RESEARCH-ARTICLE

Cost-Optimized Periodic DAG-Structured Task Offloading in Multi-User MEC Systems Using Reinforcement Learning

YAN WANG, Guangzhou University, Guangzhou, Guangdong, China

YUBIN HE, Guangzhou University, Guangzhou, Guangdong, China

GANG LIU, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

KEQIN LI, SUNY New Paltz, New Paltz, NY, United States

Published: 14 January 2026
Online AM: 26 August 2025
Accepted: 07 August 2025
Revised: 03 July 2025
Received: 02 November 2024

Cost-Optimized Periodic DAG-Structured Task Offloading in Multi-User MEC Systems Using Reinforcement Learning

YAN WANG, The school of Computer science, Guangzhou University, Guangzhou, China
YUBIN HE, Guangzhou University, Guangzhou, China
GANG LIU, Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen, China
KEQIN LI, State University of New York, New Paltz, United States

Reinforcement Learning (RL) has emerged as a promising solution for task offloading due to its adaptability to dynamic environments and ability to reduce online computational overhead. Thereby, this article explores RL for optimizing periodic Directed Acyclic Graph (DAG) task offloading in multi-user Mobile Edge Computing (MEC) systems, aiming to minimize overall costs, including user device energy consumption and server computational charges. A key contribution of this work is the explicit modeling of user competition for limited edge resources, where concurrent access leads to dynamic contention, significantly affecting offloading latency and energy usage. However, this optimization task faces two main challenges: the high dimensionality of task states and the large action space, both of which increase learning complexity. To address this, we propose a dynamic and distributed Proximal Policy Optimization (PPO)-based offloading framework. An encoder is employed to map DAG node features and structural information into a lower-dimensional representation, reducing computational overhead and improving learning efficiency. Additionally, we incorporate behavioral cloning to imitate greedy policies as the PPO agent's initial behavior, effectively narrowing the action space and accelerating convergence. By combining representation learning and imitation-based initialization, our method enables the PPO agent to quickly adapt to environmental dynamics, leveraging both prior knowledge and real-time feedback to make informed offloading decisions. Simulation results confirm that our approach achieves rapid convergence and outperforms existing baselines in cost reduction, demonstrating its effectiveness for periodic task offloading in MEC scenarios. The source code and implementation details are available at: https://github.com/xiaolutihua/GAT/tree/master.

CCS Concepts: • **Computing methodologies → Distributed computing methodologies;**

Additional Key Words and Phrases: Cost efficient, MEC, periodic DAG-based application, task offloading optimization

Keqin Li Fellow, IEEE.
Authors' Contact Information: Yan Wang, The school of Computer science, Guangzhou University, Guangzhou, China; e-mail: bessie11@yeah.net; Yubin He, Guangzhou University, Guangzhou, China; e-mail: yubinghe111@163.com; Gang Liu (corresponding author), Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen, China; e-mail: liug@hnu.edu.cn; Keqin Li, State University of New York, New Paltz, United States; e-mail: lik@newpaltz.edu.

ACM 1533-5399/2026/01-ART2
https://doi.org/10.1145/3762993

ACM Reference Format:
Yan Wang, Yubin He, Gang Liu, and Keqin Li. 2026. Cost-Optimized Periodic DAG-Structured Task Offloading in Multi-User MEC Systems Using Reinforcement Learning. *ACM Trans. Internet Technol.* 26, 1, Article 2 (January 2026), 27 pages. https://doi.org/10.1145/3762993

1 Introduction

The proliferation of **Internet of Things** (**IoT**) technologies has led to an exponential growth in the interconnection of devices to the internet, resulting in the generation of voluminous data. This surge has precipitated a significant escalation in data volume on **user equipment** (**UE**), consequently amplifying the operational demands on network infrastructures and cloud computing centers, and concurrently escalating the associated economic costs. Numerous application scenarios, including autonomous driving, industrial automation, and telemedicine, necessitate extremely low latency and real-time response capabilities. Consequently, there is a critical need for data processing to be performed at the site of data generation to mitigate data transmission and augment processing velocity. However, the limited computational resources and processing capabilities inherent to edge UE may result in inadequate computational power when tasked with managing complex computational tasks. This challenge necessitates the exploration of alternative solutions to enhance computational efficacy at the edge of the network. In response to these challenges, **Mobile Edge Computing** (**MEC**) has emerged as a viable solution, facilitating the decentralization of computational resources and services toward the network periphery [15]. Within this paradigm, users are empowered to delegate their computation-intensive tasks to infrastructures rich in resources, such as MEC servers. Therefore, there has been a burgeoning interest from both the industrial sector and academic circles in the development of optimal strategies for task offloading.

In the pursuit of augmenting task offloading decisions, an array of optimization methodologies is under investigation, encompassing heuristic algorithms [11, 24, 26], machine learning models [28, 29], and game-theoretic approaches [12, 16]. These methodologies are designed to reduce latency, curtail energy expenditure, and diminish computational costs, whilst concurrently striving to augment the overall system throughput. A pivotal facet of this realm pertains to the optimization of periodic **Directed Acyclic Graph** (**DAG**) task offloading, wherein computational tasks are migrated from UE to edge or cloud servers at predetermined intervals. Despite the substantial practical significance of the behavioral and economic modeling of such task offloading paradigms, research endeavors that integrate user behavioral decision-making traits encounter considerable challenges. These challenges arise from the intricate and multifaceted attributes of periodic DAG-structured tasks within the context of heterogeneous multi-user settings. This article endeavors to confront these challenges by examining the interplay among user behavioral characteristics, the intricacies of heterogeneous multi-user environments, and the computational expenses associated with Multi-Access Edge Computing servers. Unlike prior works that assume independent offloading decisions, we consider the competitive dynamics of multiple users sharing constrained edge resources. These interactions introduce substantial variability in task latency and energy consumption, necessitating competition-aware scheduling strategies. By explicitly modeling such dynamics, our approach aims to minimize offloading costs in a manner robust to both user contention and system heterogeneity. This methodology empowers UEs to tailor their offloading strategies in accordance with real-time conditions and the presence of other users within the network environment.

Reinforcement Learning (**RL**) emerges as a promising paradigm for orchestrating distributed, heterogeneous multi-user MEC systems, particularly in addressing the unique computational dependencies inherent in DAG-structured tasks. Its capacity to address multi-user task offloading challenges through collaborative agent optimization enhances the collective synergistic efficacy

of the system. In this article, we study an RL algorithm to address the periodic DAG-structured task offloading problem within multi-user MEC settings. While existing studies have investigated generic task offloading challenges [18], the structural complexity of DAG-structured tasks introduces two distinctive challenges that have not been sufficiently addressed in multi-user MEC environments: (1) DAG-Driven State Space Explosion: Unlike conventional linear task models, the topological ordering and inter-task dependencies in DAG workflows exponentially expand the state representation dimension. Specifically, the state space must encode not only the conventional UE-device parameters but also precedence constraints between vertices and dynamic branch execution probabilities, resulting in a combinatorial state space complexity of $O(N^D)$ where N denotes concurrent UEs and D represents average DAG depth. (2) The Expansive Action Search Space: The parallel execution constraints imposed by DAG edges fundamentally alter the action space characteristics. Each offloading decision must simultaneously satisfy: (a) parent node execution precedence, (b) branch synchronization requirements, and (c) heterogeneous resource contention across multi-user edge servers. This creates a cascading action space where local decisions at one node propagate constraints through the entire DAG graph. With an increasing number of users generating real-time computation tasks, the endeavor to minimize offloading expenditures for the collective user base leads to a substantial prolongation of the episode time required for the optimization of RL algorithms. This augmentation of the environmental action search space presents a formidable challenge for the training of RL algorithms.

Within the scope of this scholarly endeavor, we conduct an iteration of the **Proximal Policy Optimization**, a type of RL algorithm, designated as **PPO**, aimed at optimizing the offloading of periodic DAG-structured tasks within a multi-user Mobile MEC environment. The PPO algorithm, as articulated by Schulman et al. [22], adeptly encapsulates and delineates the intrinsic structural attributes of tasks amidst a heterogeneous server environment. This approach demonstrates equivalence or superiority in performance when juxtaposed with existing state-of-the-art methodologies, while concurrently offering a more streamlined implementation and calibration process. To address the challenges of high dimensionality and expansive search spaces, this article delineates the incorporation of Encoding and Behavioral Cloning techniques. Within the framework of the PPO, the amalgamation of an encoder and behavioral cloning enables the PPO model to efficiently assimilate and accommodate the nuances of the environment. Furthermore, by incorporating the energy expenditure of UEs and the computational costs of servers into the task cost functions, the agent is endowed with the capacity to render judicious decisions regarding task offloading. This holistic approach ensures a balanced optimization of both energy conservation at the UE level and economic efficiency in terms of server utilization.

The main contributions of the article are summarized as follows.

— We introduce a dynamic and distributed RL approach, PPO, which is specifically designed for managing periodic task offloading in multi-user MEC environments. This strategy incorporates behavioral cloning technology and an encoding mechanism to augment computational performance, thereby facilitating intelligent decision-making processes that are adept at minimizing costs while adhering to time and resource constraints.

— To address the issue of high dimensionality in task states, we design a task-specific Encoder that effectively integrates node-level features and DAG structural dependencies into a compact, low-dimensional embedding. This customized representation is tailored to our scheduling environment and enhances the PPO agent's capacity to make informed decisions in complex, high-dimensional state spaces.

— To cope with the enlarged action space caused by the presence of numerous users and tasks, we introduce a behavioral cloning pre-training phase. Instead of merely replicating existing strategies, we construct a lightweight expert policy based on domain-specific heuristics,

which guides the initial learning process of the PPO agent. This significantly accelerates convergence and improves policy quality, particularly in the early stages of training.
— We propose a low-complexity greedy matching algorithm designed to provide an initial task offloading policy for PPO agents. By harnessing behavioral cloning techniques, this algorithm not only diminishes the search space but also bolsters the overall efficiency of the task offloading process, ensuring that the PPO agents can swiftly adapt to the dynamic requirements of the MEC environment.

The rest of the article is organized as follows. Section 3 describes the system model, execution models, cost models, and problem formulation. In Section 4, the cost optimal task offloading algorithm based on PPO learning model for the MEC environments is proposed. We evaluate the performance of our proposed technique and compare it with state-of-the-art techniques in Section 5. Section 6 concludes the article.

2 Related Work

2.1 Periodic DAG-Structured Task Offloading in MEC

Several studies [3, 7, 9, 14, 17, 21, 27, 30] have addressed the periodic task offloading problem and provided insights into efficient strategies and algorithms. Dinh et al. [9] explored task offloading scenarios involving multiple edge servers for a single user. They proposed linear relaxation and **Semidefinite Relaxation** (**SDR**) techniques to allow users to offload independent tasks to different edge servers while satisfying latency constraints. Pham et al. [21] formulated the periodic task offloading problem by formulating it as an integer non-linear programming problem. They introduced a nested genetic algorithm to maximize MEC utilization. However, these methodological approaches, while ostensibly effective for singular user paradigms, encounter inherent limitations and formidable challenges when scaled to accommodate the complexities of multi-user interactions.

