Client-Master Multiagent Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing

TESFAY ZEMUY GEBREKIDAN, SEBASTIAN STEIN, and TIMOTHY NORMAN, University of Southampton, UK

As mobile applications grow in complexity, there is an increasing need to perform computationally intensive tasks. However, user devices (UDs), such as tablets and smartphones, have limited capacity to carry out the required computations. Task offloading in mobile edge computing (MEC) is a strategy that meets this demand by distributing tasks between UDs and servers. Deep reinforcement learning (DRL) is a promising solution for this strategy because it can adapt to dynamic changes and minimize online computational complexity. However, various types of continuous and discrete resource constraints on UDs and MEC servers pose challenges to the design of an efficient DRL algorithm. Existing DRL-based task-offloading algorithms focus on the constraints of the UDs, assuming the availability of enough resources on the server. Moreover, existing Multiagent DRL (MADRL)-based task-offloading algorithms are homogeneous agents and consider homogeneous constraints as a penalty in their reward function. We propose a novel Client-Master MADRL (CMMADRL) algorithm for task offloading in MEC that uses client agents at the UDs to decide on their resource requirements and a master agent at the server to make a combinatorial action selection based on the decision of the UDs. CMMADRL is shown to achieve up to 59% improvement in performance over existing benchmark and heuristic algorithms.

CCS Concepts: • **Computing methodologies** → **Multi-agent systems**; *Cooperation and coordination*; **Multi-agent reinforcement learning**; **Learning from critiques**.

Additional Key Words and Phrases: Multiagent Deep Reinforcement Learning, Combinatorial Action Selection, Mixed Constraints, Client-Master Multiagent Deep Reinforcement Learning, Distributed Solution

ACM Reference Format:
Tesfay Zemuy Gebrekidan, Sebastian Stein, and Timothy Norman. 2025. Client-Master Multiagent Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing. *ACM Trans. Autonom. Adapt. Syst.* ?, ?, Article ? (September 2025), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Recently, there has been an explosion of mobile applications that perform computation-intensive tasks, such as video streaming, virtual reality, augmented reality, image processing, video processing, face recognition, and online gaming [1, 2, 9, 26]. However, UDs, such as tablets and smartphones, have limited capability to perform the computational tasks of these applications. To address this limitation, Mobile Cloud Computing (MCC) has emerged as a key solution by enabling task offloading to cloud servers [13]. Nevertheless, one of the key drawbacks of MCC is latency caused by the distance of the MCC server from the UDs [17]. MEC has emerged as a promising technology for addressing the challenges of MCC and the increasing computing demands of UDs by providing MCC services on the edge of the network.

Authors' Contact Information: Tesfay Zemuy Gebrekidan, tzemuy13@gmail.com; Sebastian Stein, ss2@ecs.soton.ac.uk; Timothy Norman, T.J.Norman@soton.ac.uk, University of Southampton, Southampton, England, UK.

ACM 1556-4703/2025/9-ART?
https://doi.org/XXXXXXX.XXXXXXX

Task offloading in MEC has become an attractive solution for meeting the diverse computing needs of UDs [7], by distributing computational tasks between UDs and MEC servers. Many existing task-offloading algorithms use traditional convex optimization methods for single-agent offloading scenarios [16]. DRL is a common solution for task offloading problems due to its advantage in reducing online computational complexity [15] and adapting to dynamic changes [11]. However, the existence of various types of resource constraints on UDs and MEC servers and the combination of discrete, continuous, and combinatorial action spaces pose challenges to the design of an efficient DRL-based task-offloading strategy. UDs have limitations such as finite battery life and limited computational capabilities [9, 24], as well as quality of service (QoS) requirements, such as latency. Similarly, MEC servers come with storage constraints. DRL techniques, such as deep Q network (DQN), have yielded encouraging results by modeling the task-offloading problem as MDP using deep neural network (DNN) as a function approximator [11]. However, due to the curse of dimensionality, DQN is insufficient for learning with large discrete action spaces [3] and a combination of continuous and discrete action spaces [25]. Although multiagent deep deterministic policy gradient (MADDPG) algorithms can handle continuous action spaces, the representation of discrete and continuous action spaces still poses a challenge [8, 25]. Furthermore, despite the advances of MADRL algorithms in task offloading, such as cooperative offloading decisions [14] and mixed continuous and discrete action spaces [8, 25], they formulate the resource constraints as penalties in the reward function. However, hard constraints, such as the server's storage capacity, cannot be effectively captured by a penalty term.

A comprehensive survey on task offloading in [7] has presented task offloading strategies in MEC from different perspectives, including the computational model, the decision-making entity, and the algorithm paradigm. Many algorithms have considered the wireless communication resource and the computing resource of the server. For example, the insufficient computing resource of the MEC server can be alleviated by using MEC-MCC collaboration or collaboration among multiple MEC servers [2]. Sub-channels are considered in the state and action spaces by some DRL-based task offloading algorithms [8, 14]. However, the storage constraint on the server is overlooked in existing DRL-based task-offloading algorithms. This work aims to advance DRL-based task offloading algorithms to address the limitations of current algorithms as follows.

The main contributions of this work are fourfold.

- We propose a novel CMMADRL algorithm for task offloading in MEC with various types of constraints at the UDs, the wireless network, and the server. Client agents are deployed at the UDs to decide on their resource allocation, and a master agent is deployed at the server to make combinatorial action selection based on the actions of the clients. The constraints of the UDs are considered as a penalty in the reward of the client agents, whereas the channel and storage constraints are considered in the combinatorial action selection of the master agent.
- By avoiding the number of sub-channels from the state and action spaces, and considering it as a constraint in the combinatorial action selection, we reduced the dimensionality.
- This is the first DRL-based task offloading algorithm to consider combinations of continuous and discrete resource constraints on the UDs, the communication channel, and the storage capacity of the server.
- We develop different heuristic benchmarking methodologies and perform numerical analysis to determine the efficacy of the proposed algorithm, demonstrating up to a 59% improvement.

2 System Model

This section considers MEC for task offloading, which mainly includes a base station (BS), UDs, tasks, energy harvesting, and wireless networks. We consider a multi-user MEC scenario shown in Figure 1. In this scenario, there is a single wireless BS equipped with an MEC server that provides a computing and storage service as well as a software-defined network (SDN) controller that controls communication between UDs and BS. The BS serves a set of $N = \{1, 2, 3, \ldots, |\mathrm{N}|\}$ UDs. A UD in the set N is indicated by n. For local processing, we consider that each UD n has a minimum and maximum computational resource allocation budget denoted by f_n^{min} and f_n^{max}, respectively, in Gigahertz (GHz). Similarly, to offload its task to the server, we consider that each UD has a minimum and maximum transmission power allocation threshold denoted by p_n^{min} and p_n^{max}, respectively, in dBm. Furthermore, we consider a UD to have a minimum battery threshold, denoted by b_n^{min}, which triggers a low battery warning, and a maximum battery capacity of b_n^{max} in Megajoules (MJ). The BS has multiple constraints and characteristics, such as the server storage constraint z_e in bits and the number of processing units on the server U_e each having an equal processing capacity of f_e in GHz. Similar to [8, 14], we consider a wireless network of bandwidth of W in megahertz that is equally divided between K sub-channels. The list of notation and terms used in this work is presented in Table 1.

We consider a task-offloading problem over T time steps, each of duration τ_{max}. To focus on the main contributions, the following simplifying assumptions are made: each UD n generates one task per time step. If a task is not completed within τ_{max}, it is discarded before the start of the next time step. The task model, the processing model, and the energy harvesting are described in the following sections.

The task offloading model is designed by combining the settings of different existing approaches in the literature. Since [14] has used a data set from Huawei Technologies, we adapted it while excluding the blockchain-related components. We considered the energy harvesting process in [25], and took advantage of the efficient estimation of the completion times of tasks in [23], which computes the completion time of tasks on the server based on the completion time of other tasks scheduled before them. Similarly, we assume that the server processes the tasks in the order of their arrival on the server. The arrival times of the tasks are determined by their offloading times. In the following, we present the task and computing model for the task offloading problem.

2.1 Task Model

The task model is based on the settings in [14] on a data set from Huawei Telecom. At each time step, each UD n generates a task denoted by its notation as n[1] that is represented by characteristics such as the size of the task z_n in bytes, the number of CPU cycles per bit required to process the task c_n, and the maximum deadline τ_n by which task processing is expected to finish.

Before presenting how tasks are processed, we discuss three decision variables: a binary decision of whether to process a task locally or offload it to the MEC server, x_n, a local resource allocation, f_n, and a transmission power allocation, p_n, which are described in detail in Section 4.3. Thus, we define a binary decision variable $X = \{x_n | n \in N\}$ to describe the processing mode, as seen in Equation (1).

$$x_n = \begin{cases} 1, & \text{MEC processing} \\ 0, & \text{Local processing} \end{cases} \tag{1}$$

The decision variable, x_n, is determined by the joint decision of the UDs and the server, as will be formally defined in Equation (18) in Section 4.3. Next, we present the local and MEC models.

[1]Because a UD has one task at a time step, we use n to denote both the UD and its task to keep the notation simple.

Table 1. List of notations and terms

Notation	Description
N	Set of UDs
T	Number of time steps
τ_{max}	Length of a time step
n	A UD or a task of the UD
T_n	Total latency of processing task n
x_n	Binary indicator of local or offload for task n
E_n	Energy consumption of task n
$E_{\text{loc},n}$	Energy consumption of task n in local processing
$E_{\text{off},n}$	Energy consumption of offloading task n
$T_{\text{loc},n}$	Computation time for the local processing of task n
$T_{\text{off},n}$	Offloading time of task n
λ_1 & λ_2	weight coefficients of T_n and E_n
b_n	Battery level of UD n in Joules
b_n^{max}	Maximum battery level of UD n in Joules
b_n^{min}	Minimum battery level of UD n in Joules
p_n	Transmission power allocation of the UD n
z_n	Size of task n
z_e	Storage capacity of the server
U_e	Number of processing units in the server
L_n	Cost of processing task n
R	System reward of processing tasks
f_n	Resource allocation for local processing of task n
C_n	Number of CPU cycles to process one bit of task n
α_ϕ	Learning rate of the master agent
α_θ	Learning rate of client agent
J	Joules
MJ	Mega Joules
e_n	Energy harvesting of task n
A_m	Combinatorial action selection of the master agent
A_n	The tuple of actions of the client agent
$x_{c,n}$	Continuous-valued action of client n to decide local or offload
$p_{c,n}$	Continuous-valued action of client n to decide p_n
$f_{c,n}$	Continuous-valued action of client n to decide f_n
$T_{\text{ear},n}$	Availability time of a u_e after arrival of task n
$T_{\text{ser},n}$	Processing time of task n in the server
$T_{\text{MEC},n}$	Total latency of offloading and processing task n
S_n^{task}	State of task n
S_n^{gain}	State of channel gain of UD n
S_n^{pow}	State of power UD n
S_n^{res}	State of resource of UD n
S_n^{batt}	State of battery of UD n

