay exceeds the threshold. The weighted sum of the energy consumption and delay generated by completing the service S is as follows:

$$\text{cost} = \alpha T_s + (1 - \alpha) E_s + \frac{g_t (T_s - T_M)}{T_M} + \frac{g_e (E_s - E_M)}{E_M}, \tag{16}$$

where α is the weight, T_M is the delay threshold, E_M is the energy consumption threshold, and $g_t \in \{0, 1\}$ and $g_e \in \{0, 1\}$ are the penalty factors for delay and energy consumption, respectively. When the penalty mechanism is not triggered, g_t and g_e are 0.

The optimization objective is to minimize the cost value, and the formula is:

$$\begin{aligned} &\text{Min cost,} \\ \text{Subject to}: \ & C_1 : x_i \in \{0, 1\}, i \in [0, K-1], \\ & C_2 : rt_i = \max ft_j, P(i) \neq \phi, j \in P(i), \\ & C_3 : rt_i = 0, P(i) = \phi, \\ & C_4 : \alpha \in [0, 1]. \end{aligned} \tag{17}$$

Constraint C_1 indicates that each task can only choose to be computed locally or offloaded to an edge server for computing. Constraint C_2 indicates that a task starts execution at the finish time of all its predecessor tasks. Constraint C_3 indicates that the execution time of the starting task is 0. Constraint C_4 indicates weight α is a number between 0 and 1.

3. DTOS Algorithm

3.1. Model Training Process of DTOS. Offloading optimization of dependent tasks in edge computing is an NP-hard problem [25], and reinforcement learning can continuously interact with the environment and can transform the offloading optimization problem of dependent tasks into an optimal policy problem in Markov decision-making. The model training process of DTOS is illustrated in Figure 2, and the main steps are as follows.

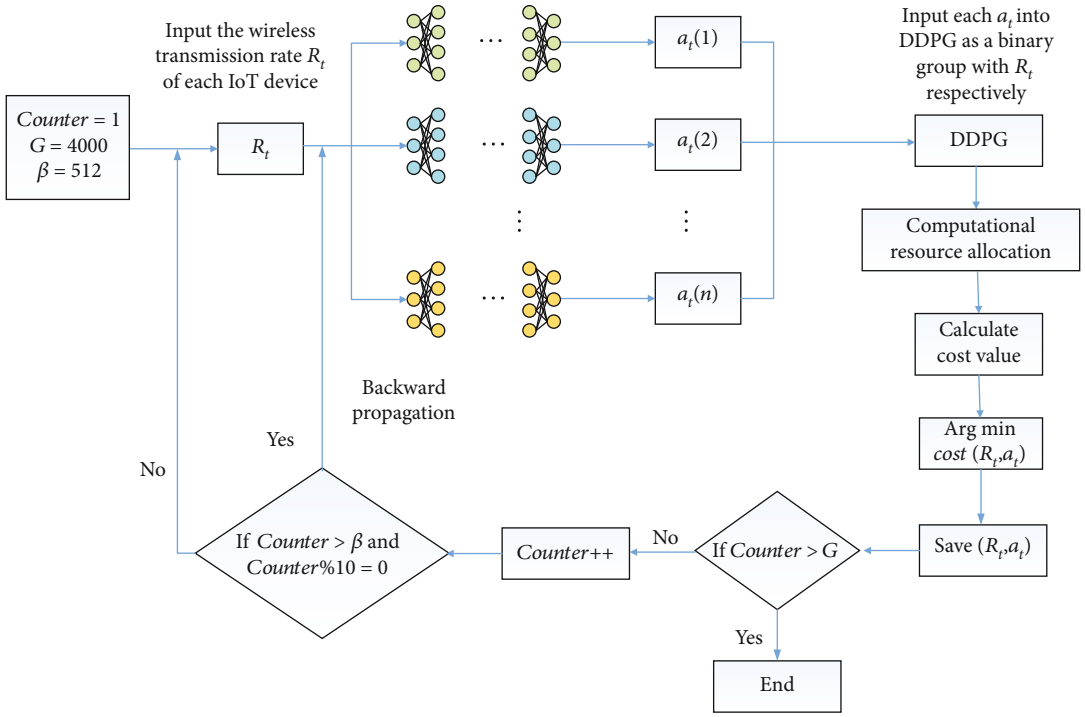


Figure 2: Model training process of DTOS.

3.1.1. Forward Propagation. Firstly, the experimental parameters are initialized and $n(n > 1)$ DNNs are constructed with randomly generated weight parameters and bias parameters. The number of neurons in the input and output layers of the DNN is the number of IoT devices K, and the number of neurons in the 1st hidden layer and the 2nd hidden layer is L_1 and L_2, respectively.

The wireless transmission rate $R_t = \{v_0, v_1, \cdots, v_{K-1}\}$ from each IoT device to the edge server at time t is used as the input layer data of the DNN and is input to the hidden layer for calculation, and all hidden layer neurons use ReLu as the activation function. The output layer neurons do not need an activation function, and different offloading decisions are directly after the data are calculated with the weight parameters and bias parameters.