Optimizing multi-user periodic task offloading aims to improve distribution efficiency, reduce latency, maximize resource utilization, and enhance overall system performance. Recently, this problem has received significant research attention [4, 6, 13, 18, 23]. In their work, Josilo et al. [13] proposed a task offloading and resource allocation algorithm based on game theory for offloading real-time tasks in heterogeneous communication MEC environments. Shahryari et al. [23] developed a suboptimal algorithm that combines genetic algorithms and particle swarm optimization to address the issue of limited battery capacity and task delay sensitivity in UEs.

2.2 RL for DAG-Structured Task Offloading in MEC

Using RL to solve DAG-structured task offloading in MEC, the state space is defined using both the DAG structure of the application, MEC environment status information, and the corresponding offloading strategy for its tasks. The action space, on the other hand, determines whether a task will be executed locally or offloaded to a MEC server. The current research landscape in periodic task offloading focuses on developing RL strategies to address challenges such as latency, energy consumption, and network bandwidth constraints [2, 5, 8, 10, 14, 19, 30, 31]. Deng et al. [8] presented a DAG-structured task offloading and resource allocation algorithm based on the DDPG model, aiming to reduce task execution delays and enhance service quality in MEC systems. In [10], Goudarzi et al. proposed an actor-critic-based distributed application placement technique utilizing importance-weighted Actor-Learner Architectures. This technique efficiently addresses the application placement problem of Directed DAG IoT applications in heterogeneous MEC environments, where edge and cloud servers collaborate. In their work, [20] introduced an attention-driven **double deep Q network** (**DDQN**) aimed at reducing both task completion delay and energy consumption over the long term.

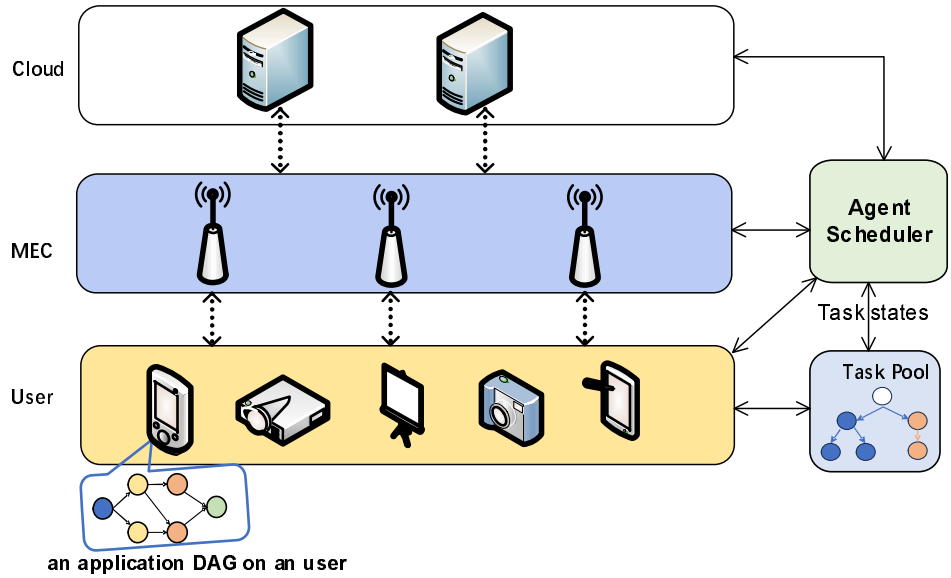


Fig. 1. An overview of our system model.

However, it is imperative to acknowledge a substantial limitation inherent in the aforementioned RL task offloading methodologies. These techniques fail to account for the contention among User Equipment UE users, which can lead to certain tasks being queued on the server, pending the completion of other UE tasks, thereby exacerbating the overall execution latency. Consequently, this article study endeavors to incorporate the rivalry among multiple UEs in its approach to tackling the offloading of periodic DAG-structured tasks within the ambit of multi-user MEC environments. This comprehensive consideration is essential to mitigate the delays induced by concurrent task processing and to enhance the efficacy of task offloading strategies in a shared computing context.

3 Models

In this section, we present the system model, the execution models for both offloaded and non-offloaded tasks, the cost consumption models for both energy consumption on local users and computation fees on servers, and the problem formulation. Table 1 gives a summary of notations and definitions introduced in this article.

3.1 System Models

As shown in Figure 1, we consider a multi-user MEC system consisting of N local UEs, denoted as $\{UE_1, \ldots, UE_N\}$, and M heterogeneous servers, denoted as $\{MEC_1, \ldots, MEC_M\}$. These heterogeneous servers, including both edge servers and remote cloud servers, offer computation offloading services for UEs.

Each user UE_i has a periodic DAG application represented by $G_i = (V_i, E_i)$, consisting of N_i subtasks. The set of subtasks for DAG G_i is defined as $V_i = \{v_{i,1}, \ldots, v_{i,j}, \ldots, v_{i,N_i}\}$, where each node $v_{i,j}$ represents the jth subtask in DAG G_i. Each subtask $v_{i,j} \in V_i$ is defined as $v_{i,j} = (r_{i,j}, d_{i,j}, ram_{i,j})$, where $r_{i,j}$ represents the total instructions (in BI) required for the execution of $v_{i,j}$, $d_{i,j}$ denotes the self-data needed (in MB) during the execution of $v_{i,j}$, and $ram_{i,j}$ indicates the minimum memory required for executing task $v_{i,j}$. The edge set $E_i \subseteq V_i \times V_i$ represents the dependency relationships among periodic subtasks in DAG G_i. An edge $e_i(u, j) \in E_i$ indicates that tasks $v_{i,u}$ and $v_{i,j}$ have a

Table 1. Summary of Notations and Definitions

Notation	Definition
N, UE_i	the number of UEs and the i-th UE
M, MEC_m	the number of servers and the m-th MEC
f_i^{ue}	the CPU-cycle frequency of UE_i
f_m^{mec}	the CPU-cycle frequency of MEC_m
$P(i,j)$	the processor for execution $v_{i,j}$
β_i^{ue}	the number of CPU cycles for a instruction on UE_i
β_m^{mec}	the number of CPU cycles for a instruction on MEC_m
$v_{i,j}, V_i$	the j-th task and the tasks set of UE_i
$r_{i,j}$	the computation requirement of $v_{i,j}$
$d_{i,j}$	the communication requirement of $v_{i,j}$
$ram_{i,j}$	the memory requirement of $v_{i,j}$
$P_{i,m}^t$	The transmission power from UE_i to MEC_m
$R_{i,m}$	the communication speed from UE_i to MEC_m
$T_{i,j,m}$	the execution time of $v_{i,j}$ on MEC_m
$T_{i,j,0}$	the execution time of $v_{i,j}$ on UE_i
$T_{i,j}^e$	the real execution time for task $v_{i,j}$
$T_{i,j,m}^c$	the communication time of $v_{i,j}$ from UE_i to MEC_m
$T_{i,j}^c$	the real communication time for self-data of task $v_{i,j}$
$T_{i,j}^s$	the start time of task $v_{i,j}$
$T_{i,j}^{end}$	the finish time of task $v_{i,j}$
$E_{i,j}^e$	the computation energy consumption of $v_{i,j}$ on UE_i
$E_{i,j}^d$	the transmission energy for self-data of $v_{i,j}$
$E_{i,j}^p$	the receive energy from parents of $v_{i,j}$
$T_{i,j}^{delay}$	The processing latency of $v_{i,j}$
T_i^{total}	the total latency of UE_i in a period
$TC_{i,j}$	the communication time of task $v_{i,j}$
$T_i^p(u,j)$	the communication time from task $v_{i,u}$ to $v_{i,j}$
$E_i^p(u,j)$	the transmission energy from $v_{i,u}$ to $v_{i,j}$
$Cost_{i,j}^{ue}$	the total energy consumption on UE_i for $v_{i,j}$
$Cost_{i,j}^{mec}$	the service cost for $v_{i,j}$
$Cost_{i,j}^{total}$	the total cost consumption 0f $v_{i,j}$

precedence constraint, meaning task $v_{i,j}$ cannot be executed until task $v_{i,u}$ has finished. The value associated with $e_i(u,j)$ indicates the amount of data from $v_{i,u}$ required for the execution of task $v_{i,j}$.

Each subtask $v_{i,j}$ can be executed locally on UE_i, offloaded to an edge server, or further offloaded to a cloud server. The UE sends its application offloading requests to the scheduler at the network's edge. The scheduler then makes decisions based on period time, energy consumption, computation fees, and resource constraints, determining whether to execute tasks locally or on a server. This ensures efficient task allocation, aiming for lower total cost including UE's energy consumption and servers' computation fees.

In our model, we assume that the input DAG has been pre-processed and decomposed into atomic task nodes. Each node represents a fine-grained subtask that cannot be further partitioned, either due to data dependencies or granularity constraints. This assumption aligns with common

practice in DAG-based computing frameworks such as TensorFlow and Apache Spark, where the DAG structure reflects post-partitioned operations. Consequently, each node is assigned to a single processor for execution, and our focus lies in optimizing the placement and scheduling of these subtasks in multi-user edge environments.

3.2 Task Assigned and Execution Models

Task assignment: Each task $v_{i,j}$ can be assigned to a server for remote execution or to the UE for local execution. Let the binary variable $x_{i,j,m}$ denote whether task $v_{i,j}$ is scheduled on server MEC_m. If task $v_{i,j}$ is assigned to server MEC_m, then $x_{i,j,m} = 1$; otherwise, $x_{i,j,m} = 0$. Each task node can execute on only one processor (UE or server):

$$\sum_{m=0}^{M} x_{i,j,m} = 1, \quad \forall i \in [1, N], \forall j \in [1, N_i]. \tag{1}$$

The processor $P(i, j)$ on which task $v_{i,j}$ is executed can be defined as

$$P(i, j) = \sum_{m=0}^{M} m \times x_{i,j,m}, \quad \forall i \in [1, N], \forall j \in [1, N_i]. \tag{2}$$

Here, if $x_{i,j,0} = 1$, then $P(i, j) = 0$ that means the task $v_{i,j}$ is executed on local UE_i.

Local execution model: If task $v_{i,j}$ is not offloaded and is executed locally on UE_i, the computation time of $v_{i,j}$ on UE_i is given by

$$T_{i,j,0} = \frac{r_{i,j}\beta_i^{ue}}{f_i^{ue}}, \tag{3}$$

where f_i^{ue} represents the CPU-cycle frequency of UE_i and β_i^{ue} is the number of CPU cycles needed to execute one instruction on UE_i.

Offloading execution model: If task $v_{i,j}$ is offloaded to server MEC_m, the computation time of $v_{i,j}$ on server MEC_m can be expressed as

$$T_{i,j,m} = \frac{r_{i,j}\beta_m^{mec}}{f_m^{mec}}, \tag{4}$$

where f_m^{mec} denotes the CPU-cycle frequency of server MEC_m and β_m^{mec} is the number of CPU cycles needed to execute one instruction on server MEC_m.