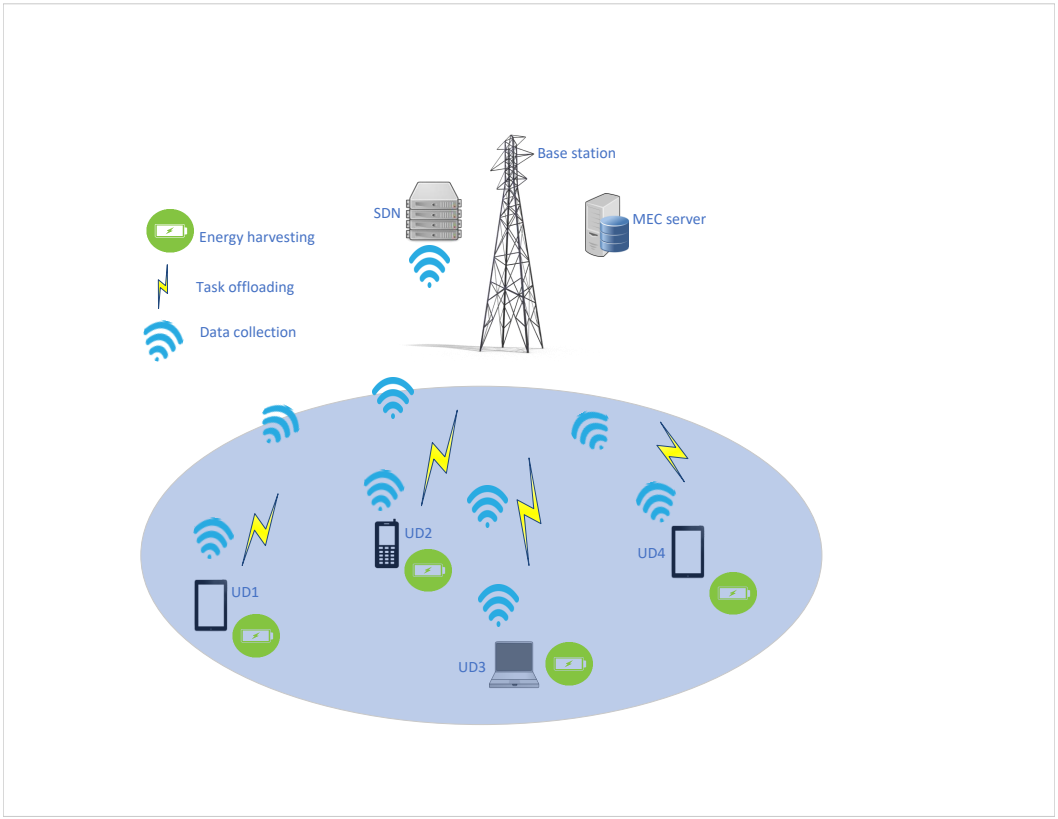

Fig. 1. Network model

2.2 Local Processing

A task is processed locally in either of the following two conditions as discussed in Section 4.3: if the UD decides to process the task locally; if a UD proposes the task to the master agent, but the master agent did not select it to be offloaded in the combinatorial action selection. Then, the UD processes the task using its local computational resource assigned to the task, which is restricted within its own resource allocation budget as $f_n | f_n^{min} \leq f_n \leq f_n^{max}$. The local computing latency to process the task is calculated as:

$$T_{\text{loc},n} = \frac{z_n \cdot c_n}{f_n} \tag{2}$$

where z_n is the size of the task and c_n is CPU cycles required to process one bit of the task.

The energy consumption in the local processing mode is calculated based on the size of the task and the allocation of resources to process the task, as shown in Equation (3).

$$E_{\text{loc},n} = \kappa \cdot z_n \cdot c_n \cdot (f_n)^2 \tag{3}$$

where κ is the energy consumption coefficient [25].

2.3 MEC Processing

In this mode, the task is transferred to the MEC server to be processed by one of the processing units U_e of the server. The decision happens when the UD proposes the task to be offloaded and

the master agent accepts it. To be processed on the server, the task needs transmission resources, which are a function of transmission power p_n. The transmission power p_n is decided by the UD from its transmission power budget $p_n | p_n^{min} \leq p_n \leq p_n^{max}$ as discussed in Section 4.3. Then, the data transmission rate d_n for a single channel of the wireless network is calculated using Shannon's capacity as

$$d_n = \frac{W}{K} \cdot \log_2 (1 + p_n \cdot g_n) \tag{4}$$

where $g_n = h_n/\sigma^2$ is the normalized channel gain of the uplink channel between UD n and the BS, with channel gain h_n and the background noise variance σ^2. The channel gain h_n is impacted by many factors, including distance. For simplicity, we assume that the UDs are stationary and have a stationary normalized channel gain depending on their distance from the BS. The background noise variance, σ^2, is assumed to be constant. We did not consider interference between multiple UDs because we assume that a channel is used by one task at a time.

Once the data transmission rate is determined, the transmission time $T_{\text{off},n}$ is computed as:

$$T_{\text{off},n} = \frac{z_n}{d_n} \tag{5}$$

where z_n is the size of the task and d_n is the data transmission rate.

Then, using the transmission time $T_{\text{off},n}$ and the power p_n of the UD, the energy consumption of offloading task n to the server is calculated as:

$$E_{\text{off},n} = p_n \cdot T_{\text{off},n} \tag{6}$$

Like much other work on task offloading [14, 22, 25], we assume that the communication resource required to return the information about the processed task to the UD is negligible, as only analytical results are transmitted rather than the raw data itself.

Note that the energy consumption in task offloading is computed only for the UDs as they are battery-powered. However, the latency of processing the tasks on the server matters because the tasks have deadline constraints. Therefore, the total latency of processing a task on the server is determined by the transmission time, the earliest availability of the processing unit on the server, and the time required to process the task on the server. The processing time of task n in one of the processing units on the server is computed as $T_{\text{ser},n} = \frac{z_n \cdot c_n}{f_e}$. However, the processing of the task on the server does not start as soon as the task has arrived at the server. The processing units on the server process one task at a time. Tasks transferred to the server are processed in the order of arrival at the server, which is determined by $T_{\text{off},n}$. The tasks are assigned to the earliest free processing unit. Therefore, the start of processing task n depends on the earliest availability of a processing unit, which is determined by the number of processing units on the server U_e, and $T_{\text{ser},n}$ and $T_{\text{off},n}$ of other tasks that have shorter $T_{\text{off},n}$ than that of task n. Consequently, the total latency of the offloading task n to the MEC server $T_{\text{MEC},n}$ is calculated as:

$$T_{\text{MEC},n} = T_{\text{ser},n} + \max \left(T_{\text{off},n}, T_{\text{ear},n}\right) \tag{7}$$

where $T_{\text{ear},n}$ is the estimated availability time of the first available processing unit U_e of the server after the arrival of task n and max(.) ensures that the processing of the task starts when a free processing unit is found after the offloading of the task is finished. The $T_{\text{ear},n}$ is calculated based on the completion time of other accepted tasks on the server with the earliest offloading time than that of task n. This estimate is adapted from the work of [23].

2.4 Energy Harvesting

The energy harvesting process is adapted from the work in [25]. For simplicity, we assume that the UDs harvest e_n of energy in joules at the beginning of each time interval. This energy may come

from renewable sources such as solar or kinetic harvesting, depending on the device's capabilities and environmental context. Initially, each UD starts with its maximum battery capacity of b_n^{max} in joules. Over the course of T time steps, the battery level dynamically evolves based on both energy consumption and harvesting, ensuring realistic constraints on device operation. The evolution of the battery level is described by the following update equation:

$$b_n(\mathrm{t+1}) = \min\left(\max\left(b_n(\mathrm{t}) - E_n(\mathrm{t}) + e_n(\mathrm{t}), 0\right), b_n^{\max}\right) \tag{8}$$

Here, $b_n(t+1)$ represents the battery level of UD n in joules at time step $t+1$, and $b_n(t)$ is the battery level at the current time step. The term $E_n(t)$ denotes the energy consumed by the UD, which is computed as the sum of local computation energy $E_{\mathrm{loc},n}$ (Equation (3)) and offloading energy $E_{\mathrm{off},n}$ (Equation (6)). The harvested energy $e_n(t)$ is added to replenish the battery. The use of $\max(\cdot, 0)$ ensures that the battery level does not fall below zero, and the outer $\min(\cdot, b_n^{\max})$ enforces the upper bound, preventing overcharging. This battery model plays a crucial role in ensuring sustainable operation of energy-constrained UDs across the time horizon.

3 Problem Formulation

The cost of processing a task is collectively determined by its processing latency T_n and the energy consumption E_n, which are defined as follows.

$$T_n = (1 - x_n) \cdot T_{\mathrm{loc},n} + x_n \cdot T_{\mathrm{MEC},n} \tag{9}$$

where $T_{\mathrm{loc},n}$ and $T_{\mathrm{MEC},n}$ are computed using Equations (2 and 7) respectively. This formulation captures the trade-off in latency between local execution and MEC offloading, controlled by the binary decision variable x_n.

$$E_n = (1 - x_n) \cdot E_{\mathrm{loc},n} + x_n \cdot E_{\mathrm{off},n} \tag{10}$$

where $E_{\mathrm{loc},n}$ and $E_{\mathrm{off},n}$ are computed using Equations (3 and 6) respectively. Similarly, E_n represents the energy consumption trade-off between local computation energy (when $x_n = 0$) and task offloading transmission energy (when $x_n = 1$).

Then, the cost function is specified as:

$$L_n = \lambda_1 \cdot T_n + \lambda_2 \cdot E_n \tag{11}$$

where λ_1 and λ_2 are weighting coefficients used to compute the scalarized cost, corresponding to latency and energy consumption, respectively. This allows flexibility in adapting the optimization objective to different application priorities, such as delay-sensitive or energy-constrained environments.

The CMMADRL aims to solve the optimization problem that can be formulated as the cost minimization for all UDs and T time steps while meeting the different constraints at the UDs and the server as follows:

$$\underset{\{x_n, p_n, f_n\}}{\text{minimize}} \quad \sum_{t}^{T} \sum_{n \in N} L_n(\text{t}) \tag{12a}$$

$$\text{subject to} \quad x_n \in \{0, 1\}, \quad \forall n \in N \tag{12b}$$

$$p_n^{\min} \leq p_n \leq p_n^{\max}, \quad \forall n \in N \tag{12c}$$

$$T_n \leq \tau_n, \quad \forall n \in N \tag{12d}$$

$$b_n \geq b_n^{min}, \quad \forall n \in N \tag{12e}$$

$$f_n^{\min} \leq f_n \leq f_n^{\max}, \quad \forall n \in N \tag{12f}$$

$$\sum_{n \in N} x_n \leq K \tag{12g}$$

$$\sum_{n \in N} x_n \cdot z_n \leq z_e \tag{12h}$$

where the decision variables in Equation (12a) represent the actions taken by the client and master agents as outlined in Section 4.3, Equation (12b) implies that a task is processed locally if $x_n = 0$ or uploaded to the MEC server otherwise, Equation (12c) indicates that the transmission power should be between the power allocation budget, Equation (12d) ensures that the processing time of each task cannot exceed its processing deadline, Equation (12e) guarantees that the battery level should not fall below the minimum battery threshold, Equation (12f) ensures that the local computational resource allocated to each task should be in the preset minimum and maximum values, Equation (12g) ensures that the number of offloaded tasks does not exceed the number of sub-channels by ensuring that only one task uses a channel. It is used if and only if it is necessary to use only one channel for one user as used in [9]. Equation (12h) guarantees that the sum of the sizes of the off-loaded tasks does not exceed the storage capacity of the server.

4 Combinatorial Client-Master MADRL Algorithm for Task Offloading in MEC

To solve the optimization problem of the cost minimization in Equation (12a), we convert the optimization problem into a reward maximization problem and apply CMMADRL. The states, client and master actions, and the formulation of the reward function are presented as follows. First, MADDPG is introduced.