Each DNN outputs different offloading decisions after one forward propagation is completed. The i-th DNN outputs the offloading decision as $a_t(i) = \{x_0, x_1, \cdots, x_{K-1}\}$. Since $x_i \in \{0, 1\}$, but the output of DNN is not always 0 or 1, the output of DNN needs to be transformed. The method is to convert each value greater than 0 in the array to 1, otherwise to 0, resulting in n offloading decisions.

3.1.2. Computational Resource Allocation. In existing studies, the computational resources of the edge server are usually distributed equally to all tasks offloaded to the edge server for execution, which will waste computing resources and increase the delay of edge computing in IoT services with dependent tasks.

Computational resource allocation is a continuous control problem, so in this paper, the edge server is used as an agent, and DDPG (Deep Deterministic Policy Gradient) is used to allocate computational resources of the edge server. Reinforcement learning has three key elements: state, action, and reward. The state consists of two components

$$\text{state} = [O, U]. \tag{18}$$

$O = [o_0, o_1, \cdots, o_{K-1}]$ denotes the network transmission rate of the device that needs to offload the task to the edge server, and the vector O is assigned values based on each offloading decision a_t generated by DNN and the wireless transmission rate R_t from the device to the edge server. If the task needs to be offloaded to the edge server for computation, the wireless transmission rate v_i from device i to the edge server will be assigned to o_i; otherwise, o_i is set to zero, indicating that the task is computed locally. U denotes the remaining computational resources of the edge server.

The action (Act) is the number of computational resources allocated by the edge server to the task of each device

$$\text{Act} = (0, U]. \tag{19}$$

At the start of task execution, DDPG selects a value from the remaining computational resources based on the current state as computational resources amount allocated to the task by the edge server.

The reward is determined by the current state and the current action with the following formula:

$$\text{reward} = R(\text{state}, \text{Act}). \tag{20}$$

The reward function $R(\text{state}, \text{Act})$ is associated with the objective function, and the optimization objective of resource allocation is to minimize the total delay of edge computing, which is calculated as follows:

$$T^{\text{edge}} = \sum_{i=0}^{k-1} T_i^{\text{edge}}. \tag{21}$$

The reward function is as follows:

$$R(\text{state}, \text{Act}) = -\left(T_i^{\text{edge}}\right). \tag{22}$$

The negative of the edge-computing delay of task i is used as the reward value, which can minimize the edge-computing delay. Each state-action group has a value $Q^*(\text{state}, \text{Act})$, which represents the expectation of the long-term reward obtained by performing the action Act under state. For the state-action group, its Q^* value is calculated and stored in the table. The Q^* value is updated by the following formula:

$$\begin{aligned} Q^*(\text{state}_t, \text{Act}_t) = {} & Q^*(\text{state}_t, \text{Act}_t) + \gamma \\ & \times \left[\text{reward}_t + \delta \times \max_{\text{Act}_{t+1}} (\text{state}_{t+1}, \text{Act}_{t+1}) \right. \\ & \left. - Q^*(\text{state}_t, \text{Act}_t)\right], \end{aligned} \tag{23}$$

where $Q^*(\text{state}_t, \text{Act}_t)$ is called the action value function, state_t is the system state at moment t, γ is the learning rate, and δ is the discount factor. We designs two deep neural networks, the action value network $Q(\text{state}_t, \text{Act}_t \mid \theta^Q)$ and the action network $\mu(\text{state}_t|\theta^\mu)$, where θ^Q and θ^μ are network parameters. The action network $\mu(\text{state}_t|\theta^\mu)$ is a mapping of the state space and action space and can directly generate the desired action according to the state. The action value network $Q(\text{state}_t, \text{Act}_t \mid \theta^Q)$ is used to approximate the action value function and can provide gradient for the training of the action network. The training of this action value network is to minimize the following loss function:

$$L\left(\theta^Q\right) = \left(\text{reward}_t + \gamma Q'\left(\text{state}_{t+1}, \text{Act}_{t+1}|\theta^{Q'}\right) - Q\left(\text{state}_t, \text{Act}_t|\theta^Q\right)\right)^2, \tag{24}$$

where Q' is the target value network. Q' synchronizes the weights from the Q network. The action network parameters are updated by the policy gradient algorithm, and the gradient update is as follows:

$$\begin{aligned} & \nabla_{\theta^\mu} Q\left(\text{state}, \text{Act}|\theta^Q\right)\Big|_{\text{state}=\text{state}_t, \text{Act}=\mu(\text{state}_t|\theta^\mu)} \\ & = \nabla_{\text{Act}} Q\left(\text{state}, \text{Act}|\theta^Q\right)\Big|_{\text{state}=\text{state}_t, \text{Act}=\mu(\text{state}_t|\theta^\mu)} \nabla_{\theta^\mu} \mu(\text{state}|\theta^\mu)\Big|_{\text{state}=\text{state}_t}. \end{aligned} \tag{25}$$

After inputting the state into the action network, the action network generates the action required for the current state, thus minimizing the computational delay of the edge server.