Therefore, the actual computation time for task $v_{i,j}$ can be obtained by the following equation:

$$T_{i,j}^{e} = \sum_{m=0}^{M} T_{i,j,m} x_{i,j,m}. \tag{5}$$

3.3 Modeling Competition Among Multiple Users

In our system, multiple users share a limited set of edge and cloud resources. When tasks from different users are offloaded to the same MEC server, resource contention arises, including competition for CPU cycles and memory capacity. To model this contention, we introduce the following computation capacity constraint for each MEC server MEC_m:

$$\sum_{i=1}^{N} \sum_{j=1}^{N_i} x_{i,j,m} \cdot \frac{r_{i,j}\beta_m^{mec}}{f_m^{mec}} \leq C_m, \quad \forall m \in [1, M], \tag{6}$$

where C_m denotes the maximum available CPU time in each scheduling period at MEC_m. This constraint ensures that server resources are not oversubscribed and that offloading decisions are aware of inter-user competition.

Moreover, when multiple users compete for the same server, tasks may experience queuing delays and increased waiting times, which directly impact execution latency and energy consumption. Our scheduling algorithm, introduced in Section 4, addresses this issue by jointly optimizing task placements while minimizing both UE-side energy usage and server-side computation costs.

3.4 Communication Model

For task $v_{i,j}$, the communication consists of two parts: the communication for its self-data volume $d_{i,j}$ and the communication for receiving required data from its parent tasks.

Communication time for the self-data $d_{i,j}$: If task $v_{i,j}$ is offloaded to server MEC_m for remote execution, the communication time $T^c_{i,j,m}$ of its self-data $d_{i,j}$ can be defined as

$$T^c_{i,j,m} = \frac{d_{i,j}}{R_{i,m}}, \tag{7}$$

where $R_{i,m}$ represents the transmission rate between UE_i and MEC_m. Therefore, the actual communication time $T^c_{i,j}$ required for the self-data $d_{i,j}$ of task $v_{i,j}$ is defined as

$$T^c_{i,j} = \sum_{m=1}^{M} T^c_{i,j,m} x_{i,j,m}. \tag{8}$$

Communication time for receiving data from parents: If task $v_{i,u}$ is a parent of task $v_{i,j}$, executing task $v_{i,j}$ requires receiving data from task $v_{i,u}$. If both tasks are processed on the same processor, the communication time is 0. If the tasks are processed on different processors, the communication time depends on the amount of data to be accessed and the bandwidth between the processors. Thus, the communication time $T^p_i(u, j)$ required to receive data from task $v_{i,u}$ can be defined as

$$T^p_i(u,j) = \begin{cases} \dfrac{e_i(u,j)}{R_{k,h}}, & k \neq h, e_i(u,j) \in E_i \\ 0, & k = h, e_i(u,j) \in E_i \end{cases} \tag{9}$$

where $k = P(i, j)$ and $h = P(i, u)$ are the processors for executing task $v_{i,j}$ and $v_{i,u}$, respectively.

3.5 Total Execution Time

The start time $T^s_{i,j}$ of task $v_{i,j}$ depends on the finish time of all its parent tasks. If task $v_{i,u}$ is a parent of task $v_{i,j}$, then the start time $T^s_{i,j}$ of task $v_{i,j}$ can be calculated by

$$T^s_{i,j} = \max(T^s_{i,u} + T^e_{i,u} + T^p_i(u,j)). \tag{10}$$

In the case where a task $v_{i,j}$ has no parent tasks, its start time $T^s_{i,j}$ depends on the time required to transfer its self-data $d_{i,j}$, we can define this start time $T^s_{i,j}$ as

$$T^s_{i,j} = T^c_{i,j}. \tag{11}$$

By adding the start time and the execution time, we get the finish time of the task. Thus, the finish time $T^{end}_{i,j}$ of task $v_{i,j}$ is given by

$$T^{end}_{i,j} = T^e_{i,j} + T^s_{i,j}. \tag{12}$$

Therefore, the total execution time T^{total}_i of the DAG G_i is the maximum of the finish times of all tasks in G_i, it can be defined as

$$T^{total}_i = \max(T^{end}_{i,j}). \tag{13}$$

3.6 Cost Model

There are two components in the cost consumption i.e., energy consumption by local UEs and computation fees for servers.

Energy consumption by local UEs: Following conventional mobile computing models [25], we adopt a dynamic CPU power consumption model where the energy consumption of local execution is defined as

$$E^e_{i,j} = \xi_i (f^{ue}_i)^2 \times T_{i,j,0} \times x_{i,j,0}, \tag{14}$$

where ξ_i is a device-specific constant representing the effective switched capacitance. If a task is offloaded to server MEC_m, the energy consumption relates to communication. Let $P^t_{i,m}$ be the transmission power of UE_i to MEC_m. Then, the energy consumption for transmitting $d_{i,j}$ from UE_i to MEC_m is

$$E^d_{i,j} = \sum_{m=1}^{M} P^t_{i,m} \times T^c_{i,j,m} \times x_{i,j,m}. \tag{15}$$

Additionally, if task $v_{i,j}$ is executed locally, we must consider the energy consumption of receiving data from parents processed remotely. Conversely, if executed on a server, the energy consumption of receiving data from parents processed locally must be considered. Let $E^p_i(u,j)$ represent the energy consumption of receiving data from its parent $v_{i,u}$. It can be defined as

$$E^p_i(u,j) = \begin{cases} T^p_i(u,j) \times P^t_{i,k}, \text{if } x_{i,j,0} = 1, x_{i,u,0} = 0, \\ T^p_i(u,j) \times P^t_{i,h}, \text{if } x_{i,j,0} = 0, x_{i,u,0} = 1. \end{cases} \tag{16}$$

Thus, the energy consumption of receiving data from its parents for task $v_{i,j}$ is defined as

$$E^p_{i,j} = \sum_{e_i(u,j) \in E_i} E^p_i(u,j). \tag{17}$$

Therefore, the total energy consumption for task $v_{i,j}$ on UE_i can be defined as

$$Cost^{ue}_{i,j} = pr \times (E^e_{i,j} + E^d_{i,j} + E^p_{i,j}), \tag{18}$$

where pr represents the price of unit energy consumption on UEs.

Computation fees on servers: The computation fees depend on the computation time of tasks on servers. Let $price_m$ denote the price of unit computation time on server MEC_m. The computation fees for each task $v_{i,j}$ on servers can be defined as

$$Cost^{mec}_{i,j} = \sum_{m=1}^{M} price_m \times T_{i,j,m} \times x_{i,j,m}. \tag{19}$$

In summary, the total cost consumption for task $v_{i,j}$ is defined as

$$Cost^{total}_{i,j} = Cost^{ue}_{i,j} + Cost^{mec}_{i,j}. \tag{20}$$

3.7 Problem Formulation

Assume a multi-user MEC system with N UEs and M servers, including edge and cloud servers. The computation frequency f^{ue}_i of each UE_i, the computation frequency f^{mec}_m of each server MEC_m, the communication power $P^t_{i,m}$, and the communication speed $R_{i,m}$ between processors are known in advance. The objective is to find a computation offloading strategy for each UE_i that minimizes

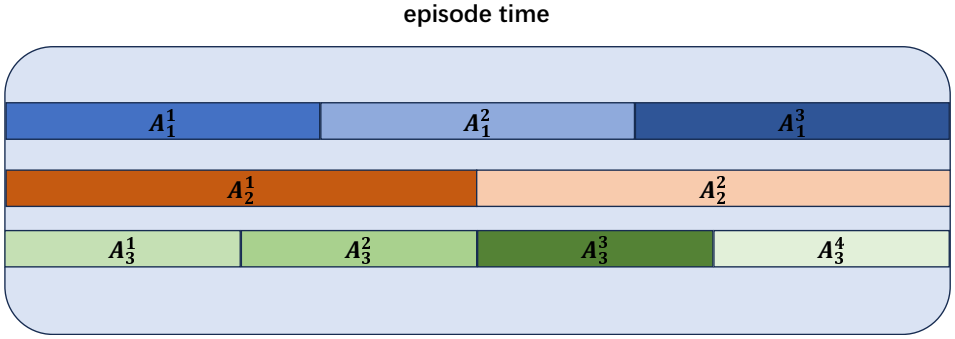


Fig. 2. The episode time.

total costs while satisfying the periodic tolerance latency T_i of each UE_i. Thus, the optimization problem can be formulated as

$$\begin{aligned} &\min_{x_{i,j,m}} \ Cost = \sum_{i=1}^{N} \sum_{j=1}^{N_i} Cost_{i,j}^{total}, \\ &s.t.\ C1 : \sum_{m=0}^{M} x_{i,j,m} = 1, \quad \forall i \in N, j \in N_i \\ &\quad C2 : T_i^{total} \leq T_i, \quad \forall i \in N \\ &\quad C3 : (6), \\ &\quad C4 : ram_{i,j} \leq x_{i,j,0} \cdot um_i^{free} + \sum_{m=1}^{M} x_{i,j,m} \cdot sm_m^{free}, \end{aligned} \tag{21}$$

where um_i^{free} and sm_m^{free} represent the free main memory of UE_i and MEC_m, respectively, that can be allocated for executing tasks. Constraint $C3$ guarantees that server resources are not over-allocated and that offloading decisions consider the competition among users. Constraint $C4$ specifies that the available memory on the processor serving a task must be at least equal to the memory required for that task. For a server MEC_m, the available memory sm_m^{free} fluctuates based on the number of tasks it handles.

Since each application on a local user UE_i has its own period time T_i, the number of iterations executed by each UE varies over time, making it difficult to calculate the total computation cost for all UEs. Fortunately, we can reformulate the problem to minimize the computation cost over a specific optimization period. We call this optimization period the "episode time." To better reflect the actual scenario, we set this "episode time" to the **least common multiple** (**LCM**) of all the UE cycles T_i. For example, as shown in Figure 2, there are three applications A_1, A_2, and A_3. Suppose the periods of these three applications are 4 units, 6 units, and 3 units, respectively. The LCM of these three periods is the smallest positive integer that is divisible by all three periods. Thus, the LCM of these three periods is 12 units. Within these 12 units, applications A_1, A_2, and A_3 complete 3 cycles, 2 cycles, and 4 cycles, respectively. To evaluate the minimization of the total computation cost for these three applications, we only need to calculate the total computation cost within these 12 units. Therefore, the aforementioned objective function can be transformed into:

$$\min_{x} \ Cost = \sum_{i=1}^{N} \sum_{j=1}^{N_i} \sum_{g=1}^{\frac{LCM}{T_i}} Cost_{i,j,g}^{total}, \tag{22}$$

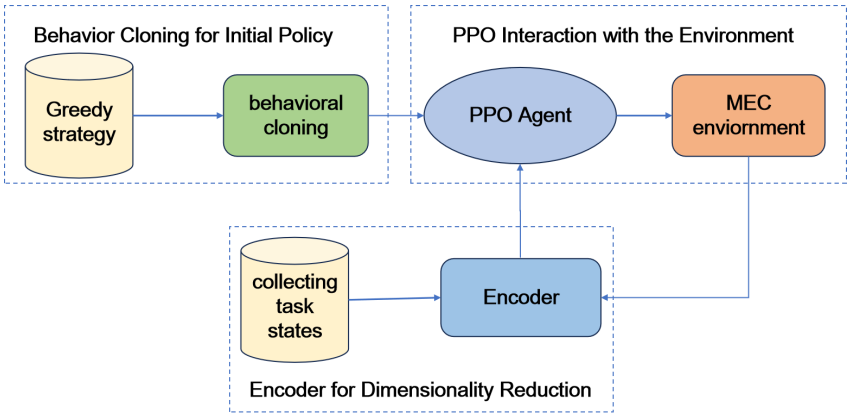


Fig. 3. The overview of PPO learning model.

where LCM is the *episode time*, and $Cost_{i,j,g}^{total}$ represents the cost of task $v_{i,j}$ in the gth iteration within its *episode time*.