4.1 MADDPG

CMMADRL is derived by incorporating the coalition action selection [4] and the per-action DQN [6] on top of MADDPG. We start by introducing MADDPG. MADDPG is a multiagent extension of the Deep Deterministic Policy Gradient (DDPG) [10].

DDPG is a model-free DRL algorithm specifically designed for environments with continuous action spaces. It employs an actor-critic architecture consisting of two neural networks: the actor, which deterministically maps states to actions to maximize expected long-term rewards, and the critic, which estimates the Q-value (expected return) for given state-action pairs. Unlike stochastic policy methods, DDPG's actor outputs a specific action rather than a probability distribution, enabling deterministic policy learning. DDPG is an off-policy algorithm, meaning it utilizes a replay

buffer to store and sample past experiences. To ensure stable training, DDPG introduces target networks for both the actor and critic, which are updated slowly to reduce the risk of divergence.

MADDPG [12] is an extension of the DDPG algorithm tailored for multi-agent DRL environments, where the presence of multiple learning agents introduces the need for coordination. In MADDPG, multiple DDPG agents cooperate through a centralized training with decentralized execution framework. During training, each agent utilizes a centralized critic that has access to the observations and actions of all agents, allowing it to more effectively estimate the joint action-value function and adapt to the dynamic policies of other agents. In contrast, during execution, each agent relies solely on its decentralized actor, which makes decisions based on its own local observations, enabling scalable and independent operation.

CMMADRL employs client agents analogous to the actors in MADDPG. The primary distinction lies in the value function. MADDPG uses the critic network only when training. In CMMADRL, the master agent is utilized during both training and execution. It performs the same role as the MADDPG critic during training. But it also performs combinatorial action selection during execution. The difference in the Q-value computation for both CMMADRL and MADDPG are presented in Equation (21) and Equation (22), respectively.

4.2 State

The state $S(t)$ of the MEC environment at time t, which includes the set of states of the UDs, is described as $S(t) = \{S_n(t)\}, \quad \forall n \in N$. Constant values such as the number of sub-channels K, the number of processing units on the server U_e, the processing capacity f_e, and the storage capacity z_e of the server are excluded from the state information. The state of a UD, $S_n(t)$, is characterized by five components: task state $S_n^{\text{task}}(t)$, normalized channel gain state $S_n^{\text{gain}}(t)$, power transmission budget $S_n^{\text{pow}}(t)$, local resource allocation budget $S_n^{\text{res}}(t)$, and battery state $S_n^{\text{batt}}(t)$ as defined in Equation (13):

$$S_n(t) = \{S_n^{\text{task}}(t), S_n^{\text{gain}}(t), S_n^{\text{pow}}(t), S_n^{\text{res}}(t), S_n^{\text{batt}}(t)\} \tag{13}$$

where $S_n^{\text{task}}(t) = [z_n(t), c_n(t), \tau_n(t)]$, $S_n^{\text{gain}}(t)$ is $g_n = h_n/\sigma^2$ as described in Section 4, $S_n^{\text{pow}}(t) = p_n^{max}$, $S_n^{\text{res}}(t) = f_n^{max}$, and $S_n^{\text{batt}}(t)$ is as described in Equation (8).

4.3 Action

At the beginning of each time step, the UDs make decisions about their resource allocations using client agents. Then, the SDN controller collects information about the state and action of the UDs and performs one of the following three procedures using a master agent: 1) for the UDs that decide to make a local processing, the server does not interfere. 2) if the number of UDs that are proposed to offload is greater than the number of sub-channels or if the sum of their sizes is greater than the capacity of the storage capacity of the server, the server makes a combinatorial action selection on which of the requests of the UDs to approve and which of them to reject. 3) If the proposed requests are less than the constraints, the server accepts all of them. Finally, sub-channels are assigned to accepted UDs, and then the task offloading and processing process starts.

Existing DRL-based task offloading algorithms, such as [14] and [8], included the number of sub-channels in their state and action space. However, under the assumption of homogeneous sub-channel quality, each channel offers equal transmission capacity as seen in Equation (4). Therefore, if each channel is restricted to one UD at a time, the specific channel assignment becomes irrelevant. The inclusion of it in the state and action spaces adds unnecessary complexity. We excluded channel information from the state and action space and considered them as a constraint in the combinatorial action selection. Note that a channel can be reused by multiple UDs one after the other, by lifting

the constraint that one channel must be used by one UD. In such a case, only the storage capacity of the server becomes the constraint in the combinatorial action selection.

The actions of the client agents and master agent are as follows.

Client Actions: At each time step, each client agent produces three actions, which are all continuous value actions between [0, 1] inclusive. The action space can be expressed as:

$$A_n(t) = x_{c,n}(t), p_{c,n}(t), f_{c,n}(t) \leftarrow \theta_n(S_n(t)), \quad \forall n \in N \tag{14}$$

where $S_n(t)$ is the state of UD n as described in Equation (13) and θ_n is the parametrized policy function of the client agent. A_n is a tuple of three actions of the client agent as follows: $x_{c,n}(t)$ is the task offloading decision by client n (if $x_{c,n}(t) < 0.5$ then x_n in Equation (1) becomes 0 otherwise the task is proposed to be considered for the combinatorial action selection), $p_{c,n}(t)$ is the client action that decides the transmission power using Equation (15), and $f_{c,n}(t)$ is the action that decides the local computational resources allocation using Equation (16).

The actions of the client agents determine p_n and f_n as follows:

$$p_n = \max\left(p_n^{min}, p_{c,n}(t) \cdot p_n^{max}\right) \tag{15}$$

$$f_n = \max\left(f_n^{min}, f_{c,n}(t) \cdot f_n^{max}\right) \tag{16}$$

where $p_{c,n}(t)$ and $f_{c,n}(t)$ are the outputs of the client agent. Both are in the range of [0,1]. They are used to scale the maximum power and the computational power, respectively, to compute p_n and f_n. Then, p_n is used to calculate the data rate in d_n in Equation (4), which is used to transfer the task to the server (if the task is decided for offloading). The f_n is used to calculate the energy consumption to process the task locally $E_{\text{loc},n}$ in Equation (3) (if decided for local processing). The $max(.)$ ensures that the allocation of power and computational resources is above the minimum threshold.

Master Action: For the client actions with $x_{c,n}(t) \geq 0.5$, the master takes the combinations of states and actions of the clients and provides a binary output for the combinatorial action selection on which of them should be allocated locally and which of them should be accepted for processing by the MEC server.

$$A_m(t) = x_{m,n}(t) \leftarrow \phi(S, A, S_n, A_n), \quad \forall n \in \{n \in N \mid x_{c,n}(t) \geq 0.5\} \tag{17}$$

where S and A are the set of states and actions of all client agents, and S_n and A_n are the set of states and actions of the client agents whose $x_{c,n} \geq 0.5$ and ϕ is the policy of the master.

The final task offloading decision is determined by merging the local decisions of the client agents and the combinatorial action selection of the master agent to compute x_n of Equation (1) using Equation (18).

$$x_n = \begin{cases} A_m, & x_{c,n} \geq 0.5 \\ 0, & otherwise \end{cases} \tag{18}$$

where A_m is the master agents decision in Equation (17). This x_n is used to compute T_n, E_n and b_n, which are then used to calculate L_n in Equation (11) and L'_n in Equation (19).

The combinatorial action selection in the master agent is built by modifying the critic in the classical MADDPG algorithm and incorporating the per-action DQN [6].

Figure 2 shows the interaction diagram of the CMMADRL algorithm and the MEC system. Client agents represent the policies of the UDs. The master agent represents the policy on the MEC server. The environment represents the allocation of resources in the UDs and on the server. After a client agent produces its output, it does the following, as mentioned in Section 4.3: if $x_{c,n} < 0.5$ assigns $x_n = 0$ and starts the local allocation. Otherwise, it forwards $x_{c,n}, p_n, f_n$ to the master agent for the

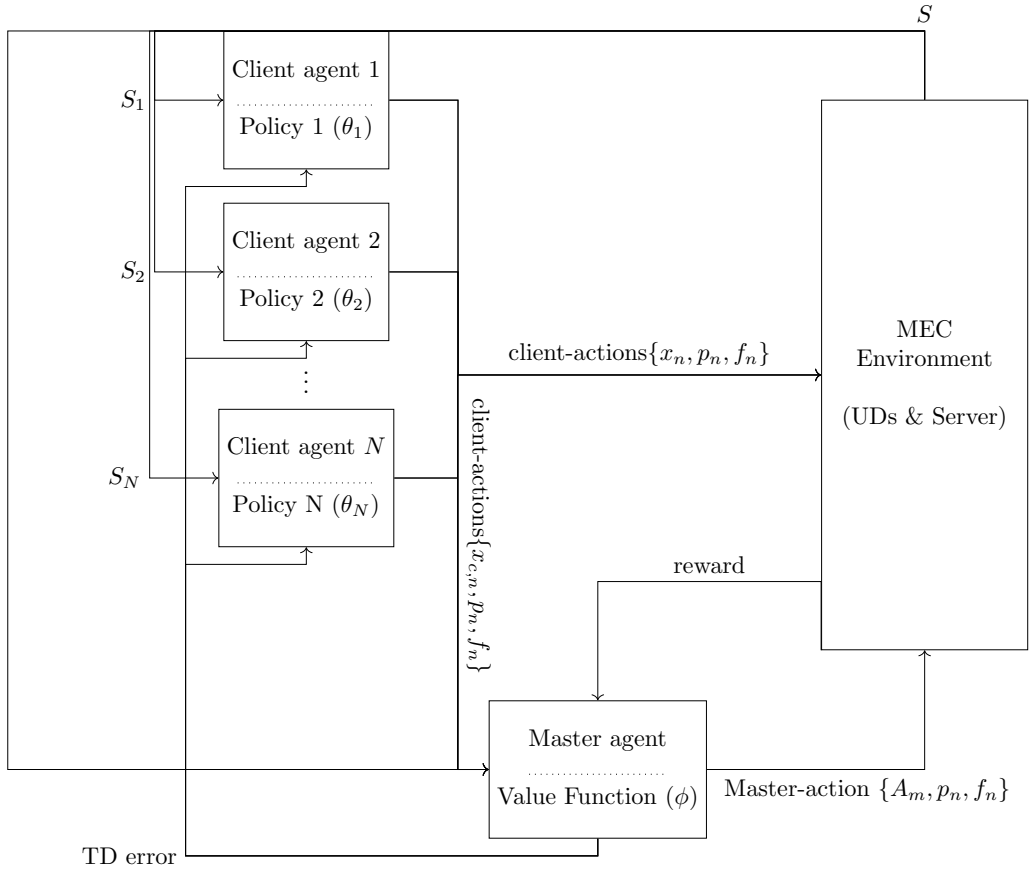

Fig. 2. Interaction diagram of the agents and the MEC environment. Each client agent agent outputs three actions $\{x_n, p_n, f_n\}$. Clients with $\{x_{c,n} < 0.5\}$ start local processing, while others propose their tasks to the master agent for combinatorial action selection. The master agent ranks the proposed tasks based on their Q-values and greedily selects tasks until channel and storage constraints are reached. Tasks not selected by the master agent are processed locally by the respective clients.

combinatorial action selection. Then, the master agent produces the binary decision and applies it to the UDs and the server. Finally, a shared reward is computed and provided to the master agent to train its value function. The client agents are also trained using a TD error computed by the master agent as feedback.