3.1.3. Generate Optimal Offloading Decisions. Each offloading decision a_t and the wireless transmission rate R_t are substituted into the model to obtain the cost value under each offloading decision. The offloading decision with the smallest cost value is selected as the output of this round, and the offloading decision a_t and the device transmission rate R_t are stored in memory as one data entry. Set a counter, and increase 1 when one data entry is stored.

3.1.4. Backward Propagation. Different from traditional supervised learning, the offloading decision in the dynamic network environment does not have a dataset with labels for neural network training. Thus, DTOS uses the data in memory to train DNNs through experience replay mechanisms. When the counter is higher than the set threshold β, the algorithm starts to perform backward propagation. In this paper, we use priority experience replay mechanism to extract valuable experiences. The lower the cost value of the sample, the higher the priority of the sample. The experience extraction is performed in a probabilistic way. The probability of each sample being selected is as follows:

$$\text{Pro}(j) = \frac{1/\text{cost}_j}{\sum_1^{\text{mem}} 1/\text{cost}_j}, \tag{26}$$

where mem is the number of samples in the current memory. The m data are selected in the memory cache as training data. Input the R_t of them again to each DNN to generate the offloading decision a_t^* and calculate the cost function $J(\omega, b)$ according to the following formula:

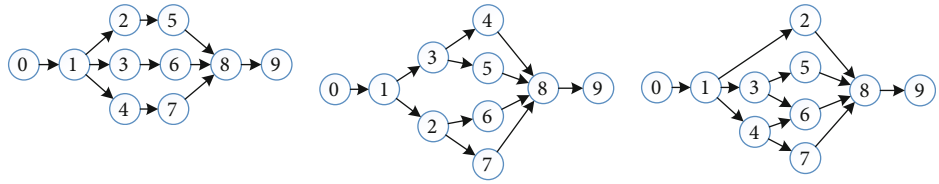
$$J(\omega, b) = \frac{1}{m} \sum_{i=1}^{m} L(a_t{}^*, a_t), \tag{27}$$

where $L(a_t^*, a_t)$ is the cross-entropy loss function, which is formulated as follows:

$$L(a_t{}^*, a_t) = -(a_t \log a_t{}^* + (1 - a_t) \log (1 - a_t{}^*)) \tag{28}$$

To minimize the cross-entropy loss, a gradient descent algorithm is used to optimize the network parameters of each DNN by performing one backpropagation after every 10 forward propagations. When the memory capacity is full, the oldest data entry will be discarded to store a new one. The quality of the training data will become higher and higher as the DNN parameters are continuously updated, and the deep reinforcement learning network also gets closer to the objective function. After reaching the set number of training rounds G, the algorithm terminates.

3.2. Usage of DTOS. After the model is trained, the edge server detects the wireless transmission rate of each device at the start of using DTOS and inputs them into the deep reinforcement learning network to generate the optimal-offloading decision.



(a) Network A (b) Network B (c) Network C

Figure 3: Three types of task dependency diagram.

Using the offloading coordinator in the edge server as the management module for computing offloading, the service is executed as follows:

(1) When an IoT service is started, the service number is transmitted to the edge server via remote command. The edge server loads the corresponding network model trained in advance according to the service number. The offloading coordinator detects the current network state, obtains the network transmission rate of each IoT device, and inputs it into the network model

(2) Execute DTOS, a deep reinforcement learning-based dependent task-offloading policy, to obtain the optimal-offloading decision

(3) Transmit the offloading decision to the corresponding IoT device

(4) The IoT device that receives the offloading decision determines whether the task is to be processed locally or uploaded to the server for processing based on the offloading decision. If it is uploaded to the server for processing, the edge server dynamically allocates computing resources

(5) The edge server finishes the calculation and transmits the result back to the IoT device

3.3. Computational Complexity Analysis of DTOS. The wireless transmission rate of IoT devices is input to the neural network, and the number of neurons in the input layer is K since the number of IoT devices is K. The number of neurons in the first hidden layer of the network is L_1, then the matrix operation from the input layer to the first hidden layer is $[L_1 \times K] \times [K \times 1]$, and the computational complexity of the matrix is $O(L_1K^2)$. The number of neurons in the second hidden layer of the network is L_2, then the matrix operation from the first hidden layer to the second hidden layer is $[L_2 \times L_1] \times [L_1 \times 1]$, and the computational complexity of the matrix is $O(L_2{L_1}^2)$. The number of neurons in the output layer is K. The matrix operation from the second hidden layer to the output layer is $[L_2 \times K] \times [K \times 1]$, and the computational complexity of the matrix is $O(L_2K^2)$. The number of DNNs is n, thus the computational complexity of DTOS forward propagation is $O(n((L_1 + L_2)K^2 + L_2{L_1}^2))$.

During the model training process of DTOS, the gradient descent algorithm is used to update the network parameters several times until the algorithm converges.

The computational complexity of each parameter update is equal to the computational complexity of forward propagation, so the computational complexity of DTOS is $O(nM((L_1 + L_2)K^2 + L_2{L_1}^2))$, and M is the size of batch training.