4 PPO Learning Model for Periodic Task Offloading

4.1 The Overview of PPO

This section employs the PPO model to address the optimization challenges of periodic DAG-structured task offloading within multi-users MEC systems. The algorithm framework as shown in Figure 3 consists of three critical components: (1) **Behavior Cloning** (**BC**) for Initial Policy: The initial policy for the PPO agent is derived using BC. In this phase, the greedy algorithm's strategy is replicated and transferred to the PPO agent. This ensures that the agent starts with a reasonable policy based on pre-existing knowledge to handle the expansive action search space. (2) Encoder for Dimensionality Reduction: The Encoder is employed to process the task states, transforming them into low-dimensional representations. This step is essential for reducing the complexity of the state space, making it more manageable for the PPO learning model. (3) PPO Interaction with the Environment: The PPO agent interacts with the environment to learn and refine its policy. Through these interactions, the agent collects new experiences, which are used to update its policy.

4.2 Markov Decision Process

In our model, we formulate the task offloading process as a **Markov Decision Process** (**MDP**), where each state s_t captures the current execution context and the action a_t denotes the scheduling decision. The multi-user learning model is defined by a tuple $< S, A, P, R, \gamma >$, where S and A are the state and action spaces, respectively. P denotes the state-action probability, and R stands for the reward function. The parameter γ, ranging from 0 to 1, acts as a discount factor determining the importance of future rewards. We assume iterations are divided into multiple episodes, denoted as $t \in K$, with K being the number of episodes. During each episode t, the agent interacts with the environment, perceives the current state s_t, and selects an action a_t based on its policy $\pi(a_t|s_t)$, which maps states to actions. The state transition to s_{t+1} is deterministically defined by task completion rules, resource constraints, and system timing updates. That is, the environment follows a deterministic transition function $s_{t+1} = f(s_t, a_t)$.

Although the transition model is deterministic in simulation, the overall system still exhibits non-stationary and partially observable behavior due to multi-user interference and dynamic workloads. Therefore, the learning process remains complex and challenging. This modeling strategy

is consistent with existing literature on MEC scheduling [8], and facilitates reproducible evaluation of algorithm performance under controlled dynamics.

State Space: In our task offloading problem, the state represents the agent's observation within the MEC environment. Thus, the state in each episode consists of information regarding the statuses of all tasks and processors. Hence, the system's state space can be defined as

$$S = \{S_t | S_t = (S_t^{task}, S_t^{processors})\}, \tag{23}$$

where S_t^{task} and $S_t^{processors}$ are the state sets of tasks and processors, respectively. The set of task states S_t^{task} is determined by the output fused by a task encoder, expressed as follows:

$$S_t^{task} = Encoder(V_{tasks}^{'}, O_{adj}). \tag{24}$$

Here, $V_{tasks}^{'}$ represents the feature matrix of all tasks in the current environment, while O_{adj} denotes the symmetric matrix of all tasks' adjacency matrices. The formula for expressing $V_{tasks}^{'}$ is provided in Equation (25). The attribute vector $v_{i,j}^{'}$ for tasks includes: (a) computational workload of the task $d_{i,j}$; (b) memory size required for task execution $ram_{i,j}$; (c) size of task source codes (e.g., instructions) $r_{i,j}$; (d) tolerance of task delays T_i.

$$V_{tasks}^{'} = \{v_{i,j}^{'} | i \in [1, N], j \in [1, N_i]\} \tag{25}$$

For processor states, $S_t^{processors}$ represents the state matrix of all assignable computing nodes, including both local UEs and shared servers. The mathematical representation of $S_t^{processors}$ can be observed in Equation 26. Concerning the feature vector of an individual processor, it incorporates the following characteristics: (a) the average CPU load of the processors, (b) the processing capability of the CPU, (c) the remaining memory capacity, and (d) the utilization rate of network bandwidth. Its formula expression is as follows:

$$S_t^{processors} = \{s_t^{UE_i} | i \in N, s_t^{MEC_g} | g \in M\}, \tag{26}$$

where $s_t^{UE_i}$ is the state of local UE_i, and $s_t^{MEC_g}$ is the state of server MEC_m.

Action space: Actions involve assigning available services to tasks within applications. Therefore, the action a_t in the tth iteration (tth episode) involves assigning a server MEC_m to the current task $v_{i,j}$. Considering the placement configuration of each task $v_{i,j}$, we can define $a_t^{i,j}$ as

$$a_t^{i,j} = \{x_{i,j,m} | m \in [0, M]\} = P(i, j). \tag{27}$$

Since $P_{i,j} = 0$ denotes the task being assigned locally for execution, the action space A can be defined as the set of all available servers and local UE, presented as follows:

$$A = \{0, MEC_m | m \in [1, M]\}. \tag{28}$$

Reward function: The goal is to minimize the cost model while meeting the periodic time constraint. Thus, the reward function consists of two components: survival reward ($r_{survive}$) for satisfying the periodic time constraint and offloading computation cost reward (r_{cost}). The formula expression of the reward function is as follows:

$$R = r_{survive} + \mu \cdot r_{cost}, \tag{29}$$

where μ serves as the balancing coefficient to adjust the relative importance of the two rewards. The Survival reward aims to encourage the action to ensure that the applications of all local users

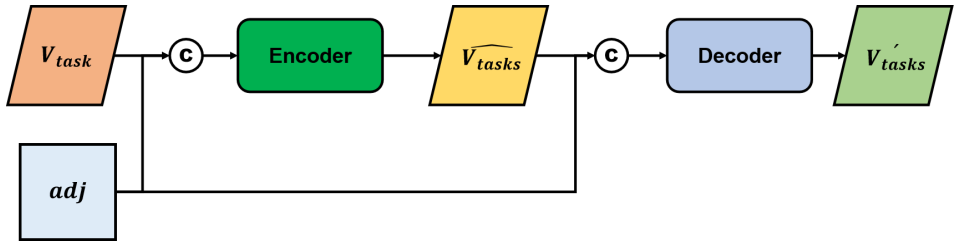


Fig. 4. The Encoder.

do not time out, thereby successfully completing the entire episode. Its formula can be expressed as

$$r_{survive} = \begin{cases} C, & \text{if } T_i^{total} < T_i, \forall i \\ -\infty. & \text{if } T_i^{total} > T_i, \forall i \end{cases} \tag{30}$$

where C represents a normal constant value. For simplicity, in this article, we set C as the LCM.

For each task $v_{i,j}$, the reward associated with offloading computation costs is adjusted through an exponential function to account for the cost impact, as shown in Equation (31):

$$r_{cost} = \sum_{v_{i,j}} \exp(-cost_{i,j}^{total}). \tag{31}$$

By calculating the exponential function of the negative total cost, we ensure that higher cost values result in a lower adjusted reward. This adjustment facilitates a tradeoff between the potential benefits and the associated costs of offloading the computation.

The PPO learning process is defined in a multi-agent MEC setting, where agents learn under shared resource constraints. The state space $S_t^{processors}$ includes real-time resource usage of shared MEC servers, which reflects how tasks from different users compete for limited computational and memory resources. This competition implicitly influences the action-selection process, as each user's agent must learn to avoid resource contention that could result in increased computation time or task failure.

The impact of this user-level competition is embedded into the reward function. The cost term $cost_{i,j}^{total}$ increases if the selected MEC node is congested due to offloading requests from multiple users. Therefore, the PPO agent is incentivized to learn cooperative strategies that reduce both task delay and overall energy consumption by considering the current state of shared resources. In this way, the multi-user competition is indirectly modeled through shared processor states and reflected in the PPO agents policy learning.

4.3 Encoder

Figure 4 illustrates the unsupervised process pipeline diagram of the Encoder employed in this model. The Encoder primarily combines the node feature V_{tasks} with the structural details of the input tasks to derive the fused tasks feature denoted as $\widehat{V_{tasks}}$. The task features V_{tasks} contain the state information of all current tasks and can be represented as follows:

$$V_{tasks} = \{\overrightarrow{v_{i,j}} | i \in [1, N], j \in [1, N_i]\}, \tag{32}$$

where $\overrightarrow{v_{i,j}}$ is a task state vector for task $v_{i,j}$.

Conversely, the Decoder reconstructs the fused data by utilizing $\widehat{V_{tasks}}$ and structural information as inputs. After extracting the feature structural details embedded by the Encoder, it generates the de-fused outcome denoted as $V_{tasks}^{'}$.

To ensure that the information fused by the task graph encoder possesses robust discriminative capabilities and comprehensively encompasses task graph nodes along with structural information, we employ the following structural loss function and feature loss function to train the encoder.

To enhance the discriminative capabilities of the fused task information $\widehat{V_{tasks}}$ and ensure that the similarity between neighboring tasks is higher than that between non-neighboring tasks, we introduce a structured loss function called StructureLoss. This function aims to optimize the similarity among neighboring nodes and reinforce the representation of structural information. The structured loss function is defined as follows:

$$StructureLoss = \sum_{i,j} \sum_{l \in N_{i,j}} \log \frac{1}{1 + \exp(-\widehat{v_{i,j}}^T \widehat{v_l})}, \tag{33}$$

where $\widehat{v_{i,j}}$ and $\widehat{v_l}$ are the node feature vector obtained after fusing task information and structural information. $N_{i,j}$ represents all the neighbors of task $v_{i,j}$. By optimizing this loss, we reinforce the representation of structural information in the fused task features, thereby improving the discriminative capabilities of the model.

To ensure that the feature vector output by the decoder closely matches the feature vector input to the encoder, an introduced feature loss function called FeatureLoss is employed. This function aims to minimize the difference between the task features before and after encoding, ensuring that the encoder effectively captures and retains the original graph's node information and structural characteristics. The formula for the feature loss function is defined as follows:

$$FeatureLoss = \sum_{i=1}^{N} \sum_{j=1}^{N_i} (\overrightarrow{v_{i,j}} - v'_{i,j})^2, \tag{34}$$

where $\overrightarrow{v_{i,j}}$ and $v'_{i,j}$ are the original feature vector and the feature vector output by the decoder, respectively. Minimizing this loss encourages the encoder to capture and preserve the important features of the input tasks during the encoding process.

4.4 Behavior Cloning

BC technology can address challenges caused by large action search spaces in an environment. In this approach, before training the intelligent agent using the PPO model, the policy of a greedy algorithm is initially replicated. Although the greedy algorithm may not represent the optimal allocation strategy, it does possess the ability to generate multiple trajectories that satisfy the latency constraints imposed on the intelligent agent during its initial training phases. As a result, this technique significantly reduces the time required for the agent to explore strategies aligned with these constraints.

