Intelligent and Converged Networks ISSN 2708-6240
2023, 4(1): 76−92 DOI: 10.23919/ICN.2023.0007

Energy-efficient multiuser and multitask computation offloading optimization method

Meini Pan, Zhihua Li, and Junhao Qian*

Abstract: For dynamic application scenarios of Mobile Edge Computing (MEC), an Energy-efficient Multiuser and Multitask Computation Offloading (EMMCO) optimization method is proposed. Under the consideration of multiuser and multitask computation offloading, first, the EMMCO method takes into account the existence of dependencies among different tasks within an implementation, abstracts these dependencies as a Directed Acyclic Graph (DAG), and models the computation offloading problem as a Markov decision process. Subsequently, the task embedding sequence in the DAG is fed to the RNN encoder-decoder neural network with combination of the attention mechanism, the long-term dependencies among different tasks are successfully captured by this scheme. Finally, the Improved Policy Loss Clip-based PPO2 (IPLC-PPO2) algorithm is developed, and the RNN encoder-decoder neural network is trained by the developed algorithm. The loss function in the IPLC-PPO2 algorithm is utilized as a preference for the training process, and the neural network parameters are continuously updated to select the optimal offloading scheduling decisions. Simulation results demonstrate that the proposed EMMCO method can achieve lower latency, reduce energy consumption, and obtain a significant improvement in the Quality of Service (QoS) than the compared algorithms under different situations of mobile edge network.

Key words: Mobile Edge Computing (MEC); computation offloading; Reinforcement Learning (RL); optimization model

1 Introduction

Numerous mobile implementations, such as virtual reality, the Internet of vehicles, and smartphone online gaming, have evolved with the rapid development of the mobile Internet and the popularity of smart devices. However, these implementations are frequently computation-intensive and latency-sensitive. Most mobile terminals are unable to provide the desired computing services for the aforementioned landscape due to their limited computing capabilities and short battery lives. By offloading tasks with a large amount of data to remote cloud servers for execution, Mobile Cloud Computing (MCC) can effectively decrease the computational workload on mobile terminals and prolong their battery lives[1]. Objectively, because of the negative impacts of the dynamic network workload and transmission distance, there inevitably exists a communication latency, which is insufficient to meet the demands of the implementations for a rapid response. Mobile Edge Computing (MEC) is a new computing paradigm that can effectively address these issues[2]. By deploying relatively resource-rich edge servers close to mobile terminals, MEC provides computing services for computation-intensive and latency-sensitive implementations. Theoretically, MEC can efficiently supplement the insufficiency of limited computing capabilities of mobile terminals, while drastically reducing the system latency and energy consumption, thus improving the Quality of Service (QoS). Under the MEC scenario, offloading computation tasks to MEC servers necessitates data transmission via wireless links. When a large number

• Meini Pan and Zhihua Li are with the School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi 214122, China. E-mail: shirley1014@aliyun.com; zhli@jiangnan.edu.cn.
• Junhao Qian is with the School of IoT Engineering, Jiangnan University, Wuxi 214122, China. E-mail: qjhao@jiangnan.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2023-03-31; accepted: 2023-04-20

© All articles included in the journal are copyrighted to the ITU and TUP. This work is available under the CC BY-NC-ND 3.0 IGO license: https://creativecommons.org/licenses/by-nc-nd/3.0/igo/.

of mobile terminals are actively offloading their computation tasks to the MEC servers, wireless links are prone to network congestion, which easily slows down the data transmission ratio and causes a time delay. Throughout the multiuser and multitask computation offloading process, elements, such as edge network overload and link congestion, not only prolong the service response time, but frequently result in resource overload on MEC servers, which further decreases the QoS. Consequently, in practical applications, some tasks should undertake offloading computing, whereas others examine computation locally in response to the stochastic MEC edge network. This is a feasible technology route for achieving higher QoS.

A large number of works[3–29] have examined this issue from several perspectives, including the computation offloading methods based on dynamic offloading and the Stackelberg game, the schemes of multiuser and multitask offloading, the optimization methods based on heuristic algorithms or genetic algorithms, the methods of dependent task offloading, and the computation methods based on Reinforcement Learning (RL). To treat the task offloading issue in the dynamic MEC with limited computational resources and reduce costs, studies[3–5] investigated computation offloading methods from the perspective of dynamic pricing and the Stackelberg game. The simulation results demonstrate that these schemes can improve the resource utilization of the system and always achieve higher system utility. Under the consideration of multiuser and multitask offloading, through abstracting the task allocation and offloading scheduling as an optimization problem, the works[6–15] studied algorithms and methods for computation offloading. The experimental results show that these methods enable to reduce the computational cost and latency for all users. By using optimization methods based on heuristic algorithms or genetic algorithms, the studies[10–12, 16–21] can effectively manage the workloads on edge servers under the tight constraints of low latency and a fast response time. Considering the dependent task offloading, the works[22–24] presented several intelligent computation offloading schemes, which enable to capture the long-term dependency among tasks and select the optimal offloading manner for tasks. By combining deep neural networks and reinforcement learning algorithms, studies[25–29] proposed efficient computation offloading schemes adapting to various dynamic scenarios. Experiment results indicate that these methods are able to achieve the optimal solution in latency and energy consumption. However, the above studies do not consider either the dependencies among different computation tasks[3–21, 25–29] or various implementation scenarios during the offloading process[10–12, 16–21], which do not suitably meet practical applications.

This paper, by integrating deep neural networks and reinforcement learning, proposes an optimization model for adaptive computation offloading problems in stochastic MEC networks, and presents an Energy-efficient Multiuser and Multitask Computation Offloading (EMMCO) optimization method. Our main contributions are as follows:

(1) First, by mining the objective dependencies among different computation tasks within an implementation, these dependencies are abstracted as a Directed Acyclic Graph (DAG). As a result, the computation offloading problem is modeled as a Markov Decision Process (MDP) by creating a reasonable state space, action space, and reward function.

(2) Furthermore, the MDP is converted into an RNN encoder-decoder neural network prediction procedure. This implies that the task embedding sequence from the DAG is used as the input of the RNN encoder-decoder neural network. In combination with the attention mechanism, the RNN encoder-decoder neural network effectively obtains the long-term dependencies among multiple tasks. This scheme facilitates obtaining more appropriate multitask offloading scheduling decisions.

(3) Finally, an Improved Policy Loss Clip-based PPO2 (IPLC-PPO2) algorithm is developed by improving the PPO2 reinforcement learning algorithm. The presented algorithm is employed to train the RNN

encoder-decoder neural network. The loss function in the IPLC-PPO2 algorithm acts as a preference for the training process, which continuously updates the parameters in the neural network. Consequently, the above significantly determines the final optimal multitask offloading scheduling decisions.

The rest of this paper is organized as follows. In Section 2, we review some related work. In Section 3, we describe the problem model and problem formulation. In Section 4, the proposed EMMCO method is described in detail. In Section 5, extensive simulation experiments are presented to evaluate the performance of the EMMCO method. Finally, Section 6 concludes this paper.

2 Related work

This section focuses on the latest studies on mobile edge computation offloading issues. Reference [13] proposed a decentralized algorithm for balancing computation offloading decisions. This algorithm can minimize the cost by determining whether to offload tasks to the edge servers. Considering the multiuser computation offloading problem in the uncertain wireless network environment, Ref. [14] designed a distributed computation offloading algorithm to obtain the Nash equilibrium of the game. Reference [15] presented a genetics-based intelligent offloading algorithm. This algorithm can lower the overhead of the system during the offloading decision evaluation process. The above studies[13−15] mainly focus on the computation offloading process for independent tasks and do not consider the extensive internal dependencies among tasks. Considering the mobility during the task offloading process, Ref. [16] concentrated on the offloading decision and resource allocation problem among multiple users served by a single base station, it proposed a heuristic mobility-aware offloading algorithm to obtain an approximate optimal offloading scheme. By designing a weight cost model based on latency and energy consumption, Ref. [17] implemented a genetic algorithm based dynamic computation offloading model, which can optimize both latency and energy consumption. Reference [18] investigated the problem of joint computation offloading and transmission scheduling in mobile edge computing. In order to characterize the dynamic management of the system that has potential network uncertainty, it created a queuing model. Besides, it also designed an MOTM method for jointly deciding on computation offloading schemes, transmission scheduling rules, and pricing rules. In two-user MEC networks, Ref. [19] examined the effects of inter-user task dependencies on resource allocation and computation offloading decisions. it then proposed an efficient algorithm to optimize resource allocation and computation offloading decisions. This algorithm can reduce the weighted sum of device energy consumption and task execution time. Considering the heterogeneous communication modes and computing capabilities of network computing points, Ref. [20] proposed a distributed multi-hop computing task offloading framework based on an improved genetic algorithm to reduce the task computation delay and improve the global resource utilization. By decomposing the computation offloading problem into a multitask problem, Ref. [21] presented a DTOS-LBBD method. This method addressed the dynamic computation offloading problem and overcame the difficulties of computation offloading and resource allocation. The above studies[16−21] focus on using heuristic algorithms to resolve the computation offloading problem, but they are mainly conducted to address static optimization problems. However, the MEC system's computation offloading problem is essentially a mixed-integer nonlinear programming problem, it is critical to identify computation offloading strategies that may be applied to dynamic scenarios.