4. Experiments and Analysis

4.1. Experimental Parameters Set. Simulation experiments were performed using Python 3.8 and TensorFlow 2.2.0. Setting up the scenario where there exists 1 edge server with 10 IoT devices, there are three dependencies among the computing tasks, as shown in Figure 3.

Experimental parameter settings are shown in Table 1.

4.2. Effect of Weighting Factor α and Learning Rate γ. Figure 3(c) is used to investigate the effects of the weighting factor α and learning rate γ. In Figure 4, we show the effects of different weighting factors α on the service delay and energy consumption. When the weighting factor α is small, the DTOS optimization objective focuses more on the service energy consumption. The smaller the service power consumption is, the larger the reward value is. As the weighting factor α increases, the service delay will decrease, while the service energy consumption will increase. For different IoT services, users can adjust the weight factor α according to the delay and energy consumption requirements. To balance the effects of delay and energy consumption on cost values, the following experiments set the weighting factor α to 0.5.

For choosing the appropriate learning rate γ, the optimal-offloading decision generated by the exhaustive method is introduced as a comparison benchmark. The exhaustive method selects the optimal solution by enumerating all offloading decisions. Figure 5 shows the ratio of the cost values of DTOS and the exhaustive method, i.e., the gain ratio. As indicated in Figure 5, DTOS performs best and the gain ratio increases gradually when the learning rate is 0.01. The gain ratio of both offloading strategies approaches 1.0 after 3000 rounds, which indicates that the offloading decision generated by DTOS is close to the optimal solution of the exhaustive method. When the learning rate is 0.1, the algorithm cannot converge. Because the learning rate is too large and the parameter update ranges are too large, which causes the network to fail to converge to the optimal solution. When the learning rate is 0.001, because the learning rate is too small and the parameter update range is too small, the network cannot converge to the optimal solution in a short time. The optimal value of the learning rate is 0.01, so the learning rate is set to 0.01 in the following experiments.

Table 1: Experimental parameters.

Parameter	Value
Computing resources of the device if_i^{local}	[0.1, 0.5] G cycles/s
Total edge server computing resources f^{edge}	{18, 15, and 12} G cycles/s
Power at the device i idle P_i^{free}	[0.004, 0.04] W
The data size of the task i d_i	[30, 50] MB
The computational complexity of tasks w	600cycles/Kb
The device i wireless data transmission rate v_i	[0.1, 10] MB/s
Data sending power of the device iP_i^{send}	0.1 W
The local calculated power of the device iP_i^{local}	10 W
Delay threshold T_M	200
Energy consumption threshold E_M	200
Weighting factor α	0.5
Number of DNNs n	3
The memory size of experience replay	1024
Batch training size M	128
Learning rate γ	0.01
Number of training rounds G	4000
Threshold value β	512
Number of IoT devices K	10
Number of neurons in the first hidden layer L_1	120
Number of neurons in the second hidden layer L_2	80

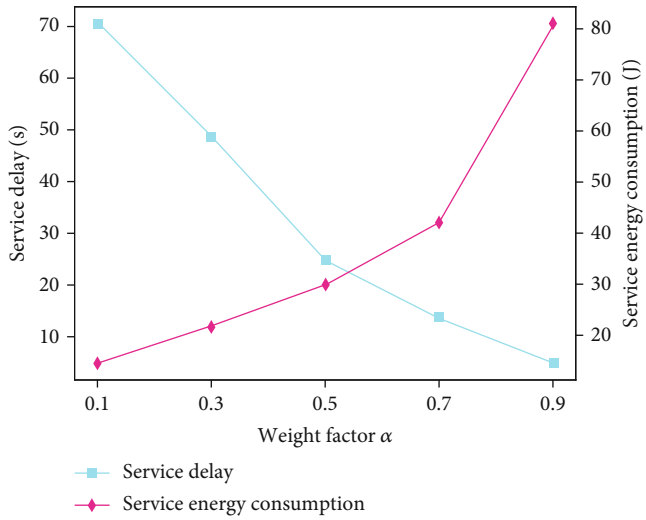


Figure 4: The effect of the weighting factor on service delay and energy consumption.