The BC technology is shown in Algorithm 1. In this cloning algorithm, the process begins with the initialization of the Actor network and data buffer (Line 1). Subsequently, interaction with the task policy obtained from the greedy algorithm occurs within the environment, and the resulting task state-action pairs (s_t^{task}, a_t) are recorded (Lines 3–4). These task states s_t^{task} are then processed by the Encoder to generate new state-action pairs (s_t^e, a_t), which are saved in the data buffer (Lines 5–6). When the data volume in the buffer reaches the threshold N_{buffer}, the following operations are performed. For each episode, N_{batch} samples are randomly selected from the data buffer to create mini-batches of data D_{batch} (Line 9). States S_t and actions A_t are then extracted from these mini-batches. The states S_t are then passed into the Actor network to obtain a probability distribution P over each action (Lines 10–11). The probability (p_t) of executing actions a_t based on a greedy strategy is calculated by utilizing a gather operation with A_t (Line 12). In each episode, the Actor

ALGORITHM 1: Behavior Cloning

Input: Total training iterations $N_{episode}$, Learning rate for the Actor: α_{actor}, Batch size N_{batch}, the strategy from Greedy, Data buffer size D_{buff}

Output: Actor network $\pi(\theta^k)$

Initially set Actor network $\pi(\theta^k)$ and data D in buffer;
while $|D| < D_{buff}$ **do**
    Get task state s_t^{task} from environment;
    Get the actor action a_t from Greedy strategy following the task state s_t^{task};
    Obtain task state s_t^e through fusion s_t^{task} by Encoder;
    store (s_t^e, a_t) in data buffer;
end while
for $k = 1, 2, \ldots, N_{episode}$ **do**
    Randomly sample N_{batch} samples from the data buffer to form a data batch D_{batch};
    Obtain the state set S_t and Actor set A_t for each sample;
    Get the set of actions probabilities P_t from Actor network based on the input S_t;
    Gather P_t and A_t to obtain the actions probabilities p_t;
    Update Actor network $\pi(\theta^k)$ through Equation (35);
    Optimize gradients using the Adam optimizer;
end for
Return θ;

network is updated by maximizing the probability p_t according to Equation (35). It is worth noting that we employ the Adam optimizer to perform gradient descent optimization on the gradients in the cloning algorithm.

$$\theta_{k+1} = \arg\min_{\theta} \frac{1}{N_{batch}} \left(\sum_{i=1}^{N_{batch}} \log(p_t) \right)^2 \tag{35}$$

Greedy matching algorithm: Next, we introduce the low-complexity greedy matching algorithm, depicted in Algorithm 2. This algorithm determines task offloading and resource allocation for periodic DAG applications on UEs. The greedy matching algorithm consists of two parts. The first part involves selecting the processor with the lowest cost for executing each task (Lines 1–14). In this part, if the total cost of offloading to server MEC_m is lower than the current cost, the task is assigned to the server MEC_m and the cost is updated. After evaluating all servers, the costs $E_{i,j}^m$ are sorted in ascending order. In the second part (Lines 15–31), for each UE_i, tasks are evaluated for delay $T_{i,j}^{delay}$ and total cost $Cost_{i,j}^{total}$, then sorted in descending order based on $T_{i,j}^{delay}$. If the total delay T_i^{total} exceeds the threshold T_i, we adjust the task allocation. The adjustment method selects the task with the longest execution time on the critical path for adjustment (Line 22). It incrementally increases the costs to adjust the machine until the adjusted task time is less than the second-longest task execution time on the critical path (Lines 23–29). It then chooses a new task for adjustment. Once the total execution time of the DAG meets the time constraints, the adjustment stops and a greedy strategy is obtained. Through the greedy strategy, the PPO agent obtains the initial action a_t and state s_t.

In the low-complexity greedy matching algorithm, it takes $O(N_T \cdot M \log M)$ to calculate costs and to sort them for each task, where N_T is the total number of tasks, i.e., $N_T = \sum_i N_i$. Reassigning tasks based on delays costs $O(N_T \log N_T)$. Therefore, the total time complexity of the algorithm is $O(N_T \cdot M \log M + N_T \log N_T)$.

ALGORITHM 2: Greedy Task Match Strategy

Input: $v_{i,j}$, f^{ue}, f_m^{mec}, β^{ue}, β_m^{mec}, $P_{i,m}^t$
Output: $P(i,j), x_{i,j,m}$

for each task $v_{i,j}$ **do**
  calculate $E_{i,j}^e$;
  $P(i,j) = 0$;
  $cost(i,j) = E_{i,j}^e * pr$;
  for $m = 1, 2 \ldots, M$ **do**;
    calculate $E_{i,j}^d$, and $Cost_{i,j}^{mec}$;
    if $cost(i,j) > pr * E_{i,j}^d + Cost_{i,j}^{mec}$ **then**
      $P(i,j) = m$;
      $cost(i,j) = pr * E_{i,j}^d + Cost_{i,j}^{mec}$;
    end if
    Store $E_{i,j}^m \leftarrow pr * E_{i,j}^d + Cost_{i,j}^{mec}$;
  end for
  Sort the element $E_{i,j}^m$ in ascending order;
end for
for each UE_i **do**
  for $j = 1, 2, \ldots N_i$ **do**
    calculate $T_{i,j}^{delay}$, $Cost_{i,j}^{total}$;
  end for
  Sort the element $t_i^{order} \leftarrow v_{i,j}$ with $cp_i = 1$ in descending order with $T_{i,j}^{delay}$;
  calculate T_i^{total};
  while $T_i^{total} > T_i$ **do**
    select the first task $v_{i,h}$ from t_i^{order} to reassign;
    select the second task $v_{i,u}$ from t_i^{order}, let $T_i^{max} = T_{i,u}^{delay}$;
    while $T_{i,h}^{delay} > T_i^{max}$ and $T_i^{total} > T_i$ **do**
      $k = GetTopItem(E_{i,j}^m)$;
      $P(i,h) = k$;
      Calculate $T_{i,h}^{delay}$ and T_i^{total};
      $RemoveTopItem(E_{i,j}^m)$;
    end while
    $RemoveTopItem(t_i^{order})$;
  end while
end for
calculate $Cost$;

4.5 Periodic DAG-Structured Task Offloading and Resource Allocation

This article proposes a novel PPO model within an actor-critic framework to address the challenges of offloading periodic DAG applications in diverse MEC environments. In this framework, the policy is directly parameterized as $\pi(a_t|s_t;\theta)$, where θ is updated through gradient ascent on the variance of the expected total future discounted reward and the learned state-value function under policy $V^{\pi}(s_t)$. This approach aims to combine the strengths of value-based and policy-based methods while reducing their respective limitations.

Actor-critic framework: As shown in Figure 5, the PPO learning model consists of an Actor network and a Critic network. The Actor network generates actions a_t for the intelligent agent to execute, while the Critic network estimates the value function $V^{\pi}(s_t)$ for the current state s_t. By incorporating the immediate reward r_t obtained after the agent performs action a_t, the Critic network evaluates the current action, thereby improving the stability of training the Actor network.

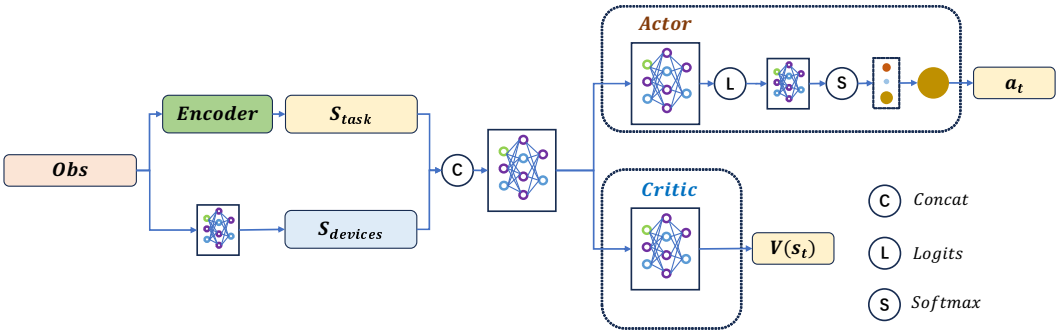


Fig. 5. The task offloading network on PPO learning model.

— Critic network update policy: The Critic network estimates the value function V and updates it using TD-error to avoid positive feedback loops. The general form of the TD-error update formula in the Critic network is

$$\Delta\phi_t(\omega) = \gamma V(S_{t+1}) + r_t - V(S_t), \tag{36}$$

where $\Delta\phi_t(\omega)$ is the TD-error, r_t is the immediate reward at iteration t, and $V(S_t)$ and $V(S_{t+1})$ are the predicted values of the current and next states, respectively.The Critic network is updated by minimizing the squared TD-error, adjusting the estimated state values toward more accurate predictions. The loss function associated with the TD-error in the Critic network is defined as

$$\begin{aligned} L_{critic}(\theta) &= \sum_t (\Delta\phi_t(\omega))^2 \\ &= \mathbb{E}_{S_1}[(\gamma V(S_{t+1}) + r_t - V(S_t))^2]. \end{aligned} \tag{37}$$

— Actor network update policy: The Actor network is updated using the PPO algorithm to ensure that the difference between the behavioral policy θ' and the target policy θ does not become too large. This is achieved by applying a clip function to restrict the range of gradient updates. The update formula for the Actor network is as follows:

$$L^{clip}(\theta) = \mathbb{E}_t[\min(J_t(\theta)\cdot A_t, \text{clip}(J_t(\theta), 1-\epsilon, 1+\epsilon)\cdot A_t)], \tag{38}$$

where ε is the clipping threshold, A_t is the advantage of taking action a_t in state s_t, and $J_t(\theta)$ is the ratio of the new policy to the old policy. $J_t(\theta)$ is defined as follows:

$$J_t(\theta) = \frac{\pi_{new}(a_t|s_t,\theta)}{\pi_{old}(a_t|s_t,\theta)}, \tag{39}$$

where $\pi_{old}(a_t|s_t,\theta)$ and $\pi_{new}(a_t|s_t,\theta)$ are probability of taking action a_t in state s_t under the old policy and the updated policy, respectively. The clip function constrains $J_t(\theta)$ within the interval $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter controlling the extent of clipping.