Considering dependent task offloading in mobile edge works, Ref. [22] proposed an intelligent Computational Offloading scheme for Dependent IoT Application (CODIA), which has good performance in convergence, latency, and energy consumption. Reference [23] designed a dependent task offloading framework for multiple mobile applications, named COFE, which assigned the offloaded tasks to the MEC and cloud adaptively to improve the user experience. Reference [24] presented an intelligent task offloading

scheme leveraging off-policy reinforcement learning to achieve the optimal solution in latency and energy consumption under various scenarios. Due to their flexibility in adapting to numerous dynamic scenarios, deep reinforcement learning algorithms have recently gained popularity in mobile edge computation offloading. Reference [25] designed a computation offloading method based on meta-reinforcement learning. This method is able to adapt to various MEC environments within a small number of samples and gradient updates. Considering a binary offloading strategy, Ref. [26] presented a Deep Reinforcement learning based Online Offloading (DROO) algorithm. The DROO algorithm can significantly lower the computational complexity by training the deep neural network and learning from experience. Reference [27] implemented a reinforcement learning based offloading scheme for IoT devices. This scheme selects edge devices and offloading rates according to the current battery level, the previous wireless transmission rate, and the predicted collected energy. Without being aware of the MEC system model, the scheme allows IoT devices to optimize offloading decisions. Regarding intelligent computation offloading in ultra-dense networks, Ref. [28] developed a DQN-based online computation offloading algorithm. This algorithm overcomes the high-dimensional problem in the state space and does not require a priori information. Reference [29] proposed an RL-based distribution offloading algorithm, which combines Long Short-Term Memory (LSTM), dueling Deep Q-Network (DQN), and double-DQN techniques. This RL-based algorithm can significantly reduce the long-term cost compared with several existing algorithms. The above studies[25–29] aim to examine the computation offloading optimization problem using deep neural networks and reinforcement learning algorithms. Motivated by them, we investigate models, algorithms, and methods for handling computation offloading problems in the MEC environment by improving the PPO2 reinforcement learning algorithm and combining the improved IPLC-PPO2 algorithm with the RNN encoder-decoder neural network.

3 Problem modeling

3.1 Problem description

The users, edges, and cloud layers are three main layers that constitute the computation offloading framework for the mobile edge network[30]. The user layer is made up of various terminal devices, the edge layer is organized with base stations and MEC servers that are close to the users, and the cloud layer is the remote data center. In MEC edge networks, the implementations with multitasks running on terminal devices include numerous computation-intensive and latency-sensitive computation tasks. These computation tasks are not independent of each other, but have rather significant dependencies, particularly for implementations in multiuser and complicated domains, such as autonomous driving, AR/VR, and cloud gaming. These pose specific challenges for the efficient offloading of computation tasks in edge networks. The analysis reveals that these computation tasks frequently have strong dependencies on one another rather than being completely independent of one another. Therefore, we abstract the dependencies among the computation tasks as a Directed Acyclic Graph (DAG). In the task scheduling process, starting a task depends on the completion of computing all of its predecessor tasks. Based on this, the analysis reveals that there are primarily three types of dependencies among tasks, namely one-to-one, one-to-many, and many-to-many, which are represented in Fig. 1. In the DAGs in Fig. 1, the vertices represent tasks, and the edges represent the dependencies among tasks. Each task embedding point contains three vectors: the vector containing the task index and estimated task cost, the index vector of the task immediately before it, and the index vector of the task immediately after it. All task embedding points in DAG yield a task embedding

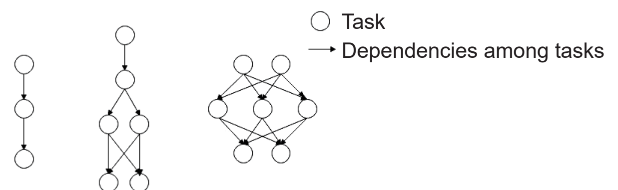

Fig. 1 Structure of directed acyclic graph.

sequence. Additionally, we primarily cover the multiuser and multitask offloading models and methods in this section under two computing paradigms: using local resources to compute the tasks and migrating the tasks to a nearby MEC server for computing.

Assume that the implementation on the terminal device contains n computation tasks, represented as $T=\{t_1,t_2,\ldots,t_i,\ldots,t_n\}$. Let $A_1:n=\{a_1,a_2,\ldots,a_i,\ldots,a_n\}$ denote the scheduling decisions of n computation tasks. $a_i=0$ represents that the task is computed locally, and $a_i=1$ indicates that the task is migrated to the MEC server for computing. $N=\{1,2,\ldots,i,\ldots,n\}$ notes the set of task IDs, d_i means the task's data size, and c_i depicts the CPU clock cycles required to execute the task.

The local computation time primarily depends on the computation time of the task. Let f_l denote the terminal device's CPU clock speed; then, the computation time of the task at the terminal device can be calculated as follows:

$$T_i^l=\frac{c_i}{f_l} \tag{1}$$

When the task is computed locally, the energy consumption is primarily generated by the computation process of the terminal device. The energy consumed by local computing is calculated as follows:

$$E_i^l=\rho f_l^{\varsigma}T_i^l \tag{2}$$

where ρf_l^{ς} indicates the computing power of the terminal device, ρ shows the power coefficient, and ς is a constant.

Three main factors contribute to the time required to migrate the task to the MEC server for execution: uplink transmission time, which is the wireless link transmission time produced by sending the task from the terminal device to the MEC server; MEC server computation time, which is the time required by the edge server to execute the task; and downlink transmission time, which is the wireless link transmission time produced by transmitting the computation result from the MEC server back to the terminal device.

Supposing that the CPU clock speed of the MEC server is f_o, when the task is migrated to the MEC server for computing, the time generated by each of the aforementioned three phases is calculated as follows:

$$T_i^{up}=\frac{d_i}{R^{up}} \tag{3}$$

$$T_i^{com}=\frac{c_i}{f_o} \tag{4}$$

$$T_i^{down}=\frac{d_i^r}{R^{down}} \tag{5}$$

where T_i^{up} means the uplink transmission time of task t_i, T_i^{com} denotes the MEC server computation time of task t_i, T_i^{down} denotes the result return time of task t_i, d_i^r denotes the data size of the received result, R^{up} expresses the uplink transmission rate, and R^{down} represents the downlink transmission rate.

The energy consumption generated by migrating the task to the MEC server for computing is calculated as follows:

$$\begin{aligned}E_i^o=&P^{ws}T_i^{up}+P_jT_i^{com}+P^{wr}T_i^{down}=\\&P^{ws}\frac{d_i}{R^{up}}+P_j\frac{c_i}{f_o}+P^{wr}\frac{d_i^r}{R^{down}}\end{aligned} \tag{6}$$

where P_j denotes the computing power of the MEC server, P^{ws} indicates the wireless sending power, and P^{wr} denotes the wireless receiving power.

3.2 Task completion time and energy consumption

Considering the dependencies among multiple tasks, as depicted in Fig. 1, the start execution time of a task is influenced by the completion time of all its predecessors. Consequently, the time required to execute all tasks in the system and dependencies are closely tied.

3.2.1 Task completion time in local computing

When task t_i is computed locally, the computation start time is the maximum of all predecessor task completion time. Therefore, the completion time of task t_i computed locally is determined as the sum of the computation start time and local computation time, which is calculated as follows:

$$FT_i^l=\max_{t_j\in pre(t_i)}\{\max\{FT_j^l,FT_j^{down}\}\}+T_i^l \tag{7}$$

where $pre(t_i)$ represents the set of predecessors of task t_i, FT_j^l denotes the completion time of the predecessor

task t_j computed locally, and FT_j^{down} indicates the completion time of the predecessor task t_j in offloading computing, whose calculation process is given in Section 3.2.2.