4.4 System Reward Function

To compute the reward, we use the negative of the objective function, and we compute a penalty function for exceeding the constraints of the task deadline and minimum battery levels. The server storage constraint is considered in the combinatorial action selection and does not need to be incorporated as a penalty. In [25], they included a dropoff penalty for the reward function if the UD runs out of battery. In ours, we impose a penalty when the battery drops below a preset threshold.

$$L'_n = \lambda_1 \cdot \min\left(\left(\tau_n - T_n\right), 0\right) \; + \lambda_2 \cdot \min\left(\left(b_n - b_n^{min}\right), 0\right) \tag{19}$$

Since the design of our cooperative learning formulation is based on the cost minimization problem in Equation (12a), our system reward function is equal to the negative of the system cost function and the penalty function. Thus, we can formulate the reward function of the system as follows.

$$R\left(S\left(t\right),A\left(t\right)\right)=-\frac{1}{|N|}\cdot\sum_{n\in N}\left(L_n\left(t\right)+L'_n\left(t\right)\right) \tag{20}$$

where $L_n\left(t\right)$ is the cost function of a single UD in a single time step as computed in Equation (11) and $L'_n\left(t\right)$ is the penalty function computed in Equation (19). Note that the reward is computed at every time step, but the cost minimization in Equation (12a) is computed for all time steps.

The power and resource allocation decisions made in the current step by the UDs affect their operational life in the next time steps by affecting energy consumption. Therefore, the DRL must blend immediate reward and long-term return using the soft update rule that incorporates Bellman target as shown in Equation (21)

$$\begin{aligned}Q\left(S,A,S_n,A_n|\phi\right)=\left(1-\alpha_\phi\right)\cdot Q\left(S,A\right)\ &+\alpha_\phi\cdot\left(R\left(S,A\right)\right.\\&\left.+\gamma\cdot\max_{S'_n,A'_n}Q'\left(S',A',S'_n,A'_n|\phi'\right)\right)\end{aligned} \tag{21}$$

where α_ϕ is the learning rate, γ is the discount factor, ϕ is the policy of the network, ϕ' is the policy of the target critic, $S=\{S_1,...,S_N\}$ and $S'=\{S'_1,...,S'_N\}$ are combined current and next states, $A=\{A_1,...,A_N\}$ and $A'=\{A_1,...,A'_N\}$ are combined current and next actions of the client agents, and S'_n and A'_n are the corresponding states and actions of the agents. The role of the four parameters in Q' is presented in the following section.

4.5 The Master Agent with Per-Client DQN

In MADDPG, there is only a single Q-value for the combined state and action pair of all actors, which is calculated as:

$$Q\left(S,A\right)=\left(1-\alpha_\phi\right)\cdot Q\left(S,A|\phi\right)\ +\alpha_\phi\cdot\left(R\left(S,A\right)\ +\gamma\cdot Q'\left(S',A'|\phi'\right)\right) \tag{22}$$

To customize the critic to select combinatorial actions, it should be able to provide a Q-value per each actor (or client in the case of CMMADRL). Therefore, the master agent with per-client DQN in the CMMADRL adapts the concept of per-action DQN where the state S_n and the action A_n of each client agent are appended to the combined state and the action of the client agents to calculate the relative Q value in the combination of the state and the action as seen in Equation (21). The combined rewards are given only to the selected clients in the task offloading problem, as seen in Algorithm 3 so that they will have different Q values to distinguish them in action selection. The reward is used by the master agent to train a value function for the selected clients using per-client DQN. Client agents are also trained using a TD error computed by the master agent as feedback.

The master agent applies the coalition action selection approach in [4]. However, for ease of benchmark comparison with MADDPG, it uses per-action DQN instead of a transformer neural network. This is due to the fact that the per-action DQN architecture more closely resembles the critic network used in MADDPG than it does a transformer-based neural network.

4.6 Algorithms

Since the master agent in the CMMADRL algorithm has two roles: providing feedback for training the clients, similar to the MADDPG, and participating in the combinatorial action selection of the clients, it follows different procedures for both. Therefore, the algorithm is presented in three parts: a main algorithm, an action selection algorithm, and a training algorithm in the following sections.

Main Algorithm: The main algorithm runs the action selection algorithm, the training environment, and the evaluation environment. At each episode, it runs for T time steps as seen in lines 7 to 12 of Algorithm 1. At each step, the action selection algorithm is called, and the rewards are computed, and then the experience is recorded to replay memory. When the iteration over the steps is complete, the training algorithm is called and the trained policies of the client agents and the master agent are evaluated.

Algorithm 1 CMMADRL main algorithm

1: Initialize $Max\_Episodes$ = 2000, $Min\_Epsilon$ = 0.01, $Max\_Epsilon$ = 1, γ = 0.99
2: Initialize client agents $\theta_n \quad \forall n \in N$ and the master agent ϕ with random weights
3: Initialize target client agents $\theta'_n \leftarrow \theta_n$ and the target master agent $\phi' \leftarrow \phi, \quad \forall n \in N$
4: Initialize replay memory RM
5: **for** episode = 1 to $Max\_Episodes$ **do**
6: Reset environment and get initial state $S_n(t=1), \quad \forall n \in N$
7: **for** t = 1 to T **do**
8: Go to Algorithm 2 using evaluation = False flag to select client and master actions
9: Execute actions and observe system reward $R(t)$ and next state $S_n(t+1), \quad \forall n \in N$
10: Store transition $(S_n(t), A_n(t), R(t), S_n(t+1)), \quad \forall n \in N$ into RM
11: Update the state $S_n(t) \leftarrow S_n(t+1), \quad \forall n \in N$
12: **end for**
13: Go to Algorithm 3 for training
14: **for** EvalEpisode in EvalEpisodes **do**
15: Reset and seed episode to EvalEpisode and find state $S(t=1)$
16: **for** t = 1 to T **do**
17: Go to Algorithm 2 using evaluation = True flag to select client and master actions
18: Execute actions and observe total reward $R(t)$ and next state $S_n(t+1), \quad \forall n \in N$
19: Update the state $S_n(t) \leftarrow S_n(t+1), \quad \forall n \in N$
20: **end for**
21: **end for**
22: **end for**

Action Selection Algorithm: Since the action selection algorithm requires exploration during training to balance the trade-off between discovering new strategies and exploiting known ones, we first present the computation of the decaying ϵ. The value of ϵ is used differently in the client and master agents. In the client agents, ϵ scales the random noise added to the output actions, as seen in line 6 of Algorithm 2. In contrast, the master agent employs ϵ-greedy exploration to decide whether to explore a new action or exploit the learned policy, as indicated in lines 13 through 48 of the algorithm.

Specifically, if a randomly generated value is greater than ϵ, the master agent explores by accepting the proposed tasks ($x_{c,n} \geq 0.5$) in a random order until the constraints are met (lines 45–47). Otherwise, the agent exploits by first computing the Q-values of the tasks (line 20) and then accepting them in descending order of their Q-values (lines 28–38). Note that when all tasks are within the constraints, they are accepted directly, as shown in lines 24–26. Additionally, during exploration (lines 45–46), tasks are accepted in a purely random order, unlike during exploitation, which involves sorting tasks by their Q-values (line 28).

This difference in exploration strategy stems from the nature of the action spaces: client agents operate in a continuous action space, while the master agent operates in a discrete action space. The value of ϵ is updated at the beginning of each episode according to Equation (23).

$$\epsilon = Min\_Epsilon + \ (Max\_Epsilon - Min\_Epsilon) \cdot e^{-\frac{episode}{Max\_Episodes}} \tag{23}$$

where $Min\_Epsilon$ and $Max\_Epsilon$ are the minimum and maximum values of the decaying epsilon, $episode$, is the current episode, and $Max\_Episodes$ is the maximum number of episodes.

The action selection approaches for the client agents and the master agent are presented separately as follows. Note that the evaluation flag is used to indicate whether the actions are running for the training environment or for the evaluation environment. In the evaluation mode, when Evaluation is True in line 4 and 13, no exploration is needed.

Action Selection in The Client Agents: For client agents, the exploration is performed by adding noise to the outputs of the client agents, as seen in lines 5 to 8 of the Algorithm 2. After adding noise to the actual action, the values are clipped to [-1,1] so that they are within the activation function of the client agents, Tanh in this case. If exploration is not flagged, the actual output is already within the range of [-1,1]. Client actions, whether explored or exploited, are scaled to be between [0,1] before applying to compute the allocation of resources in Section 4.3.

Action Selection in The Master Agent: The master agent follows ϵ-greedy for exploration and exploitation as seen in lines 41 to 47. First, a random number is generated as seen in line 11 to decide whether to explore or exploit. If the number is greater than ϵ, the master agent shuffles the proposed actions as seen in line 45 and follows the combinatorial action selection procedure described below. Otherwise, the master agent computes the Q value based on the states and actions of the proposed actions and appends the Q value along with the identifiers of the tasks n to Qs and $Index$ and follows the combinatorial action selection procedure.

Combinatorial Action Selection: After receiving the actions of the client agents, the master agent follows one of three procedures during exploitation (Section 4.3). If all clients choose local processing (line 18 of Algorithm 2), the master agent does not take any action. Otherwise, for clients proposing to offload tasks, the master computes their Q-values using per-client-DQN (line 20), and stores them in the lists Qs and $Index$ with their identifiers n.

If the number of offloaded tasks and their cumulative size are within the number of sub-channels and the storage limit of the server (lines 24–26), all tasks are accepted. Otherwise, the master agent ranks the proposed tasks by Q-value and approves them greedily until resource constraints are met. The remaining tasks are assigned back for local execution. This action selection strategy is detailed using a diagram in Figure 2.

In exploration mode, the same process is followed, except that the proposed actions are randomly shuffled instead of sorted by Q-values. Note that the exploration is only used to explore other actions during training. During evaluation of the algorithms, the actual actions are used as seen in line 15 to 21 of Algorithm 1.

Training Algorithm: The algorithm for training the client agents and the master agent is provided in Algorithm 3. Because the structure of the master agent is different from the MADDPG critic as seen in Section 4.5, Algorithm 3 is significantly different from existing MADDPG training algorithms in that: It generates multiple Q-values rather than one combined Q-value, because the master agent has to make a combinatorial action selection using the relative Q-values of the clients as seen in lines 7,17, and 33; The client agents are trained by computing the highest Q value from the tasks offloaded to the server as seen in lines 13 and 38; If all tasks are allocated locally by the

Algorithm 2 The action selection algorithm for the client agents and the master agent