PPO task offloading strategy: In the periodic DAG-structured task offloading algorithm for multiple users, as shown in Algorithm 3, PPO agents receive observation data *Obs* from the environment. This data includes task observations S_{task} and processor states $S_{processor}$. Using a fixed processing mechanism called the Encoder, these observations are combined to produce the task state S_t^{task}, which serves as the input S_t for the PPO learning model. To begin, the actor and critic networks, along with the experience replay buffer, are initialized (Lines 1–2). Before interacting with the environment, the PPO agent clones the greedy strategy (Algorithm 2). For each episode,

ALGORITHM 3: PPO Task Offloading Strategy

Input: Total iterations $N_{episode}$ in PPO, the training time of each episode N_{train}, learning rates $\alpha_{encoder}$, $\alpha_{decoder}$, α_{actor}, α_{critic}, exploration parameter ε, replay buffer D_{PPO}
Output: Actor network $\pi(\theta^t)$

1: Initialize the actor network $\pi(\theta^t)$ and critic network $V(S_t)$
2: Initialize the experience replay buffer D_{PPO}
3: Obtain the greedy strategy via Behavior Cloning
4: **for** $t = 1$ to $N_{episode}$ **do**
5: Execute policy $\pi = \pi(\theta^t)$ in the environment
6: Perform action $a_t \leftarrow \{x_{i,j,m} \mid i \in [1, N], j \in [1, N_i], m \in [0, M]\}$
7: Observe s_{t+1}, r_t, p_t
8: Encode s_{t+1} to s^e_{t+1}
9: Store experience $[s^e_t, a_t, r_t, p_t, s^e_{t+1}, done]$ in replay buffer D_{PPO}
10: **for** $k = 1$ to N_{train} **do**
11: Calculate the advantage function A_t using $V(S_t)$
12: Update actor network $\pi(\theta^t)$ via Equation 40
13: Optimize gradients with the Adam optimizer
14: Update critic network $V(S_t)$ via Equation 41
15: Optimize gradients with the Adam optimizer
16: **end for**
17: **end for**
18: **Return** the trained actor network $\pi(\theta^t)$

the policy $\pi(\theta^t)$ is executed in the environment, performing actions and observing the next state s_{t+1}, reward r_t, and probability p_t. Based on the action a_t, the system transitions to the next state s_{t+1} (Lines 6–7). The Encoder processes s_{t+1} to yield s^e_{t+1}, and the experience $[s^e_t, a_t, r_t, p_t, s^e_{t+1}, done]$ is stored in the replay buffer (Lines 8–9). Training occurs for N_{train} iterations per episode, where the advantage function A_t is calculated using the critic network $V(S^t)$ (Lines 10–11). During each training iteration, the Actor and Critic networks are updated using their respective loss functions. The parameters of the Actor network are updated using Equation 40, and the parameters of the Critic network are updated using Equation 41 (Lines 11–15).

$$\theta_{k+1} = \arg\min_{\theta} \frac{1}{D_{PPO}} \sum_{T \in D_{PPO}} L^{clip}(\theta); \tag{40}$$

$$S_{t+1} = \arg\min_{\omega} \frac{1}{D_{PPO}} \sum_{T \in D_{PPO}} \sum_{i=0}^{I} (\Delta\phi_t(\omega))^2. \tag{41}$$

We use the Adam optimizer for stochastic gradient ascent when optimizing the Actor gradients, and for stochastic gradient descent when optimizing the Critic gradients. During network updates, the advantage function A_t is defined as

$$A_t = Q^{\pi}_t(s_t, a_t) - V^{\pi}_t(s_t), \tag{42}$$

where $Q^{\pi}_t(s_t, a_t)$ is the state-action value function, predicting the expected return starting from state s_t, taking action a_t, and following policy π.

5 Experiments

5.1 Experimental Setup

This study employs simulation experiments to evaluate the performance of our proposed PPO task offloading algorithm. In a multi-user MEC environment, there are **five UEs**, two cloud servers,

Table 2. Table of Parameters in the MECs Environment

Parameter	Value
Number of local users (UEs)	5
memory size of local users	512MB
memory size of edge servers	8GB
frequency of local users	$6 \times 10^7 Hz$
Number of edge servers	5
Number of cloud servers	2
Bandwidth between edge servers	100Mbps
Bandwidth to cloud servers	200Mbps
Cores of each edge server	4
Cores of each local user	1

and five edge servers in the system. Each UE operates independently and generates a DAG-based application with distinct complexity. The transmission bandwidth of UEs ranges from 5 to 30 Mbps. The data volume and computational workload of UE tasks are evenly distributed within the ranges of 0.2–2.0 MB and 3×10^4-2×10^6 cycles/bit, respectively. Table 2 presents the detailed parameters in the MEC environment. The experiments were conducted on a Windows 10 system using an AMD Ryzen R7 7735HS processor running at 3.2 GHz with 16 GB of memory. Python 3.9 and PyTorch 1.2.1 were used for implementation.

Various real-world IoT applications can be represented using DAGs that feature diverse task counts and dependency structures. Therefore, we created multiple synthetic DAG sets, each with varying task numbers and dependency patterns, to simulate scenarios where UEs produce heterogeneous DAGs with distinct characteristics. The DAG applications of UEs in this study are generated using the method described in [10], where the shape of the DAG can be adjusted by tuning four parameters: $NodeNum$, $MaxOutDegree$, α, and β. Here, $NodeNum$ determines the total number of tasks in a DAG, $MaxOutDegree$ restricts the maximum outdegree of nodes, α determines the depth of the DAG, and β determines the alignment rate of the DAG. In this study, we set α=0.5 and β=0.5. Considering the heterogeneity of tasks on each local user, the number of tasks ($NodeNum$) and depth ($MaxOutDegree$) are set differently for applications on different UEs. To simulate diverse application scenarios in MEC environments, we construct three types of periodic DAG applications corresponding to different computational load levels:

— Light-load applications: $NodeNum \in [10, 15]$, $MaxOutDegree \in [2, 3]$
— Medium-load applications: $NodeNum \in [20, 25]$, $MaxOutDegree \in [3, 4]$
— Heavy-load applications: $NodeNum \in [30, 40]$, $MaxOutDegree \in [4, 5]$

As defined in Section 3.5, the total execution time T_i^{total} of each DAG application is determined by the longest dependent path in the task graph after scheduling decisions are applied. In our experiments, we use the LCM of all users DAG execution times T_i^{total} as the unified episode time. This allows each UE to complete an integer number of DAG execution cycles within a unified evaluation window. In our experiments, the values of T_i^{total} are computed after task scheduling decisions are made, based on the execution paths and data dependencies defined in Section 3.5. This design ensures that the episode window is dynamically aligned with the actual execution characteristics of each user's DAG. The resulting episode time typically ranges between 50 and 200 simulation time units depending on the DAG structure and system configuration.

Table 3. The PP0 and Training Hyperparameters

Parameter	Value	Parameter	Value
MLP hidden units	256	Loss Coefficient	0.5
Cross-entropy	0.01	discount Factor γ	[0.8-0.98]
Truncation constant	0.2	Policy Learning rate	[1e-6,2e-2]
Balancing coefficient μ	1.5	Critic Learning rate	[2e-6,2e-2]

Table 4. The Encoder and Training Hyperparameters

Parameter	Value	Parameter	Value
Dropout	0.2	hidden layer Dimension	10
GAT layer	3	Dimension of the output	20
heads	5	optimization Method	Adam
Encoder Learning rate	1e-3	decoder Learning rate	5e-3

5.2 PPO Hyperparameters

Hyperparameters such as learning rate, truncation coefficient, and discount factor directly impact the convergence speed of the algorithm. Table 3 provides the hyperparameter settings used for training the algorithm, including the activation functions and optimizers for the network structure. In the PPO task offloading model, an Encoder is used to perform dimensionality reduction on the graph structure and information. The detailed configuration of the network structure and specific hyperparameters for training the Encoder can be found in Table 4.

5.3 Performance Comparison

To evaluate the effectiveness of our proposed PPO task offloading algorithm (PPO) in this article, we employed six benchmark offloading strategies for performance comparison. These strategies include:

— Cloud-only: All computational tasks are completely offloaded to the cloud servers.
— Edge-only: Similar to Cloud-only, all tasks are offloaded to edge servers.
— Greedy Algorithm (Greedy): Utilizes the greedy algorithm strategy to perform tasks.
— DDRL: It is the extended and adapted version of the technique proposed in [10]. We extended this technique so that it can be used in multi-users MEC to minimize the weighted cost of energy consumption and computation fees.
— DDPG [1]: Similar the DDRL, we extended this technique so that it can be used in multi-users MEC to solve DAG-structured task offloading.
— DQN: DQN has been widely used by many researchers to solve task offloading problems [2, 5, 19]. In our experiment, we implemented the optimized DQN algorithm with an adaptive exploration for task offloading in multi-users MEC environments. The hyperparameters of this technique are set based on [19], which is a state-of-the-art DQN-based task offloading technique for DAG-based IoT applications.

Figure 6 shows the average weighted cost performance of each episode under various strategies with five UEs and a cycle shrinkage rate ξ of 0.6. The figure shows that the strategies of full offloading to the cloud servers and full offloading to the edge servers result in the highest costs. However, the costs incurred by RL algorithms are better than those of the traditional greedy algorithm. Greedy remains constant at 2080, never reducing, and algorithms like DQN and DDRL show more fluctuations in cost, with DQN dropping rapidly but plateauing higher than PPO. From the figure we can see that our proposed PPO task offloading (PPO) algorithm initially performs similarly to

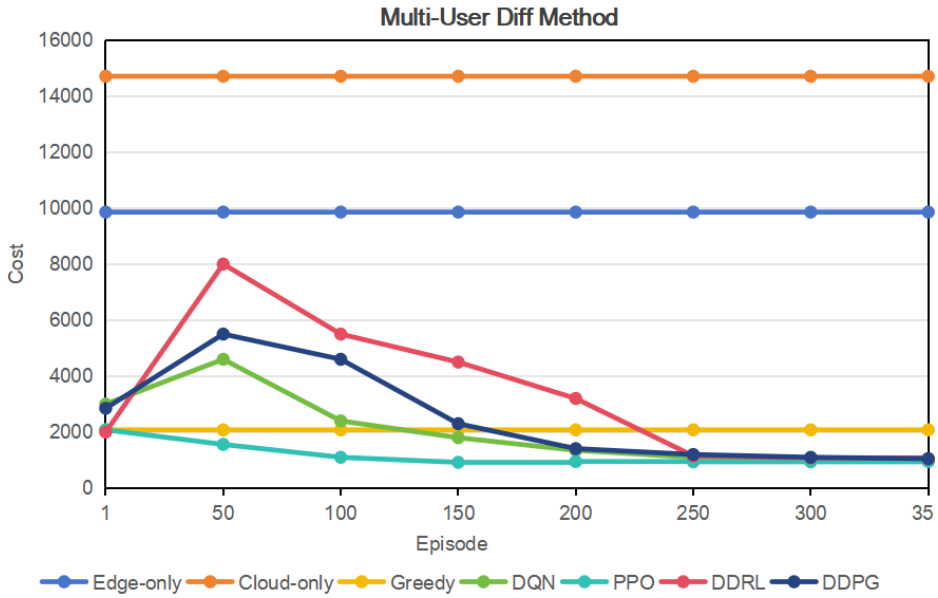


Fig. 6. Cost performance of different algorithms for multi-user on MECs.

the greedy algorithm. Then, the PPO algorithm shows significant cost reduction as the number of episodes increases. Starting from a cost of 2080 at the first episode, it quickly drops to 1560 at 50 episodes and reaches a low of around 940 from episode 200 onward. Compared to other RL algorithms like DQN or DDRL, PPO maintains consistently lower costs, especially after 150 episodes, which demonstrates its efficiency and adaptability in reducing costs over time. Therefore, PPO clearly outperforms other approaches in cost efficiency over a long period, proving its superior optimization capabilities in this context.

Figure 6 also illustrates the convergence behavior of different algorithms over training episodes. We observe that both DDRL and DDPG exhibit lower performance than the Greedy baseline during the early stages (before episode 200). This is primarily due to the high variance and sample inefficiency inherent in policy gradient methods when applied to sparse and delayed reward settings, such as DAG-based task offloading. In contrast, the Greedy method-although lacking learning capability-applies fixed heuristics that produce stable and deterministic behavior from the beginning, which gives it a temporary advantage in early training. Moreover, we note that DQN achieves better performance than DDPG across most of the training horizon. This result may seem counterintuitive, as DDPG is theoretically more powerful. However, the offloading problem in our setting involves discrete server selections, which are more naturally handled by value-based methods like DQN. DDPG, originally designed for continuous action spaces, requires action discretization in our implementation. This introduces approximation errors and leads to suboptimal performance due to the mismatch between the algorithm's action representation and the task's intrinsic discreteness. Our proposed PPO-based method consistently outperforms all baselines. By combining stable policy optimization with a task-specific encoder and BC initialization, PPO achieves faster convergence, better early-stage performance, and higher final reward. This validates the effectiveness of our design in handling both the structural complexity of DAGs and the competition-aware multi-user offloading environment.