3.2.2 Task completion time in offloading computing

When task t_i is offloaded to the MEC server for computing, it consists of three phases: upload, computation, and result return. First, before the upload phase of the task to the MEC server, all its predecessors must be uploaded. The upload start time depends on two aspects: if the predecessor task t_j is computed locally, it must wait for the completion of the task t_j local computing; if the predecessor task t_j is offloaded to the MEC server for computing, it must wait for the completion of the task t_j upload. Therefore, the upload completion time of task t_i is the sum of the upload start time and uplink transmission time, which is calculated as follows:

$$FT_i^{up} = \max_{t_j \in pre(t_i)} \{\max\{FT_j^l, FT_j^{up}\}\} + T_i^{up} \tag{8}$$

Subsequently, before the computation phase of the task on the MEC server, all its predecessors must be computed. The computation start time depends on two aspects: the completion of task t_i upload and the computation completion of all predecessors of task t_i. Therefore, the completion time of the computation phase of t_i is the sum of the computation start time and MEC server computation time, which is calculated as follows:

$$FT_i^{com} = \max\{FT_i^{up}, \max_{t_j \in pre(t_i)} FT_j^{com}\} + T_i^{com} \tag{9}$$

Finally, the result return phase of task t_i can only begin after its computation phase has been completed. Therefore, the completion time of the result return phase of t_i is the sum of the result return start time and result return time, which is calculated as follows:

$$FT_i^{down} = FT_i^{com} + T_i^{down} \tag{10}$$

3.2.3 Total time and total energy consumption

In conclusion, the completion time of task t_i computed locally is FT_i^l, and the completion time of task t_i in offloading computing is FT_i^{down}. Consequently, the total completion time of all tasks can be calculated as follows:

$$T_{total} = \max_{i \in N} \{\max\{FT_i^l, FT_i^{down}\}\} \tag{11}$$

The energy consumption of task t_i computed locally is E_i^l, and the energy consumption of task t_i in offloading computing is E_i^o. Therefore, the total energy consumption of all tasks can be calculated as follows:

$$E_{total} = \sum_{i \in N, a_i = 0} E_i^l + \sum_{i \in N, a_i = 1} E_i^o \tag{12}$$

3.3 Problem formulation

The energy consumption and latency can be used to evaluate the effectiveness of a scheduling decision[31]. Hence, QoS is defined as a comprehensive indicator of energy consumption and latency, as shown in the following:

$$Q_{A_{1:n}} = \lambda_t \frac{\sum_{i=1}^n T_i^l - T_{total}}{\sum_{i=1}^n T_i^l} + \lambda_e \frac{\sum_{i=1}^n E_i^l - E_{total}}{\sum_{i=1}^n E_i^l} \tag{13}$$

where λ_t and λ_e denote the weights of the impact of the latency and energy consumption on QoS, respectively, $\lambda_t, \lambda_e \in [0,1]$, and $\lambda_t + \lambda_e = 1$. Usually, the scheduling decision is preferable if the QoS value is higher.

Based on the above, the optimization problem of maximizing the system's QoS can be expressed as follows:

$$\arg\ \max\ Q_{A_{1:n}} \tag{14}$$

$$\text{s.t.}, a_i \in \{0,\ 1\},\ \forall i \in N \tag{15}$$

$$\lambda_t, \lambda_e \in [0,1] \tag{16}$$

$$\lambda_t + \lambda_e = 1 \tag{17}$$

Constraint (15) represents the task scheduling decision. The optimization problem of Formula (14) is a mixed integer nonlinear programming problem, which is an NP-hard problem[31]. It is challenging to find an optimal solution for these problems within an acceptable time complexity.

The main symbols used in the paper are summarized in Table 1.

4 Proposed method

In this section, we take the following strategy to resolve the defined optimization problem, namely, Formula (14). First, we transform Formula (14) into a

Table 1 Definition of notations.

Symbol	Definition
a_i	Scheduling decision of task t_i
d_i	Data size of task t_i
d_i^r	Data size of the received result of task t_i
c_i	CPU clock cycles required to execute task t_i
f_l	Local CPU clock speed
f_o	CPU clock speed of MEC server
T_i^l	Local computation time of task t_i
E_i^l	Energy consumption for local computing of task t_i
T_i^{up}	Uplink transmission time of task t_i
T_i^{com}	MEC server computation time of task t_i
T_i^{down}	Result return time of task t_i
E_i^o	Energy consumption of offloading computing for task t_i
R^{up}	Uplink transmission rate
R^{down}	Downlink transmission rate
P_j	Computing power of the MEC server
P^{ws}	Wireless sending power
P^{wr}	Wireless receiving power
$pre(t_i)$	Set of predecessors of task t_i
FT_i^l	Completion time of local computing for task t_i
FT_i^{up}	Upload completion time of task t_i
FT_i^{com}	MEC server computation completion time of task t_i
FT_i^{down}	Completion time of offloading computing for task t_i
T_{total}	Total completion time of all tasks
E_{total}	Total energy consumption of all tasks
$Q_{A_{1:n}}$	QoS of the system

deep reinforcement learning problem and model the offloading problem as an MDP. Then we use a DAG to represent the task embedding sequence with dependencies in the multiuser and multitask implementations, input the sequence to the RNN encoder-decoder neural network, and obtain the appropriate task scheduling decisions. Finally, we develop the IPLC-PPO2 algorithm by improving the PPO2 reinforcement learning algorithm, which is utilized to train the RNN encoder-decoder. By iteratively optimizing the neural network parameters with the loss function in the IPLC-PPO2 algorithm, the optimal task scheduling decisions are finally conducted.

4.1 Prediction process for task scheduling decision

A typical deep reinforcement learning model is the Markov decision model, and the goal of reinforcement learning is to maximize the reward of the MDP[32]. In this study, we define the Markov model represented by the $\langle S, A, R\rangle$ triplet, where S denotes the state space, A expresses the action space, and R represents the reward function.

Because the scheduling decision of task t_i depends on the scheduling decisions of all its predecessors, the state space is defined as a combination of DAG and partial scheduling decisions, as illustrated in the following:

$$S := \{s | s = (G, A_{1:i})\} \tag{18}$$

where G denotes a DAG consisting of a task embedding sequence, and $A_{1:i}$ indicates the set of scheduling decisions from task t_1 to t_i.

Because $a_i = 0$ denotes that the task is computed locally and $a_i = 1$ means that the task is migrated to the MEC server for computing, the action space can be expressed as follows:

$$A := \{0, 1\} \tag{19}$$

Because our optimization objective is to maximize the QoS of the system, as shown in Eq. (13), the reward function is defined as follows:

$$R(s_i, a_i) = \lambda_t \frac{\frac{1}{n}\sum_{i=1}^{n} T_i^l - (FT_i - FT_{i-1})}{\sum_{i=1}^{n} T_i^l} + \lambda_e \frac{\frac{1}{n}\sum_{i=1}^{n} E_i^l - E_i}{\sum_{i=1}^{n} E_i^l} \tag{20}$$

In addition, the computation offloading problem is abstracted as a prediction problem for an RNN encoder-decoder neural network; that is, the task embedding sequence $\boldsymbol{t} = [\boldsymbol{t}_1, \boldsymbol{t}_2, \ldots, \boldsymbol{t}_i, \ldots, \boldsymbol{t}_n]$ in the DAG that corresponds to the task dependencies is input to the RNN encoder-decoder neural network. On the one hand, the encoder combined with the attention mechanism can efficiently mine the dependencies between tasks; on the other hand, the decoder can approximate the policy and value functions of the MDP. The task scheduling decision sequence $A_{1:n}$ is then achieved and output according to the values of the policy function.

As shown in Fig. 2, the RNN encoder-decoder neural

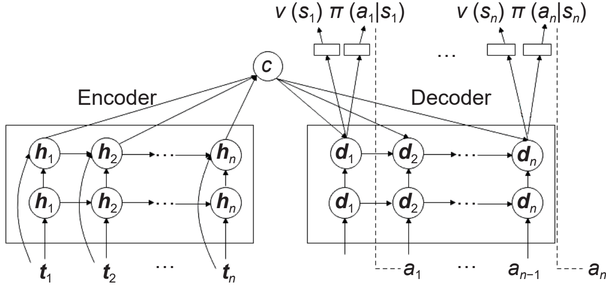

Fig. 2 Architecture of RNN encoder-decoder neural network.

network architecture consists of two RNN networks: one serves as the encoder and the other as the decoder. The encoder receives the input sequence and uses forward propagation to output the final hidden state. The decoder initializes its hidden state using the output state of the encoder, and outputs the target sequence. The following is a detailed description of the task-scheduling decision-making process.