1: Input: state S_n for each client agent n, $\{K, z_e\}$, and Evaluation flag
2: Output: client actions A_n for each n with x_n decided by collaboration with the master agent
3: Get action $A_n \leftarrow \pi_n(S_n, \theta_n), \quad \forall n \in N$
4: **if** Evaluation == False **then**
5: Compute ϵ using Equation (23)
6: $noise$ = random(|N| by |A_n|)*ϵ
7: $A_n = A_n + noise_n, \quad \forall n \in N$
8: Clip A_n to [-1, 1]
9: **end if**
10: Scale A_n to [0, 1] using $\frac{A_n}{2} + 0.5, \quad \forall n \in N$
11: Generate a $random$ number
12: Qs = [], $Index$ = []
13: **if** $random < \epsilon$ or Evaluation == True **then**
14: $S = \{S_n\}, A = \{A_n\}, \quad \forall n \in N$
15: **for** $n \in N$ **do**
16: Get $x_{c,n}$ from A_n as described in Section 4.3
17: **if** $x_{c,n} < 0.5$ **then**
18: $x_n = 0$
19: **else**
20: $Append(Qs, Q(S, A, S_n, A_n, \phi))$, $Append(Index, n)$
21: **end if**
22: **end for**
23: **if** $Length(Index) \leq K$ and $Sum\left(z_n \ \forall n \in N \ and \ x_{c,n} \geq 0.5\right) \leq z_e$ **then**
24: **for** $\forall n \in N \ and \ x_{c,n} \geq 0.5$ **do**
25: $x_n = 1$
26: **end for**
27: **else**
28: Sort Qs, and adjust $Index$ accordingly
29: $TotalSizeOfAccepted = 0$
30: **while** $Length(Index) > K$ **do**
31: $n = Pop(Index)$
32: $x_n = 1$
33: **if** $TotalSizeOfAccepted + z_n \leq z_e$ **then**
34: $TotalSizeOfAccepted = TotalSizeOfAccepted + z_n$
35: **else**
36: $x_n = 0$
37: **end if**
38: **end while**
39: **end if**
40: **else**
41: Collect the index n of the tasks with $x_{c,n} \geq 0.5 \ \forall n \in N$ to $Index$
42: **if** $Length(Index) \leq K$ and $Sum\left(z_n \ \forall n \in N \ and \ x_{c,n} \geq 0.5\right) \leq z_e$ **then**
43: Execute lines 24 to 26
44: **else**
45: shuffle $Index$ into random order
46: Execute lines 29 to 38
47: **end if**
48: **end if**

client agents, the master agent uses *all_zeros* as a placeholder to hold the combined Q-value as seen in lines 11, 22, and 35. Note that after the master agent decides which of the tasks should be processed in the server as described in Section 4.3, the reward is applied only to the selected tasks during training. That is, the reward is computed at a system level using Equation (20) but it is used only by the tasks offloaded to the server when training the algorithm so that the tasks are distinguished by their Q-values in the action selection algorithm. This technique is adapted from the coalition action selection in [4].

The notation of the client and master actions is changed in the training algorithm due to the subscript i for the minibatch which is used to iterate over the entries of the minibatch of size M. Unlike MADDPG, which computes the Q-values of the minibatch as a batch, the Q-values of the minibatch in the CMMADRL are computed individually because they are processed conditionally as seen with many if clauses in the algorithm. The training algorithm starts by selecting a minibatch of size M from the replay memory. Each entry in the minibatch includes the combined state S and action A of all client agents S, the set of binary actions of the tasks by the master agent A^{mas}, the system reward of the tasks R, the combined next state S', and a flag that indicates whether the episode was terminated or not *done*.

The master agent is trained by lines 2 through 28. Line 2 computes the target action for every client and every entry of the minibatch using their next state. The target action is to be used to compute the target Q-value using the master agent. Then, lines 3 and 4 concatenate the target actions of the client agents because the master agent accepts a combined state and action of all clients as input as seen in line 7. Lines 5 and 6 check if the client agents have decided to process the tasks locally or propose them to the master agent. For each task that was proposed to the master, a relative Q-value is computed on line 7 and the maximum Q-value will be computed in line 13. Line 9 concatenates the Q-values of the offloaded tasks in the same entry of a minibatch. If no task was offloaded, a Q-value will be computed using a placeholder to train the master agent so that it is used to give feedback like the classical MADDPG. Then, the combined reward is provided to the actions selected to offload their tasks as seen in lines 16 to 20. The current and target Q-values in lines 17 and 17 are used to compute the TD error in line 26. As seen in lines 21 to 25, if all client agents, at any entry of the minibatch, decide to process their tasks locally, the master agent concatenates the state and action of all agents and appends *all_zeros* as a placeholder to learn the Q value when all tasks are processed locally, and it is only used to provide feedback in training the client agents as seen in line 35.

The training of client agents is seen from lines 29 to 44. They are trained similarly to the training of actors in classical MADDPG except that the feedback is computed differently as seen in lines 33 to 39, because the Q value is provided for the client agents that offloaded their task to the server. Therefore, if one or more clients were offloaded their task, the feedback for training the clients is computed from the Q value of one of the offloaded tasks as they are trained with the same rewards. The maximum Q value of the offloaded tasks is considered for consistency. The calculation of the maximum Q-value is the same as that of the training for the master agent.

We used DDQN [21] and prioritized experience replay [18] in the CMMADRL algorithms for better efficiency in the training.

5 Experimental Evaluation

To evaluate the merits of the combinatorial action selection by introducing a master agent to MADDPG, in a task offloading problem with various constraints, we compare our algorithm with other benchmarks and heuristic algorithms as follows.

Algorithm 3 The training algorithm for the client agents and the master agent

Sample a random minibatch of transitions $(S, A, A^{mas}, R, S', done)$ of size M from RM
Set target actions $A'_{i,n} \leftarrow \pi_n\left(S'_{i,n}, \theta'_n\right), \quad \forall n \in N$, and for $i = 1$ to M
$S'_i = \{S'_{i,n}\}, \quad \forall n \in N$, and for $i = 1$ to M
$A'_i = \{A'_{i,n}\}, \quad \forall n \in N$, and for $i = 1$ to M
Get $x_{i,n}$ from $A'_{i,n}$ as described in Section 4.3, $\quad \forall n \in N$, and for $i = 1$ to M
if $x'_{i,n} \geq 0.5, \quad \forall x'_{i,n} \in a'_{i,n}, \quad \forall n \in N$, and for $i = 1$ to M **then**
  $Append\left(Q'_N, Q\left(S'_i, A'_i, S'_{i,n}, A'_{i,n}, \phi'\right)\right)$
end if
$Q'_i = Q'_N$, for $i = 1$ to M
if $Length\left(Q'_i\right)$ is 0 for any i **then**
  $nextQ_i = Q\left(S'_i, A'_i, all\_zeros, all\_zeros, \phi'\right)$
else
  $nextQ_i = Max\left(Q'_i\right)$
end if
y = [], Qs = []
if $A^{mas}_{i,n} = 1 \quad \exists n \in N \quad \exists i \in M$ **then**
  $Append\left(Qs, Q\left(S_i, A_i, S_{i,n}, A_{i,n}, \phi\right)\right)$
  targetQ = $R_i + \gamma \cdot nextQ_i \cdot (1 - done_i)$
  append(y, targetQ)
end if
if $A^{mas}_{i,n} = 0\,, \quad \forall n \in N \quad \exists i \in M$ **then**
  $Append\left(Qs, Q\left(S_i, A_i, all\_zeros, all\_zeros, \phi\right)\right)$
  targetQ = $R_i + \gamma \cdot nextQ_i \cdot (1 - done_i)$
  $append\left(y, targetQ\right)$
end if
Compute the TD error: $\delta = \frac{1}{Length(y)} \cdot \sum_{j=1}^{Length(y)} \left(y_j - Qs_j\right)^2$
Update parameters of master agent ϕ: $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi \delta$
Update target master network $\phi' \leftarrow \phi$
for each client n **do**
  Q^N_i = [] for $i = 1$ to M, $tarQ$ = []
  Set new actions $A^{new}_{i,n} \leftarrow \pi_n\left(S_{i,n}, \theta_n\right), \quad \forall n \in N$, and for $i = 1$ to M
  $A^{new}_i = \{A^{new}_{i,n}\}, \quad \forall n \in N$, and for $i = 1$ to M
  $Append\left(Q^N_i, Q\left(S_i, A^{new}_i, S_{i,n}, A^{new}_{i,n}, \phi\right)\right), \quad \forall n \in N$ with $A^{new}_{i,n} \geq 0$, and for $i = 1$ to M
  if $Length\left(Q^N_i\right)$ is 0 for any i **then**
    $Qloc = Q\left(S_i, A^{new}_i, all\_zeros, all\_zeros), \phi\right)$
    $Append\left(tarQ, Qloc\right)$
  else
    $Append\left(tarQ, Max\left(Q^N_i\right)\right))$
  end if
  Compute the gradient for the client: $\nabla_{\phi_n} J(\phi_n) \leftarrow -\frac{1}{Length(tarQ)} \cdot \sum_{j=1}^{Length(tarQ)} \nabla_{\phi_n} tarQ_j$
  Update the client parameters θ_n:
    $\theta_n \leftarrow \theta_n - \alpha_\theta \nabla_{\theta_n} J\left(\theta_n\right)$
  Update target client networks $\theta'_n \leftarrow \theta_n$
end for

5.1 Benchmark Comparison

The main benchmark for our algorithm is MADDPG for two reasons. First, most existing task-offloading algorithms use DDPG and MADDPG. Second, our algorithm is an extension of MADDPG. However, we also developed different heuristic benchmarks. The heuristics differ from the proposed CMMADRL in that, instead of training a master agent to make combinatorial action selections about the clients, a stationary algorithm is used to decide on which of the clients to approve for the MEC server based on some ordering mechanism. The benchmark algorithms are discussed below.

- MADDPG: This benchmark uses actor agents to make decisions. Its difference from CM-MADRL is in the procedure 2) of the action selection in Section 4.3, where the SDN allocates the tasks to the sub-channels according to their order of offloading rather than making combinatorial action selection. Tasks that are not assigned to any channel are dropped. The actor agent in the UD will assign τ_{max} to its $T_{\text{MEC},n}$ for the dropped tasks as a penalty. Because the penalty can be unfair for benchmark comparison, the following heuristics are developed to have equivalent combinatorial action selection with the CMMADRL for the tasks that are not accepted.
- MADDPG with the shortest offloading time first heuristic: This is similar to MADDPG, with the distinction that tasks not assigned to sub-channels or storage are designated for local processing.
- MADDPG with deadline/size first heuristic: This differs from the MADDPG with the shortest offloading time first heuristic in that it uses the increasing order of deadline/size as a priority rather than offloading time.

Even if the way tasks are accepted by the MEC server differs between the benchmarks and the CMMADRL, the order of processing of the accepted tasks is always in the order of arrival at the MEC using Equation (5)

5.2 Experimental Settings

The experimental setting for the task offloading environment is provided in Table 2. Note that all UDs have the same minimum battery, power, and resource allocation threshold, but their maximum budget is generated from a uniform distribution. As described in Section 2, the experimental setting is customized from the settings in [14] and [25]. The typical storage capacity of modern servers is GB and TB. However, because we chose a small experimental setting due to computational resources, we considered a storage constraint of 400 MB so that the task offloading problem is combinatorial to the server. Evaluation episodes are seeded with their index. A seed of 37 is used for the reproducibility of the simulation environment. The source code is publicly available at https://github.com/TesfayZ/CCM_MADRL_MEC.

The hyperparameters for the CMMADRL and benchmark algorithms are as follows: a discount factor of 0.99, a minibatch size of 64, a replay memory size of 10,000, and learning rates of 0.0001 for the client and actor networks, and 0.001 for the master and critic networks. These values were selected based on performance results from a grid search over 16 combinations of learning rates. The output activation function is Tanh for the clients and actors, while ReLU is used for the master and critic networks.

The neural network architecture for the clients and actors consists of an input layer with 7 neurons, two hidden layers with 64 and 32 neurons, and an output layer of size 3. The critic network in the benchmark algorithms has an input layer of 500 neurons, representing the combined state and action size for the 50 actors, followed by two hidden layers with 512 and 128 neurons, and a single output neuron. The master agent has similar parameters to the critic, but its input layer has

510 neurons to account for computing the Q-value for each client using a per-action DQN [6]. All input and hidden layers use linear activation functions.