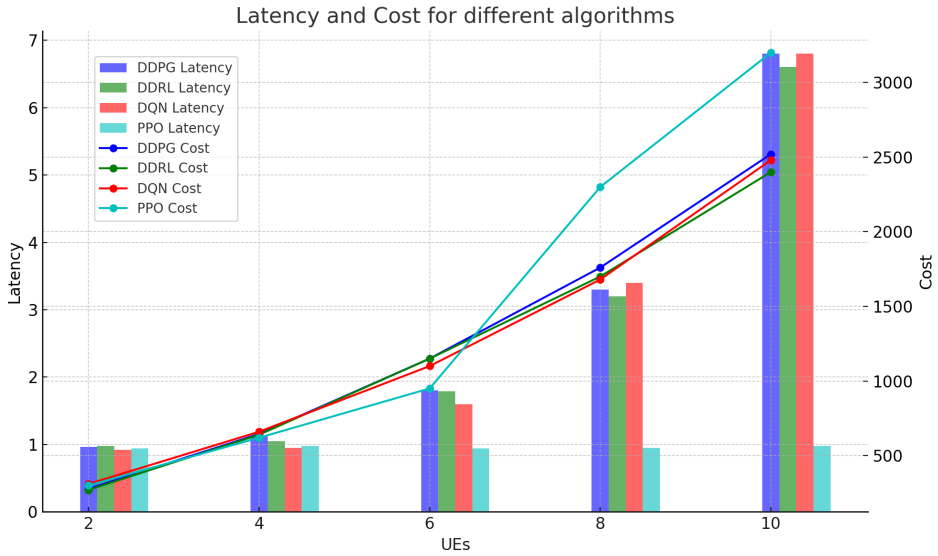


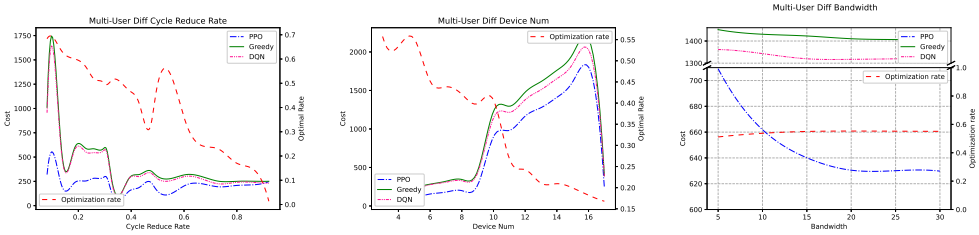
Fig. 7. Latency and cost comparison under different RL learning model.

To compare latency across DAG applications of different sizes and topologies, we normalize task completion times using the DAGs critical path. For each application, we define:

$$\text{Normalized Latency} = \frac{1}{|V|} \sum_{v \in V} \frac{T_v^{finish} - T_v^{start}}{T^{critical}} \tag{43}$$

where $|V|$ is the number of nodes in the DAG, T_v^{finish} and T_v^{start} are the actual finish and start times of node v, and $T^{critical}$ denotes the length of the critical path, i.e., the minimum theoretical time to complete the DAG without any resource contention or queuing. This metric reflects how efficiently a scheduling algorithm can handle DAG execution relative to its inherent structural constraint.

In Figure 7, we compare the latency and cost of four RL algorithms, DDPG, DDRL, DQN, and PPO, across different numbers of UEs. Here, "Cost" represents the total cost for all tasks in a single episode, while "Latency" is normalization and refers to the time limited to complete one episode. The results presented in Figure 7 are obtained after training each policy for 300 episodes. This choice aligns with the convergence patterns observed in Figure 6, where all algorithms reach performance stability beyond 250 episodes. Using episode 300 ensures a fair and consistent evaluation of the learned scheduling policies under steady-state behavior. From the figure we can see that the PPO algorithm consistently demonstrates a clear advantage in terms of latency as the number of UEs increases. For example, while the latency for other algorithms like DDPG and DDRL rises significantly and over time constraint (episode time) as the number of UEs grows, PPO maintains stable performance, keeping latency below 1 episode time across most scenarios. This low latency is critical in applications where real-time processing is essential. Additionally, PPO also shows moderate cost efficiency. Although its cost tends to increase with the number of UEs, particularly at higher scales (e.g., 3200 for 10 UEs), it still remains competitive. This is because our proposed PPO algorithm takes into account the competition between different UEs. To ensure that all UEs complete their tasks within the time cycle, the PPO algorithm assigns more tasks to be executed on the cloud server, which results in higher costs compared to other algorithms. This balance between latency and cost makes PPO an attractive choice for scenarios where both real-time performance and



(a) (b) (c)

Fig. 8. Comparative analysis of results between greedy and improved PPO.

cost are key factors. Overall, PPO stands out as the most robust and balanced option for handling varying numbers of UEs, offering lower latency while maintaining reasonable costs compared to other algorithms.

To more intuitively compare the performance advantages of the proposed algorithm with the baseline algorithms, this study uses the optimization ratio ρ as a measure. In this experiment, only the optimization rate relative to the greedy algorithm is considered. We note that other baseline algorithms, such as DDPG and DDRL, are not included in Figure 8, due to their poor convergence and unstable performance under large-scale DAG workloads. Including these methods would not provide meaningful comparison and may obscure the trends of well-performing algorithms. The formula for calculating the optimization ratio is as follows:

$$\rho = 1 - \frac{Cost_{PPO}}{Cost_{greedy}},$$

where $Cost_{PPO}$ and $Cost_{greedy}$ represent the total cost of PPO task offloading approach and that of greedy task offloading strategy, respectively.

Figure 8 (a) illustrates the comparison of offloading computation costs among the greedy algorithm, DQN, and our proposed PPO task offloading algorithm as the cycle shrinkage rate ξ varies from 0.15 to 0.95 in increments of 0.05. The results show that all algorithms experience a decrease in offloading computation costs with fluctuations as the cycle shrinkage rate changes. This behavior is mainly influenced by the adjustment of the cycle shrinkage rate, which affects the optimization duration (episode time) and consequently changes the total number of tasks processed by the terminal devices within that duration. Moreover, the figure clearly shows that the PPO task offloading algorithm proposed in this study consistently exhibits lower offloading costs compared to the greedy algorithm and DQN offloading across different cycle shrinkage rates. This is supported by the optimization ratio ρ consistently surpassing 0, further confirming the efficiency of the proposed algorithm. Additionally, the optimization ratio decreases as the cycle shrinkage rate increases. This is because, to meet latency constraints and ensure real-time requirements for tasks, more tasks need to be offloaded. Consequently, there is a gradual increase in offloading computation costs, resulting in a reduction of the optimization ratio.

Figure 8 (b) illustrates the performance of the greedy algorithm, DQN, and PPO task offloading algorithms in terms of offloading computation costs as the number of UEs increases from 3 to 16. However, it is worth noting that as the number of UEs increases, the optimization ratio ρ gradually decreases. This phenomenon occurs because with the increase in the number of UEs, there is also an increase in reliance on and computational demands for the remote servers. Consequently, offloading computation costs rise, leading to a decrease in the optimization ratio.

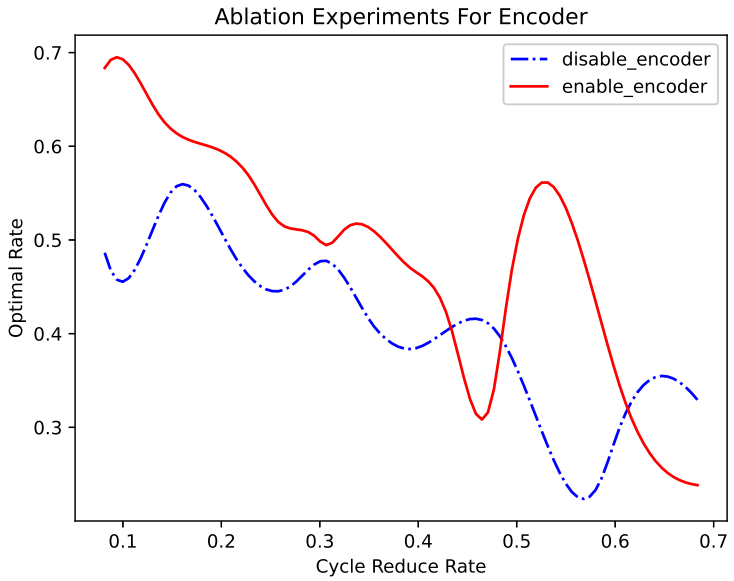


Fig. 9. The impact of Encoder on PPO at different cycle reduce rate.

Figure 8 (c) shows the changes in offloading computation costs for the greedy, DQN, and our proposed PPO algorithms as the transmission bandwidth increases from 5 to 30 Mbps in increments of 5 Mbps. Additionally, the cost reduction is more significant for the improved PPO algorithm. Consequently, the optimization ratio slightly increases with the improvement in transmission bandwidth.

5.4 The Impact of Encoder

To evaluate the specific impact of Encoder on the PPO algorithm's performance, we compared the performance changes of the PPO algorithm before and after integrating the Encoder. In this set of experiments, there were six UEs, and the network transmission bandwidth was 20 Mbps. The experimental results are shown in Figure 9. In the figure, "disable encoder" and "enable encoder" respectively represent models without and with the integrated Encoder. From the figure, it is evident that the model integrated with the task graph encoder consistently showed higher optimization rates, denoted by ρ, under various cycle reduction conditions compared to the model without integration of the Encoder. This is primarily attributed to the task graph encoder's capability to effectively integrate global node and structural information of tasks, enabling the PPO algorithm to approach task allocation strategies from a broader perspective, resulting in lower-cost task assignments.

5.5 Ablation Study on Behavior Cloning

To investigate the contribution of the BC module in our PPO-based offloading framework, we conduct an ablation study by comparing the performance of the full method with a variant that excludes the BC component. We design the ablation under the same MEC environment as described in Section 5.1, using five UEs and heterogeneous periodic DAG applications. We compare two versions: PPO (with BC): the complete model incorporating both the encoder and BC; PPO w/o

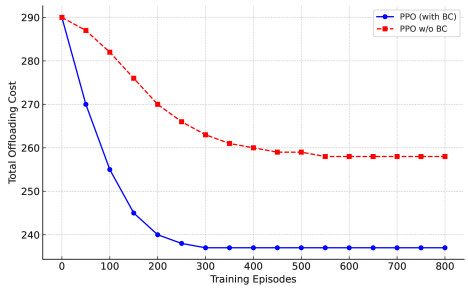


Fig. 10. Convergence comparison with and without BC.