In the encoding process, combined with the attention mechanism, the encoder assigns different attentions to the information encoded at each time step in the input sequence by generating a context vector $\boldsymbol{c}$. Long-term dependencies among tasks can be effectively captured, and information loss in the input sequence can be prevented. The encoding process is as follows: The task embedding sequence $\boldsymbol{t}=[\boldsymbol{t}_1,\boldsymbol{t}_2,\ldots,\boldsymbol{t}_i,\ldots,\boldsymbol{t}_n]$ serves as the encoder's input, a vector $\boldsymbol{t}_i$ is fed into the encoder at each time step, and the hidden layer state $\boldsymbol{h}_i$ is the output. Finally, the weighted average sum of each hidden layer state in the encoder is calculated to obtain the output of the encoder $\boldsymbol{c}$. The context vector $\boldsymbol{c}$ encodes all the information of the input sequence, which is calculated as follows:

$$\boldsymbol{c}=\sum\nolimits_{i=1}^{n}\alpha_{ji}\boldsymbol{h}_i \tag{21}$$

$$\alpha_{ji}=\frac{e^{f_{score}\left(\boldsymbol{d}_{j-1},\boldsymbol{h}_i\right)}}{\sum_{k=1}^{n}e^{f_{score}\left(\boldsymbol{d}_{j-1},\boldsymbol{h}_k\right)}} \tag{22}$$

$$\boldsymbol{h}_i=f_{enc}\left(\boldsymbol{h}_{i-1},\boldsymbol{t}_i|\theta_{enc}\right) \tag{23}$$

where α_{ji} expresses the weight of each hidden layer of the encoder, which is determined by normalization using the SoftMax function, and θ_{enc} denotes the parameter in the encoder.

In the decoding process, the decoder decodes using the obtained context vector $\boldsymbol{c}$. At each time step, the hidden layer node of the decoder has three inputs: the hidden layer state $\boldsymbol{d}_{j-1}$ of the previous time step, output a_{j-1} of the previous time step, and context vector $\boldsymbol{c}$. The decoder hidden layer state is defined as follows:

$$\boldsymbol{d}_j=f_{dec}\left(\boldsymbol{d}_{j-1},a_{j-1},\boldsymbol{c}|\theta_{dec}\right) \tag{24}$$

where θ_{dec} denotes the parameter in the decoder.

Finally, after obtaining the decoder hidden layer state $\boldsymbol{d}_j$, by adding a fully connected layer on $\boldsymbol{d}_j$ and using the SoftMax function, the output is converted into a probability distribution $\pi\left(a_j|s_j\right)$ of the scheduling decision; by adding another fully connected layer on $\boldsymbol{d}_j$, the output is used to represent the state value $v\left(s_j\right)$. The goal is to generate an output sequence with the highest probability; the higher the probability, the more reasonable the output sequence. Consequently, the scheduling decision of a task can be obtained based on $a_j=\arg\ \max_{a_j}\pi(a_j|s_j)$.

4.2 IPLC-PPO2 algorithm

The Proximal Policy Optimization (PPO) algorithm[33] has excellent performance in deep reinforcement learning. The PPO2 method[33], one of the most popular reinforcement learning algorithms, improves the PPO algorithm by merging the policy loss function, value loss function, and policy entropy. By improving the policy loss function, we overcome the shortcomings of the PPO2 algorithm's sluggish learning rate and low learning stability. Furthermore, we develop the IPLC-PPO2 algorithm. The following is a detailed discussion of the improved policy loss function.

First, because of the insufficiencies of the clip function in the PPO2 algorithm described above, the learning rate is sluggish and the learning stability is low when many policy updates are conducted using the same set of data[34]. Thus, limiting the policy probability ratio to the interval $\left[\frac{1}{1+\varepsilon},\frac{1}{1-\varepsilon}\right]$ is expected to increase the change range of the new policy, where ε is a hyperparameter, thus speeding up the algorithm's learning rate; by adding a constant v to the clip function in the PPO2 algorithm, the policy probability ratio beyond the clipping range is relimited to the

interval $\left[\frac{1}{1+\varepsilon}, \frac{1}{1-\varepsilon}\right]$, expecting that the policies will not deviate from the correct optimization, thus improving the algorithm's learning stability.

The improved clip function is shown in the following:

$$clip(r_t(\theta)) = \begin{cases} \frac{1-v}{1-\varepsilon} r_t(\theta) \geqslant \frac{1}{1-\varepsilon}, & v,\varepsilon \in (0,1) \\ \frac{1+v}{1+\varepsilon} r_t(\theta) \leqslant \frac{1}{1+\varepsilon}, & v,\varepsilon \in (0,1) \\ r_t(\theta), \text{ otherwise} \end{cases} \tag{25}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{26}$$

where $r_t(\theta)$ expresses the policy probability ratio, $\pi_\theta(a_t|s_t)$ denotes the probability distribution of the new policy, and $\pi_{\theta_{old}}(a_t|s_t)$ represents the probability distribution of the old policy. ε is used to ensure that when numerous policy updates are performed, the distance between the new policy and old policy is not great. Constant v in the clip function is used to rerestrict $r_t(\theta)$ from outside the clipping range to the clipping range.

The following proof is provided to confirm the validity and accuracy of clip function improvement:

$$\left(\frac{1}{1-\varepsilon} - \frac{1}{1+\varepsilon}\right)\pi_{\theta_{old}}(a_t|s_t) = \frac{2\varepsilon}{1-\varepsilon^2}\pi_{\theta_{old}}(a_t|s_t) \tag{27}$$

and

$$[(1+\varepsilon)-(1-\varepsilon)]\pi_{\theta_{old}}(a_t|s_t) = 2\varepsilon\pi_{\theta_{old}}(a_t|s_t) \tag{28}$$

when $\varepsilon \in (0,1)$, $\frac{2\varepsilon}{1-\varepsilon^2}\pi_{\theta_{old}}(a_t|s_t) > 2\varepsilon\pi_{\theta_{old}}(a_t|s_t)$, therefore,

$$\left(\frac{1}{1-\varepsilon} - \frac{1}{1+\varepsilon}\right)\pi_{\theta_{old}}(a_t|s_t) > [(1+\varepsilon)-(1-\varepsilon)]\pi_{\theta_{old}}(a_t|s_t) \tag{29}$$

By substituting the improved clip function from Eq. (25) into the policy loss function, the improved policy loss function is represented in the following:

$$L^{CLIP}(\theta) = E\left[\sum_{t=1}^{n} \min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta))\hat{A}_t\right)\right] \tag{30}$$

where $\hat{A}_t$ denotes the estimate of the advantage function.

The advantage function is calculated using the generalized advantage estimation method[35], as shown in the following, which has the advantage of greatly lowering the estimated variance of the policy gradient:

$$\hat{A}_t = \sum_{k=0}^{n-t+1} (\gamma\lambda)^k (r_{t+k} + \gamma v_\pi(s_{t+k+1}) - v_\pi(s_{t+k})) \tag{31}$$

where γ denotes the discount factor to reduce the variance, λ $(0 < \lambda < 1)$ expresses the parameter that can make a trade-off between the bias and the variance, and thus estimating the final policy gradient, r_t indicates the reward at time step t, and $v_\pi(s_t)$ denotes the state value at time step t.

The value loss function $L^{VF}(\theta)$ is then estimated using the method in Ref. [33]. $L^{VF}(\theta)$ denotes the squared loss between the sampled state value $v_\pi(s_t)$ and the estimated state value $\hat{v}_\pi(s_t)$, which is calculated as follows:

$$L^{VF}(\theta) = E\left[\sum_{t=1}^{n} (v_\pi(s_t) - \hat{v}_\pi(s_t))^2\right] \tag{32}$$

Finally, to improve the training efficiency, the policy loss function $L^{CLIP}(\theta)$ and value loss function $L^{VF}(\theta)$ are further integrated; that is, Eqs. (30) and (32) are combined, and the entropy function $S[\pi_\theta](s_t)$ is added to ensure efficient scheduling policy exploration. Thus, the objective function shown in the following is obtained:

$$L^{IPLC\text{-}PPO2}(\theta) = E\left[L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right] \tag{33}$$

where c_1 denotes the loss coefficient and c_2 indicates the entropy coefficient.

Based on the above, the IPLC-PPO2 algorithm is proposed, as shown in Algorithm 1.

Compared with the original PPO2 algorithm, the IPLC-PPO2 algorithm can effectively improve the learning stability and accelerate the learning rate to obtain more efficient exploration. Therefore, we train the RNN encoder-decoder network using the IPLC-PPO2 algorithm. The parameter θ of the RNN encoder-decoder neural network is continuously updated using the loss function in the IPLC-PPO2 algorithm. The training objective is to select the optimal scheduling decision for each task in the DAG to optimize the QoS. In terms of time complexity, Lines 5–11 in Algorithm 1, which represent the update phase in the iterative process, account for the majority of the time overhead. Therefore, the time complexity of the IPLC-PPO2 algorithm is $O(l \cdot k)$.