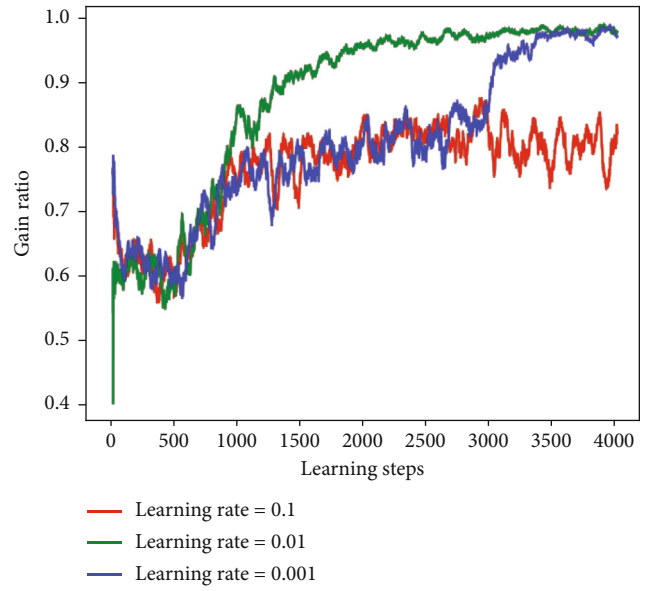
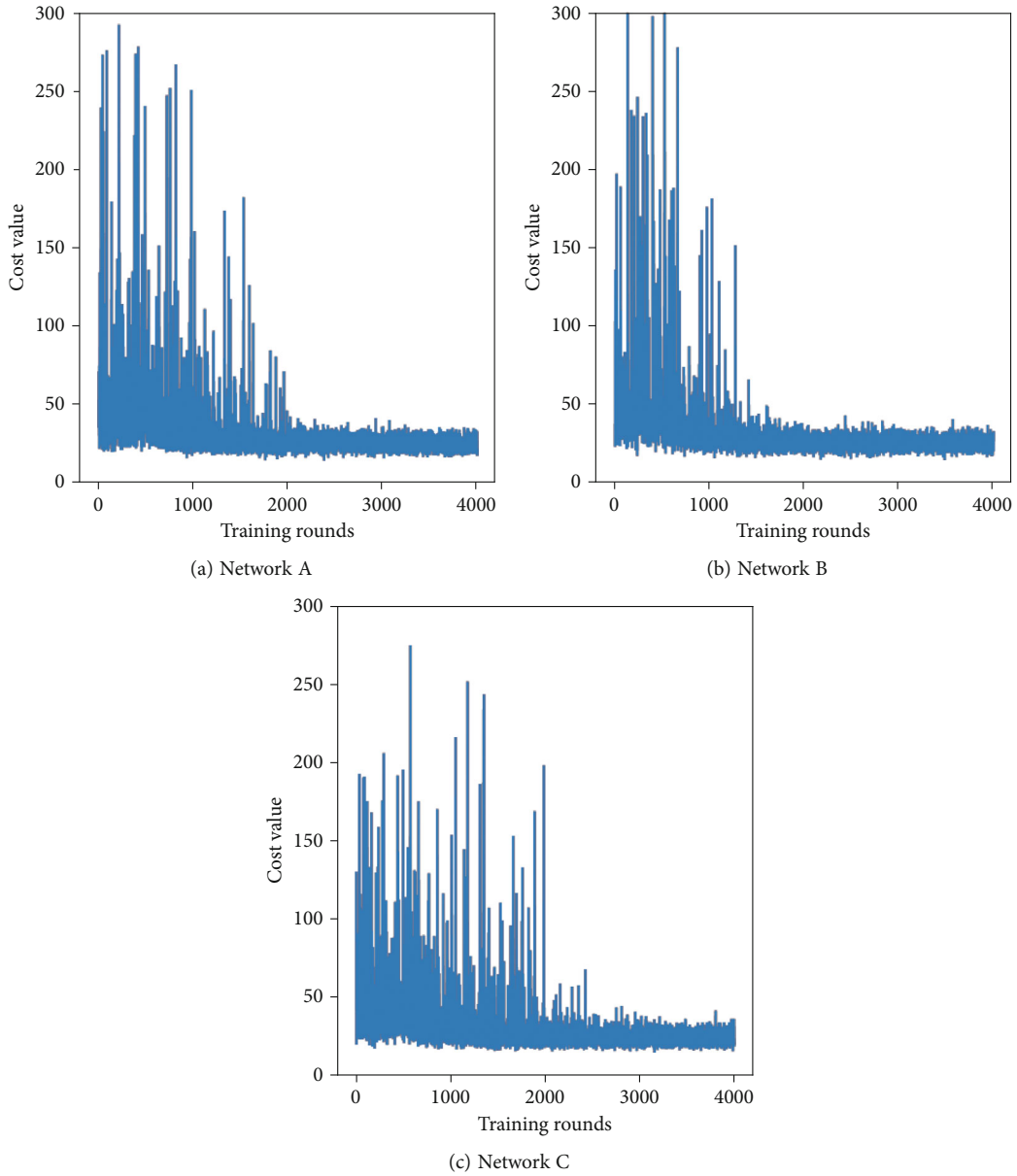


Figure 5: Gain ratio of DTOS with different learning rates.

4.3. Analysis of Experimental Results

4.3.1. Convergence of DTOS. The convergence of DTOS is firstly verified, and the experimental results are shown in Figure 6. Three networks with three different dependencies shown in Figure 3 are trained. The structure of networks A and B is simpler and they converge at rounds 2153 and 1769, respectively. The structure of network C is the most complex, which effects the speed of the algorithm. Network C converges at round 2808.