Table 5. Comparison of Task Offloading Performance with and without BC

Method	Total Offloading Cost Cost	Avg Task Completion Time (unit)	Avg UE Energy (mJ)
PPO (with BC)	**237.4**	**11.6**	**42.8**
PPO w/o BC	259.7	13.1	47.9

BC: a reduced version where the PPO agent is trained from scratch without BC. Figure 10 shows the convergence trends of both methods across 800 training episodes. PPO with BC converges within approximately 400 episodes, while PPO w/o BC requires over 700 episodes to reach a similar cost level. This validates that BC effectively guides the agent toward a good initial policy, reducing exploration inefficiency.

This highlights the critical role of BC in reducing the search space during early training and accelerating convergence, especially under high-dimensional and multi-agent MEC environments. The quantitative results are summarized in Table 5. The results confirm that BC provides a good warm start policy to the PPO agent, guiding it away from inefficient random exploration and enabling better early-stage decisions. This improvement in training efficiency is particularly beneficial in large-scale multi-user MEC systems with complex task structures.

5.6 Scalability Evaluation on Larger-Scale MEC Settings

To further evaluate the scalability of our approach, we conducted additional experiments with the number of UEs ranging from 5 to 50. Each UE generates a periodic DAG-based application with medium computational load. The MEC infrastructure includes 10 edge servers and 3 cloud servers. We compare our method (PPO) with two baselines: the greedy algorithm and the GA+PSO hybrid method.

Table 6 shows the total offloading cost and the average cost per UE as the number of users increases. While the overall system cost naturally grows with more users, the average cost per UE with PPO remains relatively stable and lower than heuristic methods when the number of users exceeds 20. This suggests that our model learns to distribute tasks more effectively under resource contention. Our PPO-based framework is inherently distributed and can be parallelized across agents representing different UEs. Moreover, once trained, the PPO agent's inference step is lightweight and suitable for online decision-making. For MEC systems with hundreds of users, we envision deploying the proposed method in a hierarchical manner, where edge domains manage subgroups of users with localized PPO models. Future work will explore federated RL and cluster-based training to further enhance large-scale deployability.

Table 6. Total and Average Offloading Cost under Different Numbers of UEs

UE Count	Method	Total Cost	Avg Cost per UE
10	PPO	489.2	48.9
20	PPO	957.3	47.9
30	PPO	1442.1	48.1
40	PPO	1967.8	49.2
50	PPO	2520.3	50.4
50	Greedy	2782.6	55.6
50	GA+PSO	2643.1	52.9

6 Conclusion and Further Work

This article focuses on addressing challenges related to optimizing periodic DAG-structured tasks for multiple UEs in MEC environments. The study aims to minimize costs associated with UE energy consumption and server computation fees across multiple UEs by utilizing the PPO task offloading algorithm. To streamline the search process for numerous UEs, a BC algorithm is integrated into PPO, improving overall efficiency by initially cloning a greedy strategy. Additionally, an encoder is introduced to transform and reduce the dimensionality of high-dimensional task statuses for multiple users. Experimental results demonstrate the effectiveness of this algorithm in optimizing joint periodic task offloading for multiple UEs in MEC environments, highlighting improvements in task management efficiency.

As part of future work, we plan to extend our proposed weighted cost model to consider other aspects such as dynamic changes in transmission power, monetary cost, and the total system cost. Moreover, we will consider the bandwidth placement strategy and battery constraints of local UEs when extending our proposed PPO model.

References

[1] Laha Ale, Scott A. King, Ning Zhang, Abdul Rahman Sattar, and Janahan Skandaraniyam. 2022. D3PG: Dirichlet DDPG for task partitioning and offloading with constrained hybrid action space in mobile-edge computing. *IEEE Internet of Things Journal* 9, 19 (2022), 19260–19272.

[2] Laha Ale, Ning Zhang, Xiaojie Fang, Xianfu Chen, and Longzhuang Li. 2021. Delay-aware and energy-efficient computation offloading in mobile edge computing using deep reinforcement learning. *IEEE Transactions on Cognitive Communications and Networking* 7, 3 (2021), 881–892.

[3] Yuwei Bian, Yang Sun, Mengdi Zhai, Wenjun Wu, Zhuwei Wang, and Junjie Zeng. 2023. Dependency-aware task scheduling and offloading scheme based on graph neural network for MEC-assisted network. In *Proceedings of the 2023 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*. 1–6.

[4] Weiwei Chen, Dong Wang, and Keqin Li. 2019. Multi-user multi-task computation offloading in green mobile edge cloud computing. *IEEE Transactions on Services Computing* 12, 5 (2019), 726–738.

[5] Xiangchun Chen, Jiannong Cao, Yuvraj Sahni, Shan Jiang, and Zhixuan Liang. 2024. Dynamic task offloading in edge computing based on dependency-aware reinforcement learning. *IEEE Transactions on Cloud Computing* 12, 2 (2024), 594–608.

[6] Ziya Chen, Qian Ma, Lin Gao, and Xu Chen. 2024. Price competition in multi-server edge computing networks under SAA and SIQ models. *IEEE Transactions on Mobile Computing* 23, 1 (2024), 754–768.

[7] Yuya Cui, Degan Zhang, Ting Zhang, Peng Yang, and Haoli Zhu. 2021. A new approach on task offloading scheduling for application of mobile edge computing. In *Proceedings of the 2021 IEEE Wireless Communications and Networking Conference (WCNC)*. 1–6.

[8] Xiaoheng Deng, Jian Yin, Peiyuan Guan, Neal N. Xiong, Lan Zhang, and Shahid Mumtaz. 2023. Intelligent delay-aware partial computing task offloading for multiuser industrial internet of things through edge computing. *IEEE Internet of Things Journal* 10, 4 (2023), 2954–2966.

[9] Thinh Quang Dinh, Jianhua Tang, Quang Duy La, and Tony Q. S. Quek. 2017. Offloading in mobile edge computing: Task allocation and computational frequency scaling. *IEEE Transactions on Communications* 65, 8 (2017), 3571–3584.

[10] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. 2023. A distributed deep reinforcement learning technique for application placement in edge and fog computing environments. *IEEE Transactions on Mobile Computing* 22, 5 (2023), 2491–2505.

[11] Qiangqiang Jiang, Xu Xin, Libo Yao, and Bo Chen. 2024. METSM: Multiobjective energy-efficient task scheduling model for an edge heterogeneous multiprocessor system. *Future Generations Computer Systems: FGCS* 152 (2024), 207–223.

[12] Qinting Jiang, Xiaolong Xu, Qiang He, Xuyun Zhang, Fei Dai, Lianyong Qi, and Wanchun Dou. 2021. Game theory-based task offloading and resource allocation for vehicular networks in edge-cloud computing. *2021 IEEE International Conference on Web Services (ICWS)*. Chicago, IL, USA, 2021, 341–346.

[13] JoiloSlaana and DnGyrgy. 2020. Computation offloading scheduling for periodic tasks in mobile edge computing. *IEEE/ACM Transactions on Networking* 28, 2 (2020), 667–680.

[14] Xiang Ju, Shengchao Su, Chaojie Xu, and Haoxuan Wang. 2023. Computation offloading and tasks scheduling for the internet of vehicles in edge computing: A deep reinforcement learning-based pointer network approach. *Computer Networks* 223 (2023), 109572.

[15] Xiangjie Kong, Yuhan Wu, Hui Wang, and Feng Xia. 2022. Edge computing for internet of everything: A survey. *IEEE Internet of Things Journal* 9, 23 (2022), 23472–23485.

[16] Keqin Li. 2022. Distributed and individualized computation offloading optimization in a fog computing environment. *Journal of Parallel and Distributed Computing* 159 (2022), 24–34.

[17] Qing Li, Shangguang Wang, Ao Zhou, Xiao Ma, Fangchun Yang, and Alex X. Liu. 2020. QoS driven task offloading with statistical guarantee in mobile edge computing. *IEEE Transactions on Mobile Computing* 21, 1 (2020), 278–290.

[18] Shaoran Li, Chengzhang Li, Yan Huang, Brian A. Jalaian, Y. Thomas Hou, and Wenjing Lou. 2021. Task offloading with uncertain processing cycles. In *Proceedings of the 22nd International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing* .

[19] Wang Li, Xin Chen, Libo Jiao, and Yijie Wang. 2023. Deep reinforcement learning-based intelligent task offloading and dynamic resource allocation in 6G smart city. In *Proceedings of the 2023 IEEE Symposium on Computers and Communications (ISCC)* . 575–581.

[20] Tong Liu, Yameng Zhang, Yanmin Zhu, Weiqin Tong, and Yuanyuan Yang. 2021. Online computation offloading and resource scheduling in mobile-edge computing. *IEEE Internet of Things Journal* 8, 8 (2021), 6649–6664.

[21] T. D. Nguyen, X. Q. Pham, and V. Nguyen. 2021. Joint service caching and task offloading in multi-access edge computing: A QoE-based utility optimization approach. *IEEE Communications Letters* 25, 3 (2021), 965–969.

[22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *CoRR* abs/1707.06347 (2017).

[23] Om Kolsoom Shahryari, Hossein Pedram, Vahid Khajehvand, and Mehdi Dehghan Takhtfooladi. 2021. Energy and task completion time trade-off for task offloading in fog-enabled IoT networks. *Pervasive and Mobile Computing* 74, 2021 (2021), 101395.

[24] Zhengyu Song, Yuanwei Liu, and Xin Sun. 2021. Joint task offloading and resource allocation for NOMA-enabled multi-access mobile edge computing. *IEEE Transactions on Communications* 69, 3 (2021), 1548–1564.

[25] Tuyen X. Tran and Dario Pompili. 2019. Joint task offloading and resource allocation for multi-server mobile-edge computing networks. *IEEE Transactions on Vehicular Technology* 68, 1 (2019), 856–868.

[26] Jiayin Wang, Yafeng Wang, Peng Cheng, Kan Yu, and Wei Xiang. 2023. DDPG-based joint resource management for latency minimization in NOMA-MEC networks. *IEEE Communications Letters* 27, 7 (2023), 1814–1818.

[27] Zhonglun Wang, Peifeng Li, Shuai Shen, and Kun Yang. 2021. Task offloading scheduling in mobile edge computing networks. *Procedia Computer Science* 184 (2021), 322–329.

[28] Zhiyuan Wang and Qi Zhu. 2020. Partial task offloading strategy based on deep reinforcement learning. In *Proceedings of the 2020 IEEE 6th International Conference on Computer and Communications (ICCC)*. 1516–1521.

[29] Yalan Wu, Jigang Wu, Long Chen, Bosheng Liu, Mianyang Yao, and Siew Kei Lam. 2024. Share-aware joint model deployment and task offloading for multi-task inference. *IEEE Transactions on Intelligent Transportation Systems* 25, 6 (2024), 5674–5687.

[30] Zeinab Zabihi, Amir Masoud Eftekhari Moghadam, and Mohammad Hossein Rezvani. 2023. Reinforcement learning methods for computation offloading: A systematic review. 56, 1, Article No. 17 (2023), 1–41.

[31] Zhiwei Zhang, Zehan Chen, Yulong Shen, Xuewen Dong, and Ning Xi. 2024. A dynamic task offloading scheme based on location forecasting for mobile intelligent vehicles. *IEEE Transactions on Vehicular Technology* 73, 6 (2024), 7532–7546.

Received 2 November 2024; revised 3 July 2025; accepted 7 August 2025