Table 2. Experimental parameters of CMMADRL

Param	Value	Param	Value
$\|N\|$	50	P^{max}	24 dBm
K	10	P^{min}	1 dBm
τ_n	[0.1-0.9] s	p_n^{max}	[P^{min} - P^{max}] dBm
W	40 MHz	p_n^{min}	P^{min} dBm
f_{max}	1.5 GHz	f_{min}	0.4 GHz
f_n^{max}	[f^{min}- f^{max}] GHz	b^{max}	3.2 MJ
f_n^{min}	f^{min} GHz	b^{min}	0.5 MJ
g_n	[5-14] dB	f_e	4 GHz
z_n	[1-50] MB	z_e	400 MB
c_n	[300 - 737.5] cycles	U_e	8
κ	5×10^{-27}	λ_1, λ_2	0.5, 0.5
e_n	0.001 J	b_n^{max}	[b^{min}- b^{max}] MJ
b_n^{min}	b^{min}		

5.3 Complexity Analysis

Table 3 presents the analysis of the inference time complexity of CMMADRL and MADDPG using multiply-accumulate operations (MACs) per forward pass in the neural networks [19]. Note that the other heuristic benchmark algorithms have the same complexity as the MADDPG because the heuristics are not neural network models. The number of MACs per layer of a neural network is computed as $h_i \times h_{i+1}$ where h_i is the number of neurons at layer i.

Using the experimental setting in Section 5.2, the actor agent of the MADDPG and the client agent of CMMADRL have the same complexity with four layers of seven neurons in the input, 64 and 32 neurons in the two hidden layers, and three neurons in the output. Their number of MACs is $7 \times 64 + 64 \times 32 + 32 \times 3 = 448+2048+96 = 2592$. The difference lies in the critic network of MADDPG and the master agent of CMMADRL. The critic network has 500 inputs while the master agent has 510. This leads to a number of MACs of $500 \times 512 + 512 \times 128 + 128 \times 1 = 256000 + 65536 + 128 = 321664$ for the critic network, and $510 \times 512 + 512 \times 128 + 128 \times 1 = 261120 + 65536 + 128 = 326784$ for the neural network of the master agent. Therefore, the total number of MACs differs due to the additional 10 input features in the master agent.

The master agent uses per-client DQN to compute the Q-values of the tasks that are proposed for combinatorial action selection by the client agents, which can range from 0 up to $N = 50$. Therefore, assuming $\mathcal{M}$ tasks are proposed by the client agents for the master agent, a total of $\mathcal{M} \times 326784$ MACs are executed by the master agent. However, because the computation of Q-values in the per-client-DQN are independent, they can be computed in parallel, resulting in a constant value of 326784 MACs. The experimental implementation does not exploit parallelism in executing the per-client DQNs.

The difference in algorithmic complexity during training is relatively smaller compared to inference, as the complexity of the critic network (321664 MACs) must also be taken into account.

Table 3. Inference time complexity comparison in MACs: MADDPG vs CMMADRL under sequential and parallel execution of the per-client-DQN

Component	MADDPG	CMMADRL (Sequential)	CMMADRL (Parallel)
Actor/Client	$50 \times 2592 = 129600$	129600	129600
Critic/Master	None in execution	$\mathcal{M} \times 326784,\ \mathcal{M} \leq 50$	326784 (if $\mathcal{M} > 0$)
Worst-case total	**129600**	11.65×10^7	456384
Best-case total	**129600**	129600	129600

5.4 Generalizability

Because convergence is affected by the initialization of the weights of the DNNs and the exploration and exploration sequence, we evaluate the algorithms in a different evaluation environment. At each episode of the training environment, the DRL is evaluated with 50 evaluation episodes.

5.5 Results and Discussion

This section presents a discussion of the experimental results across three key scenarios: the number of time steps per episode and maximum battery capacity, the trade-off between energy consumption and latency, and the number of UDs. In the first experiment, no UD fell below the battery threshold as seen in Figure 3 (C). To better capture battery dynamics, we extended the number of time steps and minimized the maximum battery capacity, as shown in Figure 4. Subsequently, we varied λ_1 and λ_1 to analyze their impact on the trade-offs between energy consumption and latency, as demonstrated in Figure 5 and Figure 6. Finally, we conducted experiments using different numbers of UDs to assess the proposed algorithm's effectiveness under varying levels of combinatorial complexities, as shown in Figure 7.

Results for 10 steps per episode: First, we run an experiment with 10 steps per episode for 2000 training episodes. We performed 10 experiments using different initializations of DNN weights and different sequences of exploration and exploitation for each run and plotted the result with a 95% confidence interval as seen in Figure 3. As seen in Figure 3(A), the CMMADRL algorithm achieved a 59.8% improvement in combined reward compared to the benchmark algorithms at their point of peak convergence. This performance gain is attributed to the master agent's ability to make combinatorial action selections based on the clients' chosen actions. On the other hand, MADDPG-based heuristic and benchmark algorithms use only actors to provide actions. The critic is only used to provide feedback to the actors. Similarly, Figure 3 (B) shows that CMMADRL results in a smaller percentage of tasks expiring before completion compared to the other algorithms, with an 18% gap over the best benchmark. Note that Figure 3 (A) is the performance based on the combined reward and the combined penalty for the time and energy consumption in Equation (20) but Figure 3 (B) is a percentage of tasks whose deadline expired before completing their processing to the total number of tasks generated in the episode. Figure 3 (C) shows that none of the UDs exceeds the minimum battery threshold in an episode. This is because the experiments are run for 10 steps per episode.

Results for 100 steps per episode and $b_{max} = b_{min} + 1J$: Next, to see the impact on the battery level, we repeat the above experiment by changing the number of steps per episode to 100 and the b_{max} to $b_{min} + 1J$. Figure 4 (C) shows that CMMADRL has more UDs running below the minimum battery threshold than the benchmark algorithms. Nonetheless, it outperformed the benchmark algorithms with a 1.7% improvement in task completion and a 3.43% improvement in combined reward, as shown in Figure 4 (B) and Figure 4(A) respectively. As explained above for Figure 3, the

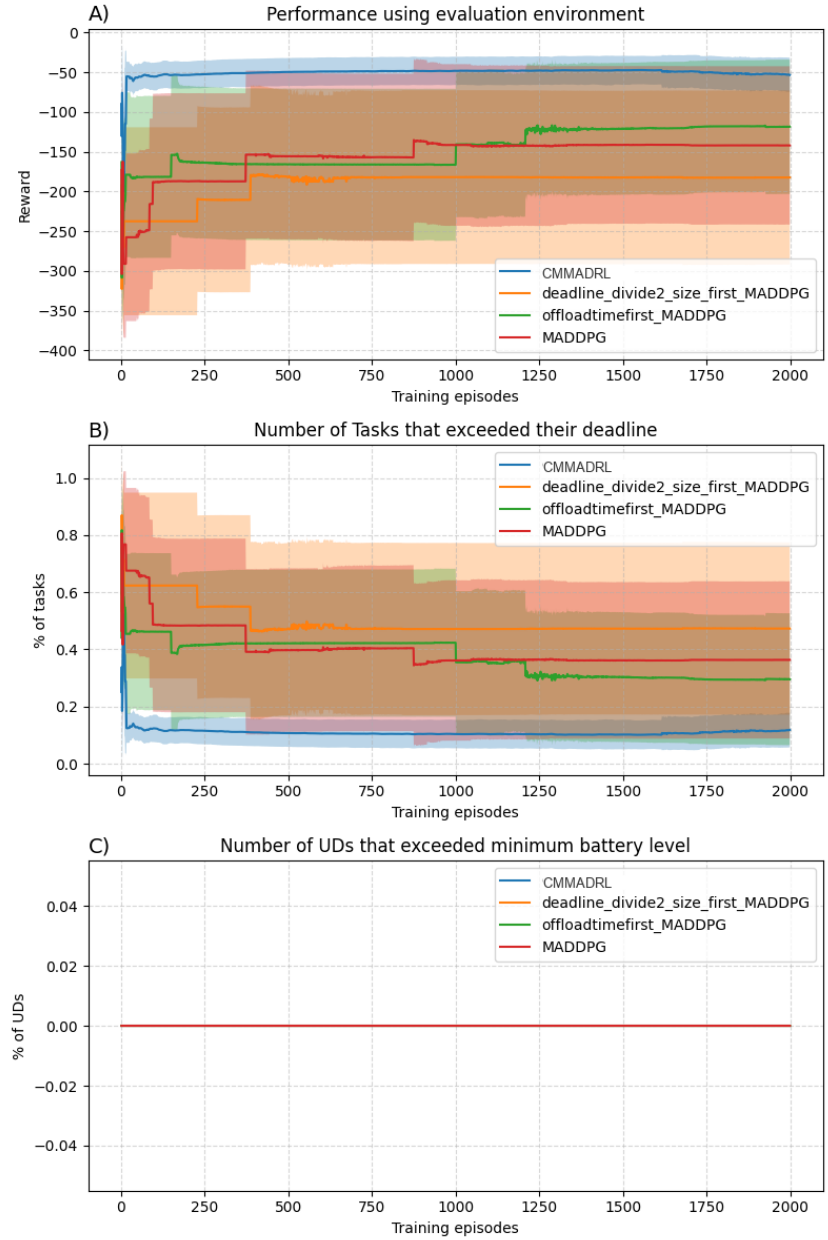

Fig. 3. Comparison of the performance of CMMADRL, MADDPG, and the heuristic algorithms under λ_1 = 0.5, λ_2 = 0.5, and 10 steps per episode.

algorithm is trained by computing a scalar reward, which is a sum of energy and time consumption, as seen in Equation (20) but the percentage of UDs that exceed their battery threshold is from the ratio of UDs that exceed the battery threshold to the total number of UDs. Therefore, it is affected by the scales given to the deadline penalty and the energy penalty. We used $\lambda_1 = \lambda_2 = 0.5$ as weight

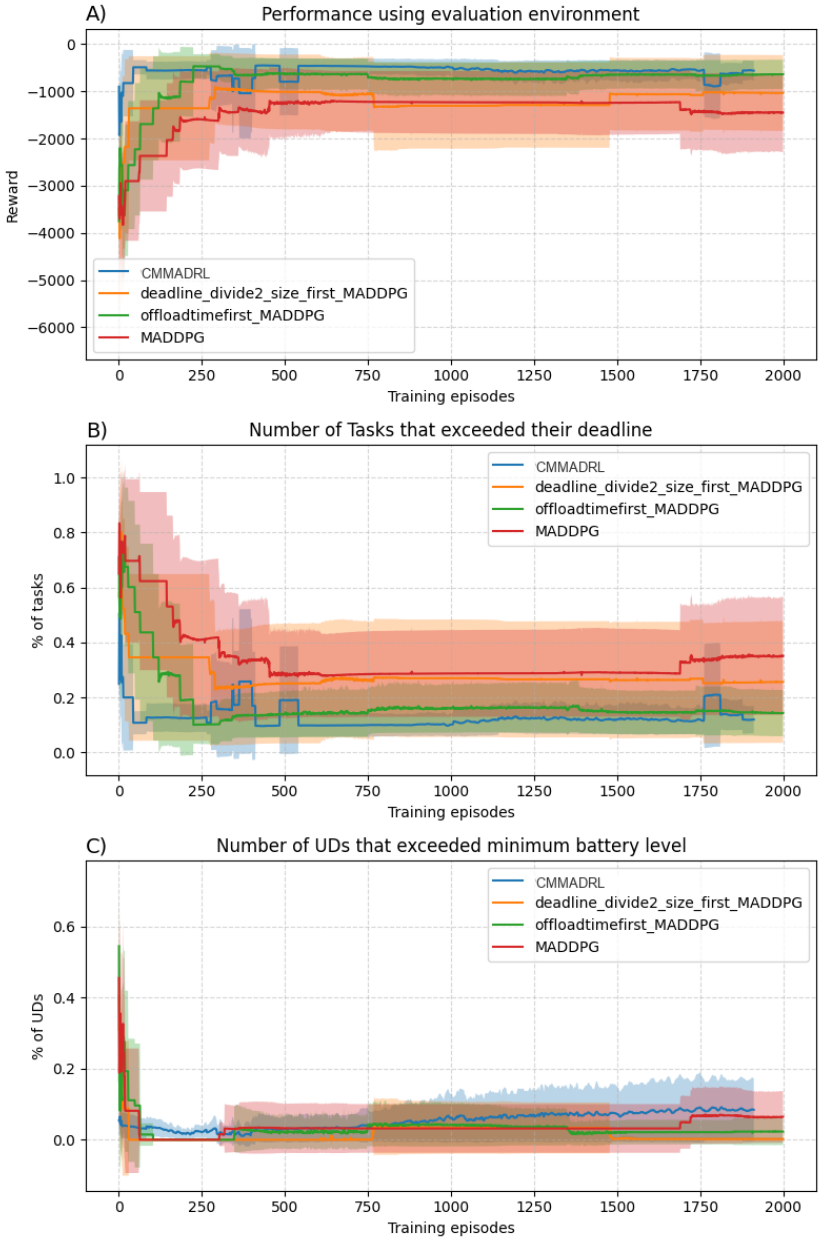

Fig. 4. Comparison of the performance of CMMADRL, MADDPG, and the heuristic algorithms under λ_1 = 0.5, λ_2 = 0.5, 100 steps per episode, and $b_{max} = b_{min} + 1J$.