Algorithm 1 IPLC-PPO2 algorithm

1: **Input:** θ_{old} //input the parameter θ_{old} of sampling neural network
2: **Output:** θ //output the parameter θ of updating neural network
3: **Initialize:** Two neural networks with θ_{old} and θ for the old policy $\pi_{\theta_{old}}$, and new policy π_θ with randomly generated initial values;
4: $\theta_{old} \leftarrow \theta$;
5: **for** $episode = 1$ to l **do**
6: Run policy $\pi_{\theta_{old}}$ in environment for T timesteps and store the sampled data on policy $\pi_{\theta_{old}}$ in D_i; // sampling phase
7: **for** $epoch = 1$ to k **do**
8: Optimize the objective function $L^{IPLC\text{-}PPO2}(\theta)$ as shown in Eq. (33) by taking mini-batch stochastic gradient descent using m sampled mini-batches from D_i; // update phase
9: **end for**
10: $\theta_{old} \leftarrow \theta$; // update the neural network parameter
11: **end for**
12: $\theta \leftarrow \theta_{old}$;
13: **return** θ

4.3 EMMCO method

Based on the above research, we further propose the EMMCO method, whose pseudocode is shown in Algorithm 2. First, in the initialization phase, two RNN encoder-decoder neural networks are initialized with the same randomly generated parameters. They are the sampling neural network and updating neural network. Thereafter, in each iteration, the sampling neural network samples a randomly selected set of DAGs from the training dataset D_{train} and stores the sampled data in the set D_i, including the scheduling decision sequence $A_{1:n}$, sampled state value sequence $[v_\pi(s_1), v_\pi(s_2), \ldots, v_\pi(s_n)]$, reward sequence $[r_1, r_2, \ldots, r_n]$ obtained by the environment based on the scheduling decision sequence $A_{1:n}$, advantage estimation sequence $\left[\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n\right]$, and estimated state value sequence $[\hat{v}_\pi(s_1), \hat{v}_\pi(s_2), \ldots, \hat{v}_\pi(s_n)]$. The mini-batch Stochastic Gradient Descent (SGD) method[36] is used to select a portion of samples from D_i randomly and nonrepeatedly to optimize the objective function $L^{IPLC\text{-}PPO2}(\theta)$ to update the neural network parameter θ. Finally, the task scheduling decision sequence $A_{1:n}$ for each DAG in the test dataset is predicted based on the updated network parameter θ, which further yields the total latency, total energy consumption, and total evaluation of QoS for the computation process of all DAGs in the test dataset D_{test}.

Algorithm 2 EMMCO method

1: **Input:** D_{train} and D_{test} //input the training dataset D_{train} and the test dataset D_{test}
2: **Output:** $T, E,$ and Q //output the total latency, total energy consumption, and total evaluation of QoS for the computation process of all DAGs in the test dataset
3: **Initialize:** Two RNN encoder-decoder neural networks with θ_{old} and θ for the old policy $\pi_{\theta_{old}}$, and the new policy π_θ with randomly generated initial values;
4: $\theta_{old} \leftarrow \theta$;
5: **for** $episode = 1$ to l **do**
6: **for** each DAG in D_{train} **do** // sampling phase
7: **for** $sampleepisode = 1$ to T **do**
8: Sample the scheduling decision sequence $A_{1:n}$, sample state value sequence $[v_\pi(s_1), v_\pi(s_2), \ldots, v_\pi(s_n)]$, and reward sequence $[r_1, r_2, \ldots, r_n]$ on policy $\pi_{\theta_{old}}$;
9: Store $A_{1:n}$, $[v_\pi(s_1), v_\pi(s_2), \ldots, v_\pi(s_n)]$ and $[r_1, r_2, \ldots, r_n]$ in D_i;
10: **end for**
11: **end for**
12: **for** $\tau \in D_i$ **do**
13: Compute advantage estimation sequence $[\widehat{A}_1, \widehat{A}_2, \ldots, \widehat{A}_n]$ according to Eq. (31);
14: Compute estimated state values $[\hat{v}_\pi(s_1), \hat{v}_\pi(s_2), \ldots, \hat{v}_\pi(s_n)]$ following the equation $\hat{v}_\pi(s_t) = \sum_{k=0}^{n-t+1} \gamma^k r_{t+k}$;
15: Store $[\widehat{A}_1, \widehat{A}_2, \ldots, \widehat{A}_n]$ and $[\hat{v}_\pi(s_1), \hat{v}_\pi(s_2), \ldots, \hat{v}_\pi(s_n)]$ in D_i;
16: **end for**
17: $\theta_{old} \leftarrow$ IPLC-PPO2(θ_{old}); // call Algorithm 1 to update the neural network parameter
18: **end for**
19: **for** each DAG in D_{test} **do** // calculation phase
20: Compute $A_{1:n}$ following the equation $a_j = \arg\ \max_{a_j} \pi_{\theta_{old}}(a_j|s_j)$;
21: Compute T_{total} according to Eq. (11), compute E_{total} according to Eq. (12), and compute $Q_{A_{1:n}}$ according to Eq. (13);
22: $T \leftarrow T + T_{total}$, $E \leftarrow E + E_{total}$, $Q \leftarrow Q + Q_{A_{1:n}}$;
23: **end for**
24: **return** T, E, Q

The time overhead of the EMMCO method is primarily generated by the update phase of the iterative process, that is, at line 17 of Algorithm 2, when calling Algorithm 1 to update the neural network parameter. The time complexity of the EMMCO method is $O(l \cdot k \cdot m)$.

5 Experimental results and analysis

The experimental environment in this paper employs Python 3.7 and TensorFlow 1.15.0. The operating system of the hardware device is Windows 10, the memory of the hardware device is 32 GB, and the processor of the hardware device is Intel Core i7-9700 CPU@3.00 GHz. The user terminal is not currently considered to offload tasks to the cloud for computing in the MEC network environment, since it is expected that the cloud is far from the user terminals. Suppose there are 100 user terminals, each of which generates a computation-intensive implementation. There are enough MEC servers and computing resources to meet the computing requirements of numerous multitask implementations. The terminal devices' CPU clock speed is set to 1 GHz, whereas the MEC servers' CPU clock speed is set to 10 GHz, which is ten times that of the terminal devices. The wireless sending power is set to 1.258 W and the wireless receiving power is set to 1.181 W.

5.1 Experimental parameters setting

The simulation data consist of a set of DAGs that represent different implementations and are generated using a simulator. The characteristics of the generated DAGs can be modified by changing the simulator's parameters. The number of tasks in each DAG is set to {10, 20, 30, 40, 50, 60, 70, 80, 90,100} and the data size of a single task is defined as [5, 50] KB.

Table 2 displays the settings of the key experimental parameters, which are set after 50 experiments by taking their average values.

The RNN encoder-decoder neural network is implemented via TensorFlow. In particular, both the encoder and decoder are set as two-layer LSTM neural networks with 256 hidden units. Table 3 shows the important parameters used during the training process.

5.2 Effectiveness of EMMCO method

The EMMCO method evolves from the proximal policy optimization algorithm. The average reward, policy entropy, policy loss, and value loss are the four metrics utilized to evaluate the proximal policy optimization algorithm[37]. Here, we also employ these four indicators to validate the effectiveness of the EMMCO method. The experimental results are shown in Fig. 3. It can be seen from Fig. 3a that the average reward shows a sharp growth trend at the beginning and gradually tends to be stable and flat when the number of episodes are greater than 500. In Fig. 3b, the policy entropy gradually decreases during the training process as the policy becomes more and more certain, and the final entropy curve tends to be flat. As for Fig. 3c, the policy loss curve starts to level out after the number of episodes are greater than 400, and the policy loss value gradually tends to be zero. For Fig. 3d, the value loss tends to be zero after the number of episodes are greater than 300. With combination of Figs. 3a−3d, we can find that the entire training process starts to stabilize after episodes are greater than 500, which

Table 2 Experimental parameters.

Parameter	Description	Value
n	Number of tasks	{10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
d_i	Data size of task t_i	[5, 50] KB
d_i^r	Data size of task received result of t_i	[5, 50] KB
c_i	CPU clock cycles required to execute task t_i	$\left[10^7, 10^8\right]$ cycles
f_l	Local CPU clock speed	1 GHz
f_o	CPU clock speed of MEC server	10 GHz
R^{up}	Uplink transmission rate	{3, 5, 7, 9, 11, 13, 15, 17, 19} Mbps
R^{down}	Downlink transmission rate	{3, 5, 7, 9, 11, 13, 15, 17, 19} Mbps
P^{ws}	Wireless sending power	1.258 W
P^{wr}	Wireless receiving power	1.181 W
ρ	Power coefficient in local computing	1.25×10^{-26}
ς	Constant in local computing	3

Table 3 Training parameters of RNN encoder-decoder neural network.

Parameter	Value
Encoder type	LSTM
Decoder type	LSTM
Number of encoder layers	2
Number of decoder layers	2
Number of encoder hidden units	256
Number of decoder hidden units	256
Constant v in clip function	1.1
Constant ε in clip function	0.2
Loss coefficient c_1	0.5
Entropy coefficient c_2	0.01
Learning rate	5×10^{-4}

shows that the EMMCO method is able to avoid updating the policy drastically and prevent the training process from falling in local optimum too early. These experimental results demonstrate that the proposed EMMCO method has good convergence and relative practicality.