4.3.2. Cost Values of Each Algorithm with Different Dependencies. DTOS is compared with four other algorithms, which are the local computing-only algorithm (LOCAL), edge computing-only algorithm (EDGE), random selection algorithm (RADOM), and DDQN algorithm [16]. The local computing-only algorithm indicates that all tasks are computed by the IoT device. The edge computing-only algorithm indicates that all tasks are uploaded to the edge server for computation. The random selection algorithm indicates that all tasks are randomly selected to be computed by the edge server or the IoT device. The experiments are conducted 50 times and the results are averaged. The experimental results are shown in Figure 7. The cost of DTOS is the lowest for all various dependencies. Since the structure of network C is the most complex, its cost value is higher than the cost values of networks A and B. Since the algorithm LOCAL schedules all tasks to be computed locally, there is no transmission delay, but it causes excessive energy consumption. The algorithm EDGE offloads all tasks to the edge server for computation, which results in less energy consumption, but causes long transmission delay when the transmission speed is low. Both DTOS and DDQN train DNNs through experience replay mechanisms, and DDQN rewards DNNs based on the cost value of a single-offloading decision. In comparison, DTOS selects the optimal offload decision from multiple parallel offload decisions and uses priority experience replay mechanism and penalty mechanism



(a) Network A

(b) Network B

(c) Network C

Figure 6: Convergence diagram of DTOS algorithm under different dependencies.

according to their cost value; thus, DTOS has better performance. To further verify the effectiveness of DTOS, Figure 3(c) will be used for the subsequent experiments.

4.3.3. Cost Values of Each Algorithm with Different Task Data Sizes. The variations of the cost value according to the average data size of the computational task in different algorithms are shown in Figure 8. The cost values generated by each algorithm are positively correlated with the average data size during the growth of the average data size from 40 MB to 120 MB. The reason is that the larger data size generates greater latency and energy consumption, whether for local or edge computing. In comparison with other algorithms, DTOS has the slowest growth in cost value and the lowest cost value. It shows that DTOS outperforms all other four algorithms for different task data sizes.

4.3.4. Cost Values of Each Algorithm with Different Task Computational Complexity. Figure 9 shows the variation of cost values with different task computational complexities. The cost value is positively related to the task computation complexity for DTOS, local computing-only algorithm, and DDQN. The reason is that the increase in computational complexity of tasks leads to an increase in computation delay and energy consumption of IoT devices. However, for an edge computing-only algorithm or random selection algorithm, the cost value does not always increase when the computational complexity increases. Since the edge