coefficients for processing time and energy consumption. We repeated the algorithm for λ_1 = 1 and λ_2 = 5 as seen in Figure 5.

Results for different values of λ_1 and λ_2: We further conducted the experiment with $\lambda_1 = 1$ and $\lambda_2 = 5$, motivated by the observation that Figure 4(B) appears to be the inverse of Figure 4(A), which corresponds to $\lambda_1 = \lambda_2 = 0.5$. This suggests that the equal weighting does not proportionally balance energy consumption, which is relatively small, and time consumption, which is comparatively larger.

The results are shown in Figure 5, plotted with a 95% confidence interval over 40 runs, in contrast to the previous two experiments, which used 10 runs. However, the plots do not show a significant difference between using 10 and 40 runs. Subplots (A) and (B) are no longer exact inverses of each other. Subplot (C) shows that CMMADRL still resulted in 8% more UDs falling below the minimum battery threshold compared to the benchmark and heuristic algorithms. Nonetheless, CMMADRL outperformed the other algorithms by 57.5% in combined performance and provided a 19.9% advantage in the number of tasks that exceed their deadlines.

Although scalar rewards are often used to combine multiple objectives, this approach may not fully capture the complexity of the underlying problem [20]. A more comprehensive treatment of multi-objective reinforcement learning is presented in Hayes et al. [5], but such an exploration is beyond the scope of this work.

We continue the experiment with $\lambda_1 = 1$ and $\lambda_2 = 1000$, as shown in Figure 6. Because energy consumption is scaled by a value of 1000, the percentage of UDs exceeding the battery threshold dropped to zero. In this setting, CMMADRL achieved a 9% improvement in task completion rate compared to the best benchmark (MADDPG combined with the shortest offloading time first heuristic). CMMADRL also demonstrated superior performance in the combined reward of latency and energy consumption. Note that it seems that the performances of the algorithms look overlapped, as seen in Figure 6 (A). But this is due to the magnitude of the total reward, which is in millions due to the scaling effect of λ_2. The maximum value at convergence of the algorithms is; CMMADRL=-3207, MADDPG with deadline/size first heuristic = -4063, MADDPG with the shortest offloading time first heuristic = -3895, and MADDPG=-4567.

It can be observed that the benchmark and heuristic algorithms performed more comparably to CMMADRL when the episode length was increased to 100 steps, as opposed to just 10 steps. The key reason lies in the training strategy: all algorithms undergo training only at the end of each episode. This causes training after 10 steps to overfit to the data collected at the early episodes. In contrast, with 100-step episodes, the algorithms are exposed to ten times more data before retraining, enabling better generalization. The benchmark and heuristic algorithms are impacted by overfitting more than CMMADRL because they use only their actors to select action, while the CMMADRL uses the advantage of both clients and master to mitigate overfitting and sticking to local optimal.

Note that the CMMADRL in Figures 4 and 5 is not plotted until the last episode. The experiment is run on Iridis [2], an HPC cluster at the University of Southampton. The experiments were run for 60 hours each. All of the experiments for the heuristic and benchmark algorithms were finished earlier, but some of the runs for CMMADRL ran out of time before reaching the last episode. For convenience in plotting with the 95% confidence interval, all runs of CMMADRL are clipped after the run with the least number of episodes. Note that the benchmark and heuristic algorithms have only one Q value in the critic for a combination of state and actions of the actors. On the other hand, the number of Q values to train in the CMMADRL is equal to the number of offloaded tasks or 1 if all of them are allocated locally.

Performance across differing numbers of UDs: The bar chart in Figure 7 illustrates the performance of the experiment across different numbers of UDs. The values shown here are the peak performance

[2] https://www.southampton.ac.uk/isolutions/staff/iridis.page

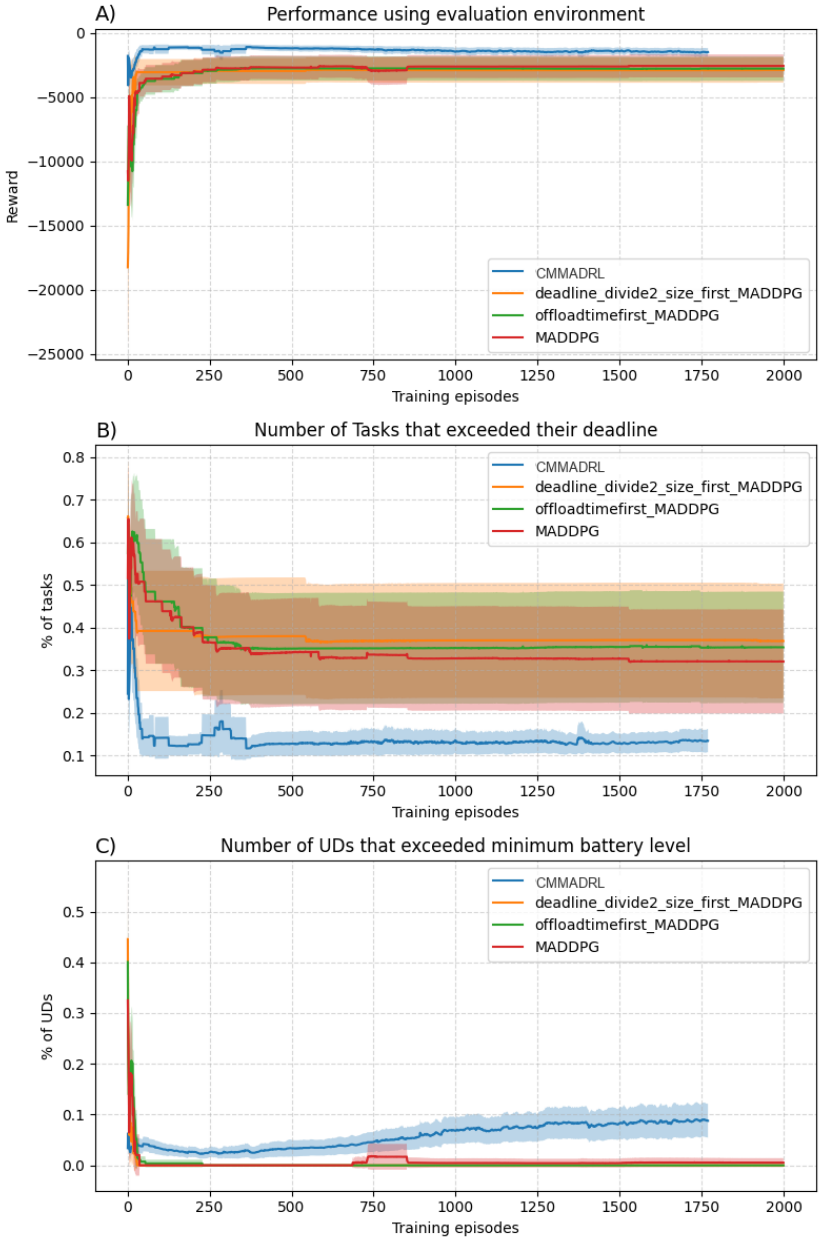

Fig. 5. Comparison of the performance of CMMADRL, MADDPG, and the heuristic algorithms under $\lambda_1 = 1$, $\lambda_2 = 5$, 100 steps per episode, and $b_{max} = b_{min} + 1J$.

at convergence for each experiment. When the number of UDs is 8, the number of tasks does not exceed the number of available sub-channels, and the total size of the tasks does not exceed the server's storage capacity. As a result, the heuristic and MADDPG algorithms perform similarly, as decisions are made solely using the actor networks of the MADDPG, without reliance on any

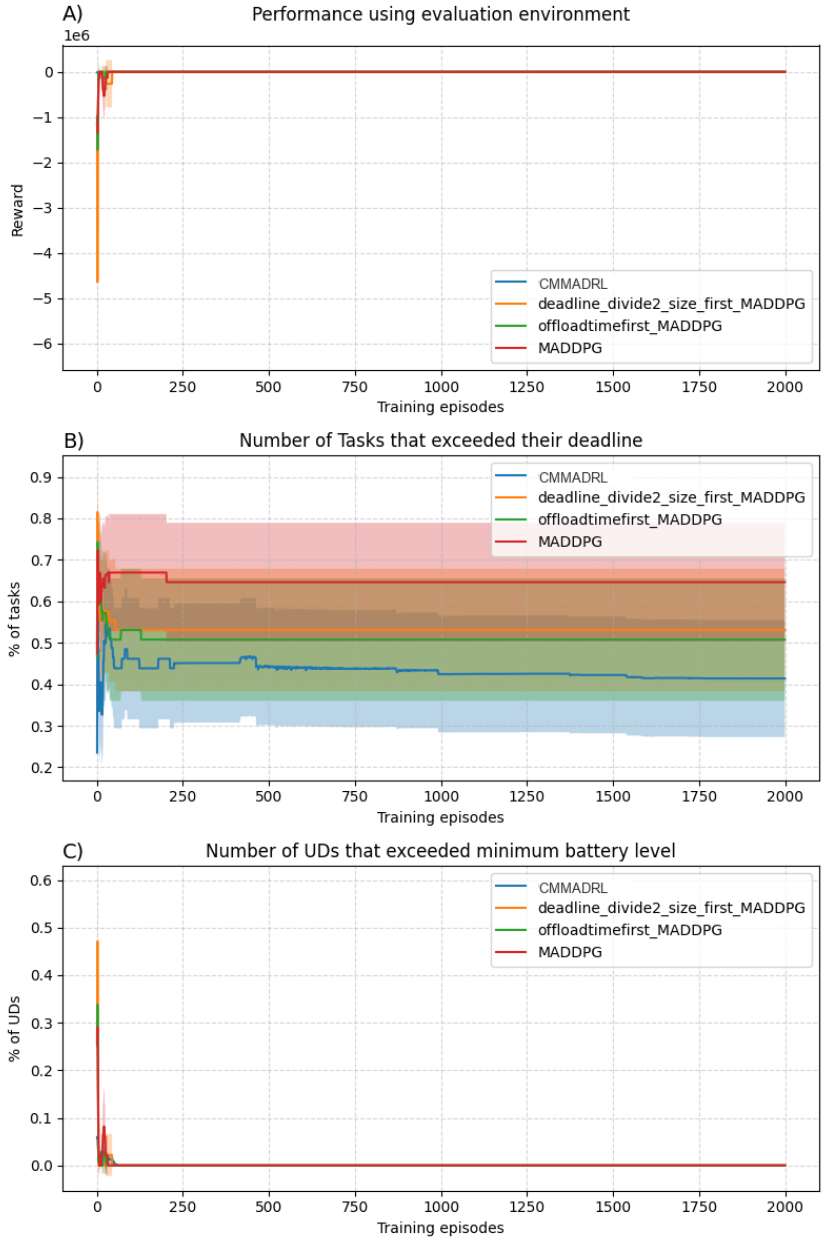

Fig. 6. Comparison of the performance of CMMADRL, MADDPG, and heuristic algorithms under $\lambda_1 = 1$, $\lambda_2 = 1000$, 100 steps per episode, and $b_{max} = b_{min} + 1J$.