Additionally, to analyze the influence of different λ_t and λ_e on the experiment results of EMMCO method, nine groups values of λ_t and λ_e are designed to compare values of the latency, energy consumption, and QoS values generated by EMMCO method. The experimental results are shown in Table 4. It can be seen that the QoS indicators corresponding to nine groups of λ_t and λ_e are decreasing. Therefore, in order to ensure the fairness of the following comparative experiments, we take a compromise parameter value, that is, $\lambda_t = \lambda_e = 0.5$.

5.3 Efficiency of EMMCO method

In this section, to verify its efficiency of the proposed EMMCO method, six typical computation offloading algorithms and an RL-based algorithm are examined to take a comparison and analysis. The seven compared algorithms include all local execution, all offload execution, random scheduling algorithm, round-robin scheduling algorithm, greedy algorithm[38], earliest

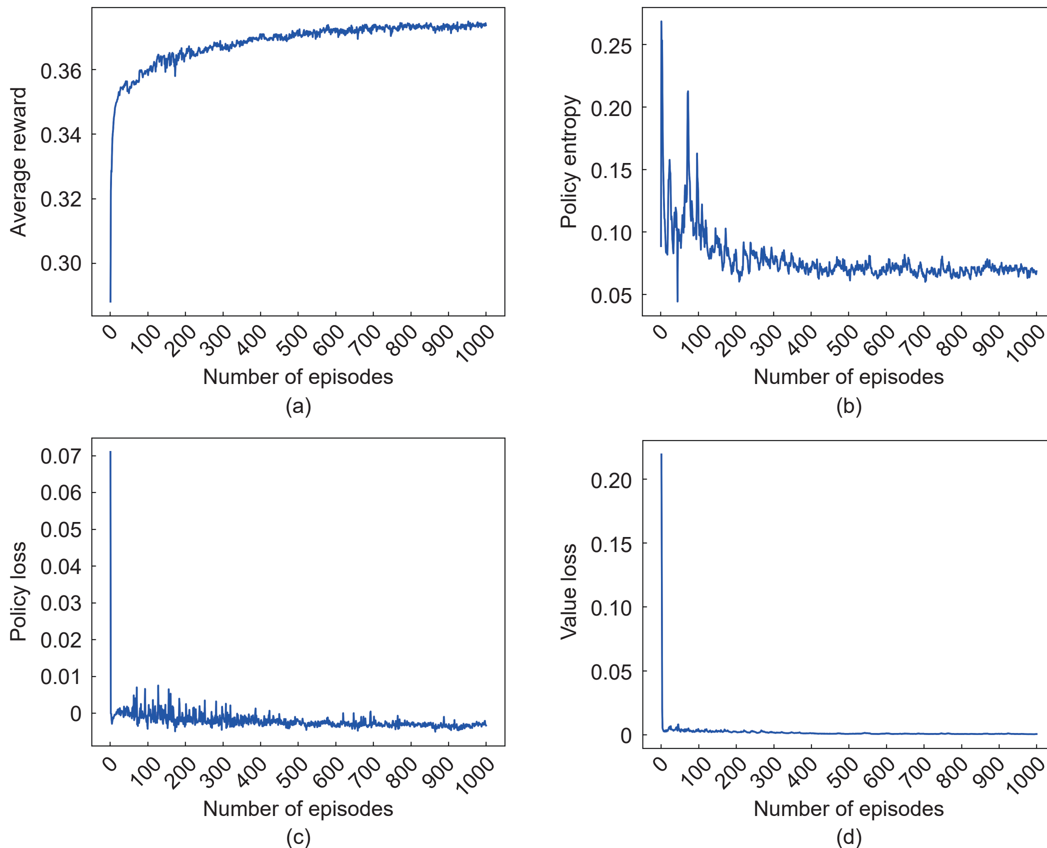

Fig. 3 Effectiveness of the EMMCO method. (a) The average reward of the EMMCO method during the training process; (b) The policy entropy of the EMMCO method during the training process; (c) The policy loss of the EMMCO method during the training process; and (d) The value loss of the EMMCO method during the training process.

Table 4 Experiment results of the EMMCO method under different λ_t and λ_e.

Values of λ_t and λ_e	Latency (ms)	Energy consumption (J)	QoS
$\lambda_t=0.1,\lambda_e=0.9$	541.83	1 092.32	0.71
$\lambda_t=0.2,\lambda_e=0.8$	541.83	1 092.32	0.66
$\lambda_t=0.3,\lambda_e=0.7$	541.83	1 092.32	0.61
$\lambda_t=0.4,\lambda_e=0.6$	541.83	1 092.32	0.55
$\lambda_t=0.5,\lambda_e=0.5$	541.83	1 092.32	0.50
$\lambda_t=0.6,\lambda_e=0.4$	480.40	1 435.73	0.47
$\lambda_t=0.7,\lambda_e=0.3$	472.81	1 543.66	0.43
$\lambda_t=0.8,\lambda_e=0.2$	465.73	1 632.07	0.41
$\lambda_t=0.9,\lambda_e=0.1$	461.55	1 880.28	0.37

completion time algorithm[39], and DDQN-based algorithm[29]. Six typical computation offloading algorithms are set up to indicate the advantages of the reinforcement learning mechanism used in the proposed EMMCO method in solving the model. The DDQN-based algorithm combines LSTM, DQN, and double-DQN techniques to handle the task offloading problem without considering the inner long dependency among tasks. The DDQN-based algorithm is set up to demonstrate the benefit of considering the long-dependency among tasks. Primarily, here the simulation experiments take into account two application cases, one is the workload, i.e., the different numbers of the offloading tasks; the other is the effects of the wireless networks, namely, the different data transmission rates.

5.3.1 Experimental results under different numbers of tasks

First, at the data transmission rate of 7 Mbps, the numbers of tasks are set to {10, 20, 30, 40, 50, 60, 70, 80, 90, 100} to evaluate the latency, energy consumption, and QoS metrics of each algorithm under the scenario of multitasks offloading.

Figures 4 and 5 show the latency and energy consumption yielded by each algorithm under different numbers of tasks, respectively. As can be seen from Figs. 4 and 5, the latency and energy consumption generated by all local execution are always the highest under different task numbers, which is due to the limited computing capabilities of the terminal devices. When the task data size gets large, the terminal device will not be capable of meeting the computing resources required for computing tasks, resulting in significant

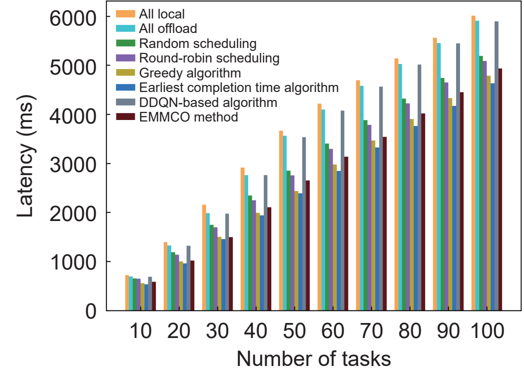

Fig. 4 Comparison of latency of algorithms under different numbers of tasks.

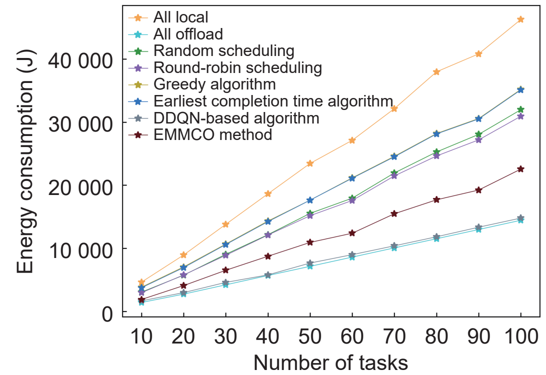

Fig. 5 Comparison of energy consumption of algorithms under different numbers of tasks.

latency and energy consumption. The latency and energy consumption yielded by all offload execution and the DDQN-based algorithm are similar. The values of energy consumption of both are lower, but their latency costs are too high. Therefore, it can be considered that because the DDQN-based algorithm does not take into account the long-dependency among tasks, it cannot achieve a better balance between latency and energy consumption in the learning process. Random scheduling algorithm obtains similar latency and energy consumption results as round-robin scheduling algorithm, although their values of energy consumption are at a moderate level, their values of latency are higher. In contrast, because the greedy algorithm and the earliest completion time algorithm only take into account latency minimization, they both yield lower latency, but their values of energy consumption are higher and second only to all local execution. The latency and energy consumption are well-balanced by the proposed EMMCO method. In

terms of latency, the EMMCO method obtains an acceptable latency close to those of the greedy algorithm and the earliest completion time algorithm; with respect to energy consumption, the EMMCO method gets a desired energy consumption value which is close to those of all offload execution and the DDQN-based algorithm.