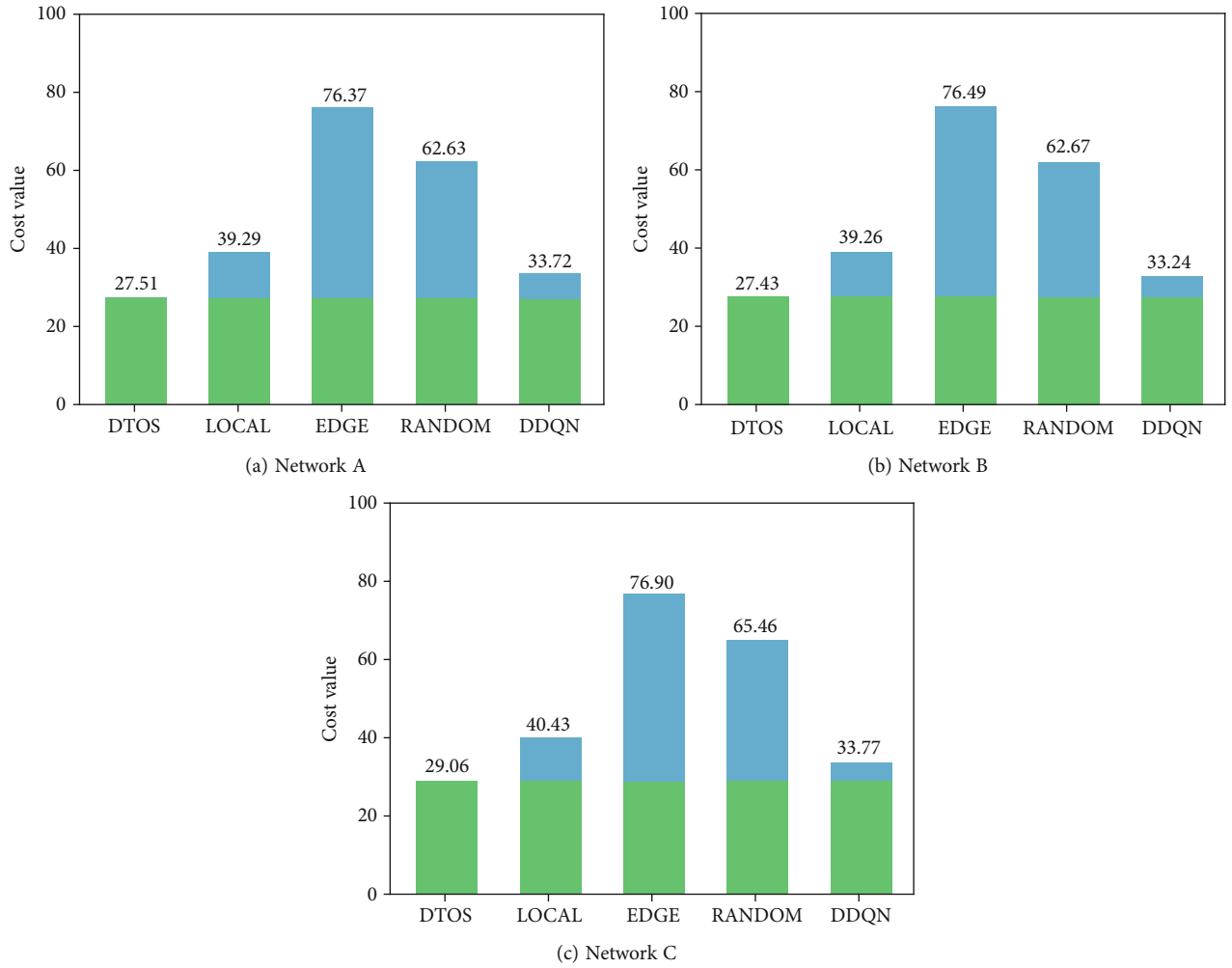


Figure 7: Comparison of cost values for each algorithm with different dependencies.

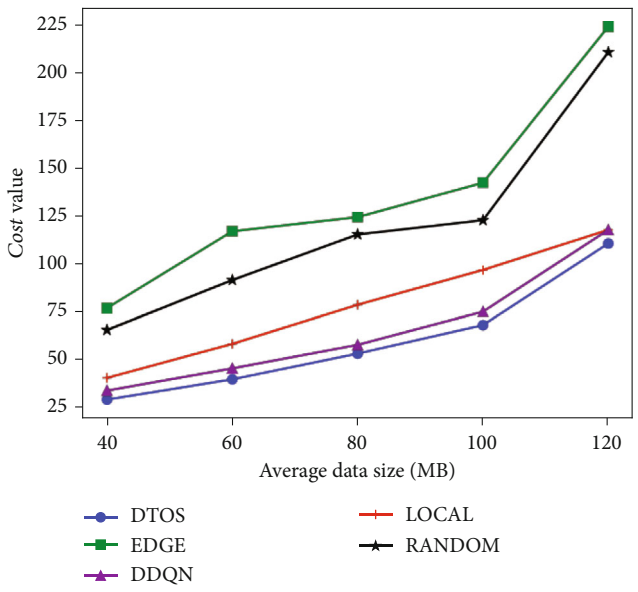


Figure 8: Cost values correspond to different average task data sizes.

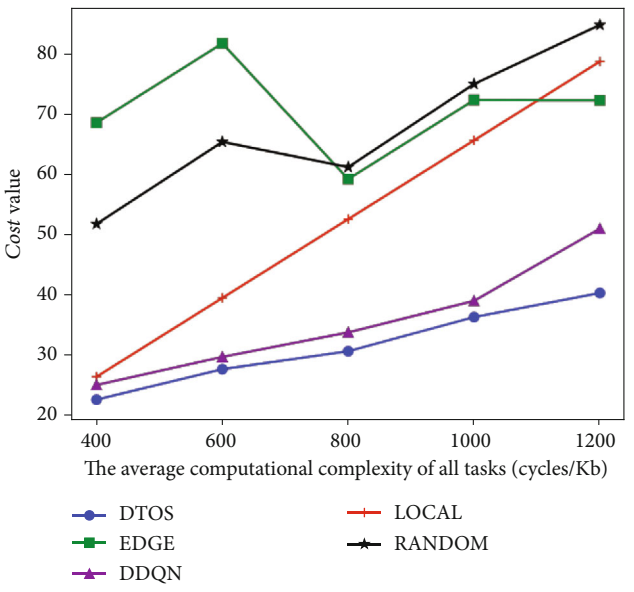


Figure 9: Cost values correspond to different task computational complexities.

Table 2: Comparison of cost values at different network sizes.

Algorithms	$K = 10$	$K = 20$	$K = 30$
DTOS	29.06	37.96	57.93
LOCAL	40.43	65.92	88.99
EDGE	76.90	108.15	143.56
RADOM	65.46	95.31	119.06
DDQN	33.77	52.06	65.77

server's computational resources are much larger than the local computation resources, the computation delay is not the main factor affecting the cost values of these two algorithms. Overall, the cost value of DTOS is lower than the other four algorithms, indicating that DTOS has the best performance for different task computational complexity.

4.3.5. Cost Values of Each Algorithm at Different Network Sizes. When the number K of IoT devices in the network is 10, 20, or 30, the variation of the cost value for each algorithm is shown in Table 2. DTOS has the lowest cost value at all network sizes, indicating that DTOS can adapt to different network sizes and perform well. DTOS outperforms DDQN by 13.9%, 27.1%, and 11.9% when the number K of IoT devices is 10, 20, and 30, respectively.

5. Conclusions

In this paper, we adopt deep reinforcement learning to research dependent-task adaptive-offloading issues in dynamic network environments in edge computing. A deep reinforcement learning-based dependent task-offloading strategy is proposed to transform dependent task offloading into an optimal policy problem under the Markov decision process, and the optimal task-offloading decision is obtained by priority experience replay mechanism and penalty mechanism. The experimental results show that DTOS outperforms the local computing-only algorithm, edge computing-only algorithm, random selection algorithm, and DDQN algorithm all time in different task data size, different task computation complexity, and different network size.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62172255).