heuristic methods. This is because the combinatorial constraints are not activated. The small performance differences observed are attributed to the randomness in the initialization of the neural network weights. CMMADRL algorithm has performed slightly lower than the benchmark algorithms for 8-UDs case. This is due to its relatively complex neural network compared to the

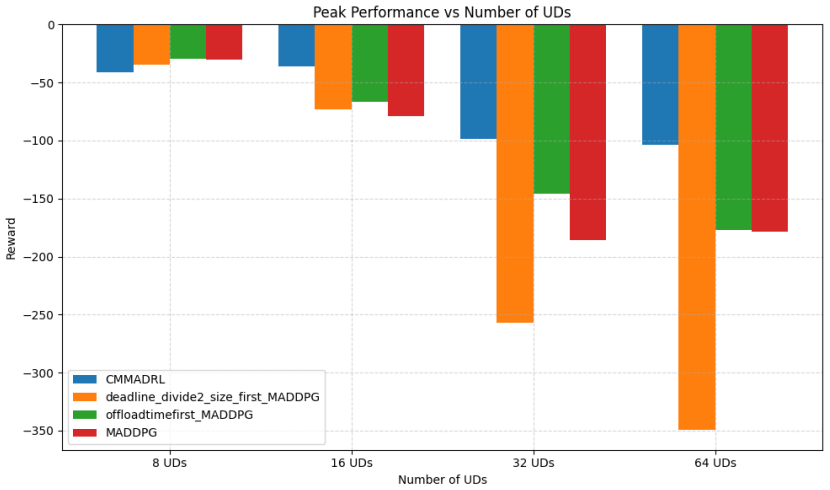

Fig. 7. Comparison of the performance of CMMADRL, MADDPG, and the heuristic algorithms at their peak convergence for varying number of UDs under $\lambda_1 = 1$, $\lambda_2 = 1$, and 10 steps per episode.

others, as detailed in Table 3. Specifically, the master agent, which is only used as a critic network in this case, has 10 more neurons in the input layer and has to compute Q-value for every task offloaded to the server to compute the maximum Q-value.

For 16, 32, and 64 UDs, the efficacy of the CMMADRL algorithm improves with increasing number of UDs. This is because the problem becomes combinatorially complex when the network and storage constraints are activated. For the benchmark algorithms, the one that prioritizes decisions based on the order of the task offloading time has outperformed the others as it minimizes latency. However, since latency is influenced not only by offloading time, but also by local processing capacity, battery level, and the decisions on other tasks, the CMMADRL algorithm ultimately achieves superior performance by learning Q-values that account for all of these factors. Compared to the offloading-time-first heuristic approach, CMMADRL demonstrates an improvement of 45.86% for 16 UDs, 32.63% for 32 UDs, and 41.35% for 64 UDs.

6 Conclusion and Limitations

Conclusion: In this paper, we propose a client-master MADRL algorithm for task offloading in MEC, that considers various constraints at the UDs, communication sub-channels, and server's storage. By combining the advantages of both policy gradient and value functions to produce continuous and discrete actions, CMMADRL provides better convergence than the existing homogeneous MADRL algorithm because the master agent applies combinatorial action selection on the actions proposed by the clients.

Limitations: Although the primary contribution of this work lies in advancing existing MADDPG-based RL algorithms to a client-master MADRL for combinatorial action selection, it has several limitations that present opportunities for future extension:

- The system model of this work considered a single server and multiple UDs. In the future, we plan to extend CMMADRL to multiserver MEC where multiple servers cooperate to make matching and combinatorial action selection.

- The real storage capacities of servers are in GBs and TBs. However, due to computational resource limits for training the multiple client and master agent algorithms, CMMADRL is simulated on 50 UDs running their client agents. To make the server storage capacity critical for combinatorial action selection to the tasks of the UDs, we assumed a server capacity of 400 MB. Note that with 50 UDs, a total of 51 DRL agents are training simultaneously, 50 of which are client agents and one master agent.
- CCMMARL accepts only a fixed number of UDs. This is because the master agent uses a standard feedforward neural network, like the critic network of MADDPG. By replacing the feedforward neural network with the transformer neural network-based coalition action selection algorithm in [4], the CMMADRL algorithm can be extended for dynamic number of users.
- As discussed in the analysis of Figure 5 the experiments use a scalar reward function defined a weighted sum of latency and energy consumption, with λ_1 and λ_2 controlling the balance between them. CMMADRL can be extended to multiobjective DRL with a vectorized reward [5] of latency and energy consumption.

Acknowledgments

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. This work was also supported by an EPSRC Turing AI Acceleration Fellowship (EP/V022067/1).

References

[1] Seble Birhanu Engidayehu, Tahira Mahboob, and Min Young Chung. 2022. Deep Reinforcement Learning-based Task Offloading and Resource Allocation in MEC-enabled Wireless Networks. In *2022 27th Asia Pacific Conference on Communications (APCC)*. 226–230.

[2] Guang Chen, Yueyun Chen, Zhiyuan Mai, Conghui Hao, Meijie Yang, and Liping Du. 2023. Incentive-Based Distributed Resource Allocation for Task Offloading and Collaborative Computing in MEC-Enabled Networks. *IEEE Internet of Things Journal* 10, 10 (2023), 9077–9091.

[3] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. 2015. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679* (2015).

[4] Tesfay Zemuy Gebrekidan, Sebastian Stein, and Timothy J. Norman. 2024. Deep Reinforcement Learning with Coalition Action Selection for Online Combinatorial Resource Allocation with Arbitrary Action Space. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 660–668.

[5] Conor F Hayes, Roxana Rădulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M Zintgraf, Richard Dazeley, Fredrik Heintz, et al. 2022. A practical guide to multi-objective reinforcement learning and planning. *Autonomous Agents and Multi-Agent Systems* 36, 1 (2022), 26.

[6] Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. 2015. Deep reinforcement learning with a natural language action space. *arXiv preprint arXiv:1511.04636* (2015).

[7] Akhirul Islam, Arindam Debnath, Manojit Ghose, and Suchetana Chakraborty. 2021. A survey on task offloading in multi-access edge computing. *Journal of Systems Architecture* 118 (2021), 102225.

[8] Wei Jiang, Daquan Feng, Yao Sun, Gang Feng, Zhenzhong Wang, and Xiang-Gen Xia. 2023. Joint Computation Offloading and Resource Allocation for D2D-Assisted Mobile Edge Computing. *IEEE Transactions on Services Computing* 16, 3 (2023), 1949–1963.

[9] Te-Yi Kan, Yao Chiang, and Hung-Yu Wei. 2018. Task offloading and resource allocation in mobile-edge computing system. In *2018 27th Wireless and Optical Communication Conference (WOCC)*. 1–4.

[10] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[11] Xiaowei Liu, Shuwen Jiang, and Yi Wu. 2022. A novel deep reinforcement learning approach for task offloading in MEC systems. *Applied Sciences* 12, 21 (2022), 11260.

[12] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 6382–6393.

[13] Michael Pendo John Mahenge, Chunlin Li, and Camilius A Sanga. 2022. Energy-efficient task offloading strategy in mobile edge computing for resource-intensive mobile applications. *Digital Communications and Networks* 8, 6 (2022), 1048–1058.

[14] D. C. Nguyen, M. Ding, P. N. Pathirana, A. Seneviratne, J. Li, and H. Poor. 2023. Cooperative Task Offloading and Block Mining in Blockchain-Based Edge Computing With Multi-Agent Deep Reinforcement Learning. *IEEE Transactions on Mobile Computing* 22, 04 (2023), 2021–2037.

[15] Lili Nie, Huiqiang Wang, Guangsheng Feng, Jiayu Sun, Hongwu Lv, and Hang Cui. 2023. A deep reinforcement learning assisted task offloading and resource allocation approach towards self-driving object detection. *Journal of Cloud Computing* 12, 1 (2023), 131.

[16] Kuanishbay Sadatdiynov, Laizhong Cui, Lei Zhang, Joshua Zhexue Huang, Salman Salloum, and Mohammad Sultan Mahmud. 2023. A review of optimization methods for computation offloading in edge computing networks. *Digital Communications and Networks* 9, 2 (2023), 450–461.

[17] Dileep Kumar Sajnani, Abdul Rasheed Mahesar, Abdullah Lakhan, and Irfan Ali Jamali. 2018. Latency aware and service delay with task scheduling in mobile edge computing. *Communications and Network* 10, 04 (2018), 127–141.

[18] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized experience replay. (2016).

[19] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[20] Peter Vamplew, Benjamin J Smith, Johan Källström, Gabriel Ramos, Roxana Rădulescu, Diederik M Roijers, Conor F Hayes, Fredrik Heintz, Patrick Mannion, Pieter JK Libin, et al. 2022. Scalar reward is not enough: A response to silver, singh, precup and sutton (2021). *Autonomous Agents and Multi-Agent Systems* 36, 2 (2022), 41.

[21] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*. 2094–2100.

[22] Suzhen Wang, Zhongbo Hu, Yongchen Deng, and Lisha Hu. 2022. Game-Theory-Based Task Offloading and Resource Scheduling in Cloud-Edge Collaborative Systems. *Applied Sciences* 12, 12 (2022).

[23] Jianyu Xiong, Peng Guo, Yi Wang, Xiangyin Meng, Jian Zhang, Linmao Qian, and Zhenglin Yu. 2023. Multi-agent deep reinforcement learning for task offloading in group distributed manufacturing systems. *Engineering Applications of Artificial Intelligence* 118 (2023), 105710.

[24] Jia Yan, Suzhi Bi, and Ying-Jun Angela Zhang. 2018. Optimal Offloading and Resource Allocation in Mobile-Edge Computing with Inter-User Task Dependency. In *2018 IEEE Global Communications Conference (GLOBECOM)*. 1–8.

[25] Jing Zhang, Jun Du, Yuan Shen, and Jian Wang. 2020. Dynamic Computation Offloading With Energy Harvesting Devices: A Hybrid-Decision-Based Deep Reinforcement Learning Approach. *IEEE Internet of Things Journal* 7, 10 (2020), 9303–9317.

[26] Weiwen Zhang, Yonggang Wen, Jun Wu, and Hui Li. 2013. Toward a unified elastic computing platform for smartphones with cloud support. *IEEE Network* 27, 5 (2013), 34–40.

Received 05 November 2024; revised 14 July 2025; accepted 12 September 2025