Figure 6 shows the QoS of all compared algorithms under different numbers of tasks. Clearly, all local executions have a QoS of zero. This is because, as shown in Eq. (13), all local executions are utilized as the baseline algorithm to evaluate the QoS. Additionally, the value of QoS of the EMMCO method is higher than those of the other compared algorithms under various numbers of tasks, which illustrates that the EMMCO method achieves a promising QoS, resulting from the goal of maximizing QoS.

5.3.2 Experimental results under different data transmission rates

Next, when the number of tasks is 10, we set the data transmission rates to {3 Mbps, 5 Mbps, 7 Mbps, 9 Mbps, 11 Mbps, 13 Mbps, 15 Mbps, 17 Mbps, 19 Mbps} to evaluate the latency, energy consumption, and QoS metrics of all compared algorithms under different data transmission rates.

Figure 7 shows the latency yielded by the algorithms under different data transmission rates. In Fig. 7, the values of latency produced by all local executions remain constant as the data transmission rate increases, which is because such algorithms do not involve task offloading. In contrast, all the other algorithms have a decreasing trend in latency, this is due to the great reduction in data transmission time. When the data transmission rates are {3 Mbps, 5 Mbps}, the latency yielded by all offload execution is the largest; while when the data transmission rate is greater than 11 Mbps, the latency produced by all offload execution is close to the minimum. This case is because that the latency yielded by all offload execution mainly results from the data transmission time, that is, the algorithm is most impacted by the data transmission rate. Additionally, the random scheduling algorithm obtains similar latency as the round-robin scheduling algorithm, and their values of latency are both greater than those of the greedy algorithm and the earliest completion time algorithm. When the data transmission rates are {11 Mbps, 13 Mbps, 15 Mbps, 17 Mbps, 19 Mbps}, the values of latency generated by the EMMCO method and the DDQN-based algorithm are almost the same, which reaches the optimal value. Based on the above, with the premise of optimizing both latency and energy consumption, the EMMCO method can obtain the lowest latency in most cases, which shows that the EMMCO method has certain feasibility and relative advantages.

Figure 8 shows the energy consumption generated by the compared algorithms under different data transmission rates. The energy consumption produced by all local execution is always the highest and the energy consumption produced by all offload execution is always the lowest. Horizontally, as the data transmission rate increases, the values of energy consumption produced by all algorithms except for all local execution decrease, owing to the reduced data

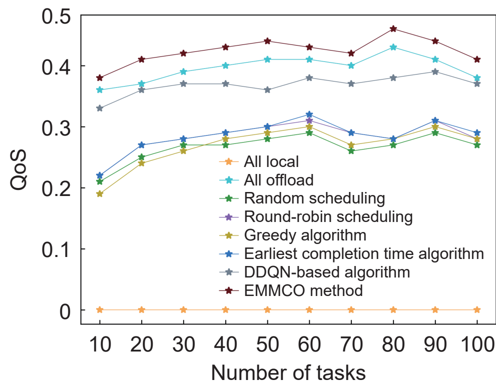

Fig. 6 Comparison of QoS of algorithms under different numbers of tasks.

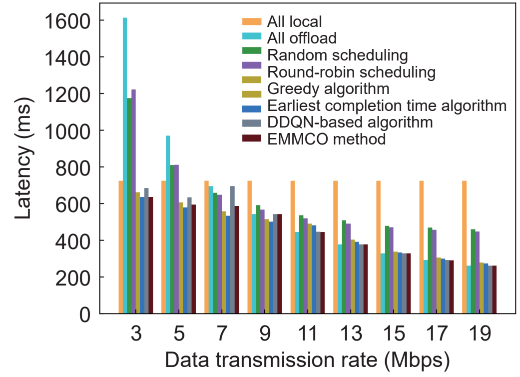

Fig. 7 Comparison of latency of algorithms under different data transmission rates.

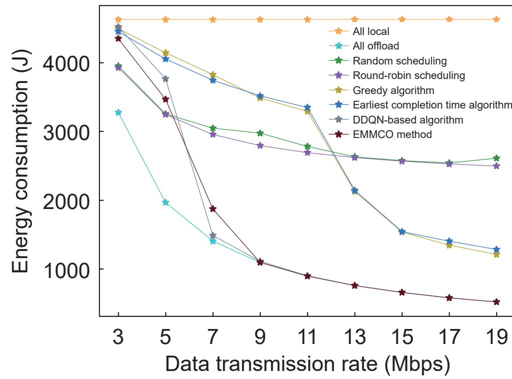

Fig. 8 Comparison of energy consumption of algorithms under different data transmission rates.

transmission time. Besides, the random scheduling algorithm obtains similar energy consumption as the round-robin scheduling algorithm, and the greedy algorithm obtains similar energy consumption as the earliest completion time algorithm, but all four heuristic algorithms have relatively high energy consumption. The EMMCO method has a tolerated energy consumption under most data transmission rates. When the data transmission rate is greater than 9 Mbps, the values of energy consumption of the EMMCO method and the DDQN-based algorithm are near the optimal value.

Figure 9 shows the QoS of all compared algorithms under different data transmission rates. The value of QoS of all local execution is zero under all data transmission rates, while the values of QoS of the other algorithms increase as the data transmission rate rises. This is because the values of data transmission time of other algorithms are shorter, which degrades energy consumption and improves QoS. Evidently, the values

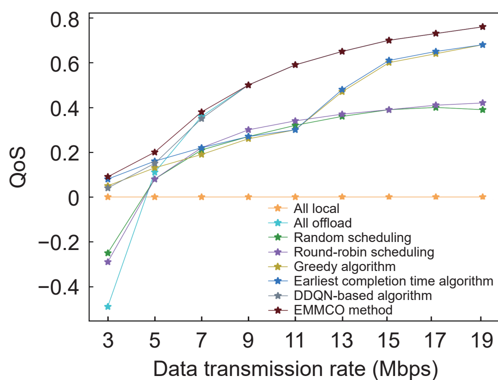

Fig. 9 Comparison of QoS of algorithms under different data transmission rates.

of QoS of the EMMCO method is higher than those of the compared algorithms, which precisely results from the optimization goal of the proposed method. This result proves the relative efficiency of the proposed EMMCO method.

6 Conclusion

The computation offloading of implementations under the stochastic application scenarios in MEC networks is a challenging issue. In this paper, we addressed on computation offloading with an energy-efficient multiuser and multitask computation offloading optimization method. By considering the existence of dependencies between multitasks within a specific implementation, we represented these dependencies with a DAG and modeled the computation offloading problem as an MDP. Then the task embedding sequence in the DAG is input to the RNN encoder-decoder neural network. With combination of the attention mechanism, the RNN network can efficiently capture the long-term dependencies between different tasks and the appropriate task offloading scheduling decisions are determined. To further select the optimal scheduling decisions, the IPLC-PPO2 algorithm is developed. The loss function in the IPLC-PPO2 algorithm is employed as a preference for the training process to constantly update the neural network parameters, as a result, the optimization computation offloading decisions are done. The simulation results show that the proposed EMMCO method obtains lower latency, degrades energy consumption, and gets a higher QoS than that of the compared algorithms. This case proves that the proposed EMMCO method has relative adaptability and higher efficiency.

However, there are still some limitations that require further research. Our presenting scheme assumes that there exist enough resources on the MEC servers. This is not an exact scenario in the actual MEC world. A practical manner needs to be developed. We will further examine the computation offloading methods of joint resource allocation and edge server workloads balancing in mobile edge computing.

Acknowledgment

This work was supported by the Smart Manufacturing

New Model Application Project Ministry of Industry and Information Technology (No. ZH-XZ-18004), the Future Research Projects Funds for the Science and Technology Department of Jiangsu Province (No. BY2013015-23), the Fundamental Research Funds for the Ministry of Education (No. JUSRP211A 41), the Fundamental Research Funds for the Central Universities (No. JUSRP42003), and the 111 Project (No. B2018).

References

[1] Z. Chen, L. Zhang, Y. Pei, C. Jiang, and L. Yin, NOMA-based multi-user mobile edge computation offloading via cooperative multi-agent deep reinforcement learning, *IEEE Trans. Cognit. Commun. Netw.*, vol. 8, no. 1, pp. 350–364, 2022.

[2] S. Nath and J. Wu, Deep reinforcement learning for dynamic computation offloading and resource allocation in cache-assisted mobile edge computing systems, *Intelligent and Converged Networks*, vol. 1, no. 2, pp. 181–198, 2020.

[3] G. Mitsis, E. E. Tsiropoulou, and S. Papavassiliou, Price and risk awareness for data offloading decision-making in edge computing systems, *IEEE Syst. J.*, vol. 16, no. 4, pp. 6546–6557, 2022.