References

[1] S. Dong, H. Li, and Y. Qu, "Survey of research on computation unloading strategy in mobile edge computing," *Computer Science*, vol. 46, no. 11, pp. 32–40, 2019.

[2] H. Hu, F. Jin, and S. Lang, "Survey of research on computation offloading technology in mobile edge computing environment," *Computer Engineering and Applications*, vol. 57, no. 14, pp. 60–74, 2021.

[3] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: a survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2017.

[4] A. Islam, A. Debnath, M. Ghose, and S. Chakraborty, "A survey on task offloading in multi-access edge computing," *Journal of Systems Architecture*, vol. 118, no. 4, article 102225, 2021.

[5] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, "Learning-based computation offloading for IoT devices with energy harvesting," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 2, pp. 1930–1941, 2019.

[6] Y. Fang, H. Mei, J. Zhou, and X. Zhang, "Multi-user dynamic energy-saving resource competition computing offloading strategy," *Application Research of Computers*, vol. 39, no. 5, pp. 1524–1528, 2022.

[7] J. A. Suradkar and R. D. Bharati, "Computation offloading: overview, frameworks and challenges," *International Journal of Computer Applications*, vol. 134, no. 6, pp. 28–31, 2016.

[8] K. Fu and J. Ye, "Computation offloading based on improved glowworm swarm optimization algorithm in mobile edge computing," *Journal of Physics: Conference Series*, vol. 1757, no. 1, article 012195, 2021.

[9] A. Zhu and Y. Wen, "Computing offloading strategy using improved genetic algorithm in mobile edge computing system," *Journal of Grid Computing*, vol. 19, no. 3, pp. 1–12, 2021.

[10] F. Wei, S. Chen, and W. Zou, "A greedy algorithm for task offloading in mobile edge computing system," *China Communications*, vol. 15, no. 11, pp. 149–157, 2018.

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

[12] H. Guo and J. Liu, "Collaborative computation offloading for multiaccess edge computing over fiber-wireless networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 5, pp. 4514–4526, 2018.

[13] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: a machine learning-based perspective," *Computer Networks*, vol. 182, no. 9, article 107496, 2020.

[14] L. Huang, X. Feng, A. Feng, Y. Huang, and L. P. Qian, "Distributed deep learning-based offloading for mobile edge computing networks," *Mobile Networks and Applications*, vol. 27, pp. 1123–1130, 2022.

[15] Y. Li, F. Qi, Z. Wang, X. Yu, and S. Shao, "Distributed edge computing offloading algorithm based on deep reinforcement learning," *IEEE Access*, vol. 8, pp. 85204–85215, 2020.

[16] H. Zhou, K. Jiang, X. Liu, X. Li, and V. C. M. Leung, "Deep reinforcement learning for energy-efficient computation offloading in mobile-edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 2, pp. 1517–1530, 2022.

[17] H. Zhang, M. Huang, H. Zhou, X. Wang, N. Wang, and K. Long, "Capacity maximization in RIS-UAV networks: a DDQN-based trajectory and phase shift optimization approach," *IEEE Transactions on Wireless Communications*, vol. 99, pp. 1–1, 2022.

[18] H. Zhou, T. Wu, H. Zhang, and J. Wu, "Incentive-driven deep reinforcement learning for content caching and D2D offloading," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 8, pp. 2445–2460, 2021.

[19] H. Zhou, X. Chen, S. He, J. Chen, and J. Wu, "DRAIM: a novel delay-constraint and reverse auction-based incentive mechanism for WiFi offloading," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 4, pp. 711–722, 2020.

[20] T. Yang and J. Yang, "Deep reinforcement learning method of offloading decision and resource allocation in MEC," *Computer Engineering*, vol. 47, no. 8, pp. 37–44, 2021.

[21] X. Zhu, M. Wu, and G. Wang, "Dynamic resource allocation strategy based on K-means," *Computer Engineering and Design*, vol. 42, no. 4, pp. 901–907, 2021.

[22] D. O. Hao, Z. H. Haiping, L. I. Zhongjin, and H. Liu, "Computation offloading for service workflow in mobile edge computing," *Computer Engineering and Applications*, vol. 55, no. 2, pp. 36–43, 2019.

[23] C. U. Yu-ya, Z. H. De-gan, Z. H. Ting, Y. A. Peng, and Z. H. Hao-li, "A multi-user fine-grained task offloading scheduling approach of mobile edge computing," *Acta Electronica Sinica*, vol. 49, no. 11, pp. 2202–2207, 2021.

[24] Y. Mao, T. Zhou, and P. Liu, "Multi-user task offloading based on delayed acceptance," *Computer Science*, vol. 48, no. 1, pp. 49–57, 2021.

[25] F. Liu, Z. Huang, and L. Wang, "Energy-efficient collaborative task computation offloading in cloud-assisted edge computing for IoT sensors," *Sensors*, vol. 19, no. 5, pp. 1105–1105, 2019.