[4] Z. Tong, X. Deng, J. Mei, L. Dai, K. Li, and K. Li, Stackelberg game-based task offloading and pricing with computing capacity constraint in mobile edge computing, *J. Syst. Architect.*, vol. 137, p. 102847, 2023.

[5] K. Zhang, X. Gui, D. Ren, T. Du, and X. He, Optimal pricing-based computation offloading and resource allocation for blockchain-enabled beyond 5G networks, *Comput. Netw.*, vol. 203, p. 108674, 2022.

[6] G. Zhang, S. Zhang, W. Zhang, Z. Shen, and L. Wang, Joint service caching, computation offloading and resource allocation in mobile edge computing systems, *IEEE Trans. Wirel. Commun.*, vol. 20, no. 8, pp. 5288–5300, 2021.

[7] Y. Chen, X. Zhou, W. Wang, H. Wang, Z. Zhang, and Z. Zhang, Delay-optimal closed-form scheduling for multi-destination computation offloading, *IEEE Wirel. Commun. Lett.*, vol. 10, no. 9, pp. 1904–1908, 2021.

[8] C. Wang, F. R. Yu, C. Liang, Q. Chen, and L. Tang, Joint computation offloading and interference management in wireless cellular networks with mobile edge computing, *IEEE Trans. Veh. Technol.*, vol. 66, no. 8, pp. 7432–7445, 2017.

[9] E. F. Maleki and L. Mashayekhy, Mobility-aware computation offloading in edge computing using prediction, in *Proc. of the 2020 IEEE 4th Int. Conf. on Fog and Edge Computing* (*ICFEC*), Melbourne, Australia, 2020, pp. 69–74.

[10] H. Tout, A. Mourad, N. Kara, and C. Talhi, Multi-persona mobility: Joint cost-effective and resource-aware mobile-edge computation offloading, *IEEE/ACM Trans. Netw.*, vol. 29, no. 3, pp. 1408–1421, 2021.

[11] R. Ezhilarasie, M. S. Reddy, and A. Umamakeswari, A new hybrid adaptive GA-PSO computation offloading algorithm for IoT and CPS context application, *J. Intell. Fuzzy Syst.*, vol. 36, no. 5, pp. 4105–4113, 2019.

[12] M. Babar, M. S. Khan, A. Din, F. Ali, U. Habib, and K. S. Kwak, Intelligent computation offloading for IoT applications in scalable edge computing using artificial bee colony optimization, *Complexity*, vol. 2021, p. 5563531, 2021.

[13] S. Jošilo and G. Dán, Computation offloading scheduling for periodic tasks in mobile edge computing, *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 667–680, 2020.

[14] L. Tang and S. He, Multi-user computation offloading in mobile edge computing: A behavioral perspective, *IEEE Netw.*, vol. 32, no. 1, pp. 48–53, 2018.

[15] M. Merluzzi, N. di Pietro, P. Di Lorenzo, E. C. Strinati, and S. Barbarossa, Discontinuous computation offloading for energy-efficient mobile edge computing, *IEEE Trans. Green Commun. Netw.*, vol. 6, no. 2, pp. 1242–1257, 2022.

[16] W. Zhan, C. Luo, G. Min, C. Wang, Q. Zhu, and H. Duan, Mobility-aware multi-user offloading optimization for mobile edge computing, *IEEE Trans. Veh. Technol.*, vol. 69, no. 3, pp. 3341–3356, 2020.

[17] S. Bi, L. Huang, and Y. J. A. Zhang, Joint optimization of service caching placement and computation offloading in mobile edge computing systems, *IEEE Trans. Wirel. Commun.*, vol. 19, no. 7, pp. 4947–4963, 2020.

[18] C. Yi, J. Cai, and Z. Su, A multi-user mobile computation offloading and transmission scheduling mechanism for delay-sensitive applications, *IEEE Trans. Mob. Comput.*, vol. 19, no. 1, pp. 29–43, 2020.

[19] J. Yan, S. Bi, Y. Zhang, and M. Tao, Optimal task offloading and resource allocation in mobile-edge computing with inter-user task dependency, *IEEE Trans. Wirel. Commun.*, vol. 19, no. 1, pp. 235–250, 2020.

[20] H. Liu, Z. Niu, J. Du, and X. Lin, Genetic algorithm for delay efficient computation offloading in dispersed computing, *Ad Hoc Netw.*, vol. 142, p. 103109, 2023.

[21] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, Dynamic task offloading and scheduling for low-latency IoT services in multi-access edge computing, *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 668–682, 2019.

[22] H. Xiao, C. Xu, Y. Ma, S. Yang, L. Zhong, and G. M. Muntean, Edge intelligence: A computational task offloading scheme for dependent IoT application, *IEEE Trans. Wirel. Commun.*, vol. 21, no. 9, pp. 7222–7237,

2022.

[23] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, Efficient dependent task offloading for multiple applications in MEC-cloud system, *IEEE Trans. Mob. Comput.*, vol. 22, no. 4, pp. 2147–2162, 2023.

[24] J. Wang, J. Hu, G. Min, W. Zhan, A. Y. Zomaya, and N. Georgalas, Dependent task offloading for edge computing based on deep reinforcement learning, *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2449–2461, 2022.

[25] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, Fast adaptive task offloading in edge computing based on meta reinforcement learning, *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 242–253, 2021.

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

[27] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, Learning-based computation offloading for IoT devices with energy harvesting, *IEEE Trans. Veh. Technol.*, vol. 68, no. 2, pp. 1930–1941, 2019.

[28] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, Performance optimization in mobile-edge computing via deep reinforcement learning, in *Proc. of the 2018 IEEE 88th Vehicular Technology Conf.* (*VTC-Fall*), Chicago, IL, USA, 2019, pp. 1–6.

[29] M. Tang and V. W. S. Wong, Deep reinforcement learning for task offloading in mobile edge computing systems, *IEEE Trans. Mob. Comput.*, vol. 21, no. 6, pp. 1985–1997, 2022.

[30] B. M. Amine, F. Farha, and H. Ning, Convergence of computing, communication, and caching in Internet of Things, *Intell. Conver. Netw.*, vol. 1, no. 1, pp. 18–36, 2020.

[31] S. Chu, Z. Fang, S. Song, Z. Zhang, C. Gao, and C. Xu, Efficient multi-channel computation offloading for mobile edge computing: A game-theoretic approach, *IEEE Trans. Cloud Comput.*, vol. 10, no. 3, pp. 1738–1750, 2022.

[32] Y. Liao, L. Shou, Q. Yu, Q. Ai, and Q. Liu, Joint offloading decision and resource allocation for mobile edge computing enabled networks, *Comput. Commun.*, vol. 154, pp. 361–369, 2020.

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, Proximal policy optimization algorithms, arXiv preprint arXiv: 1707.06347, 2017.

[34] Z. Zhang, X. Luo, T. Liu, S. Xie, J. Wang, W. Wang, Y. Li, and Y. Peng, Proximal policy optimization with mixed distributed training, in *Proc. of the 2019 IEEE 31st Int. Conf. on Tools with Artificial Intelligence* (*ICTAI*), Portland, OR, USA, 2019, pp. 1452–1456.

[35] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, High-dimensional continuous control using generalized advantage estimation, arXiv preprint arXiv: 1506.02438, 2018.

[36] I. Loshchilov and F. Hutter, SGDR: Stochastic gradient descent with warm restarts, arXiv preprint arXiv: 1608.03983, 2017.

[37] D. Zhao, D. Liu, F. L. Lewis, J. C. Principe, and S. Squartini, Special issue on deep reinforcement learning and adaptive dynamic programming, *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 6, pp. 2038–2041, 2018.

[38] S. Suzuki, M. Fujiwara, Y. Makino, and H. Shinoda, Radiation pressure field reconstruction for ultrasound midair haptics by greedy algorithm with brute-force search, *IEEE Trans. Haptics*, vol. 14, no. 4, pp. 914–921, 2021.

[39] G. Patel, R. Mehta, and U. Bhoi, Enhanced load balanced min-min algorithm for static meta task scheduling in cloud computing, *Procedia Comput. Sci.*, vol. 57, pp. 545–553, 2015.

Meini Pan received the BEng degree from Jiangnan University, China in 2020. She is currently a master student at the School of Artificial Intelligence and Computer Science, Jiangnan University. Her research interests include mobile edge computing and computation offloading optimization.

Zhihua Li recieved the PhD degree from Jiangnan University, China in 2009. He is now a professor at the School of Artificial Intelligence and Computer Science, Jiangnan University. His research interests include network technology, parallel/distributed computing, information security, edge computing, and mobile computing.

Junhao Qian received the MEng degree from Jiangnan University, China in 1998. He is now a full professor at the School of IoT Engineering, Jiangnan University. His research interests include edge computing, cloud computing, and agricultural IoT engineering technology.