A DRL-Based Task Offloading Scheme for Server Decision-Making in Multi-Access Edge Computing

Ducsun Lim and Inwhee Joe *

Abstract: Multi-access edge computing (MEC), based on hierarchical cloud computing, offers abundant resources to support the next-generation Internet of Things network. However, several critical challenges, including offloading methods, network dynamics, resource diversity, and server decision-making, remain open. Regarding offloading, most conventional approaches have neglected or oversimplified multi-MEC server scenarios, fixating on single-MEC instances. This myopic focus fails to adapt to computational offloading during MEC server overload, rendering such methods sub-optimal for real-world MEC deployments. To address this deficiency, we propose a solution that employs a deep reinforcement learning-based soft actor-critic (SAC) approach to compute offloading and facilitate MEC server decision-making in multi-user, multi-MEC server environments. Numerical experiments were conducted to evaluate the performance of our proposed solution. The results demonstrate that our approach significantly reduces latency, enhances energy efficiency, and achieves rapid and stable convergence, thereby highlighting the algorithm’s superior performance over existing methods.

Keywords: mobile edge computing; directed acyclic graphs; deep reinforcement learning; soft actor-critic; Markov decision process; task offloading

1. Introduction

In recent years, advances in wireless technology combined with the widespread adoption of the Internet of Things have paved the way for innovative computation-intensive applications, which include augmented reality (AR), mixed reality (MR), virtual reality (VR), online gaming, intelligent transportation, and industrial and home automation. Consequently, the demand for these applications has surged [1–3]. By 2020, the number of IoT devices is projected to skyrocket to 24 billion. This tremendous increase signifies that many smart devices (SDs) and sensors have been responsible for generating and processing an immense volume of data [4].

To cater to these computationally intensive applications, substantial computing resources and high performance are required. Addressing the escalating need for energy efficiency and managing the swift influx of user requests has emerged as significant challenges [5]. Initially, mobile cloud computing (MCC) was considered a viable solution for processing computationally intensive tasks. However, as the demand for real-time processing increased, the limitations of MCC became apparent [6], resulting in the introduction of mobile edge computing (MEC) as a potential solution to meet this burgeoning demand [7].

Multi-access edge computing [8] is effective at deploying computing resources close to SDs, collaborative radio resource management (CRRM), and collaborative signal processing (CRSP). Conversely, a cloud radio access network (C-RAN) employs centralized signal processing and resource allocation, efficiently catering to user requirements [9]. Collectively, the attributes of these technologies have the potential to fulfill diverse requirements in upcoming artificial intelligence (AI)-based wireless networks [10].
Leveraging MEC to offload tasks is a promising approach to curtail network latency and conserve energy. Specifically, MEC addresses the computational offloading requirements of IoT devices by processing tasks closer to the edge rather than relying solely on a central cloud [11]. However, since the task offloading problem is recognized as a non-deterministic polynomial-time hard (NP-hard) problem [12], addressing it is challenging. Although most research in this area has leaned toward heuristic or convex optimization algorithms, the increasing complexity of MEC coupled with varying radio channel conditions makes it difficult to consistently guarantee optimal performance using these conventional methods. Given that optimization problems often require frequent resolution, meticulous planning is imperative for designing and managing future MEC networks.

In recent years, deep reinforcement learning (DRL), a sub-set of AI, has gained significant attention to its ability to tackle complex challenges across various sectors. As IoT networks become more distributed, the need for decentralized decision-making to enhance throughput and reduce power consumption increases, with DRL serving as a key tool. The emergence of the multi-access edge computing (MEC) paradigm has added complexity to multi-user, multi-server environments, bringing data-offloading decision-making to the forefront [13]. This MEC landscape necessitates addressing both user behavioral aspects and server pricing policies. A recent study combined prospect theory and the tragedy of the commons to model user satisfaction and potential server overexploitation, highlighting the intricate nature of the problem. In the context of MEC, while some research has explored DRL for task offloading, the focus has been predominantly on holistic offloading, overlooking the advantages of partial offloading, such as reduced latency and improved quality of service (QoS). Collaborative efforts among MEC servers, especially within a multi-server framework, have been significantly useful in enhancing overall system performance.

In this study, we address the pressing demands and challenges in MEC environments by proposing a computational offloading and resource allocation method based on the soft actor-critic (SAC) algorithm. Our choice of SAC, although commonly used in other works, was carefully considered for its unique objective function and compatibility with Markov decision process (MDP) modeling in a multi-user, multi-server MEC environment. What sets our approach apart from other studies is the context in which SAC is applied and its dynamic interaction with the environment to make optimal offloading decisions. Our method advocates a collaborative computation offloading and resource allocation strategy across multiple MECs to improve both latency and energy efficiency. By employing SAC for task offloading, our system benefits from real-time learning, resulting in noticeable performance enhancements. The key contributions of our research include the following:

- Acknowledging the frequent utilization of SAC in related literature, our implementation innovatively models interrelated task division problems within the MDP framework, incorporating the state space, action space, and reward function. We introduce a discrete SAC-based intelligent computing offloading algorithm that prioritizes stability and sample efficiency. Notably, this variant of the algorithm is proficient at exploration within a continuous action space, setting it apart from conventional applications.
- Our research explores the complexities encountered when multiple SD users offload tasks to multiple MEC servers through a base station (BS). Given the increasing number and varied distribution of edge servers around SDs, we argue that task offloading transcends simple binary decisions. Our approach addresses both the offloading decision and the intricate selection of a collaborative MEC server.
- We stress the importance of inter-MEC collaboration in enhancing task processing within distributed systems. Through the integration of various technologies, inter-MEC collaboration not only improves task processing efficiency but also enhances the user experience.

The remainder of this paper is organized as follows: Section 2 revisits prior research that underpins the aim of this study. Section 3 elucidates the core components of the system, along with a mathematical model tailored for offloading. Then, Section 4 highlights the imperatives of the DRL-based offloading scheme and delves into its associated challenges.
and goals, and Section 5 details the cost function, optimization challenges, architecture, and mechanics of the proposed DRL-based offloading scheme. Then, Section 6 presents an evaluation of the performance of the scheme through empirical analysis. Finally, Section 7 rounds off the conclusions of the study and outlines potential avenues for future work.

2. Related Work

Recent research in the field of MEC has aimed to reduce latency and energy consumption through computation offloading and resource allocation techniques. A heuristic offloading algorithm designed to efficiently manage computationally intensive tasks was introduced [14]. This algorithm can achieve high throughput and minimize latency when transferring tasks from an SD to an MEC server. However, despite its critical role in enhancing overall system performance, the decision-making process for offloading under the algorithm is overly focused on task priorities.

A collaborative method between fog and cloud computing to curtail service delays on IoT devices was explored [15]. This study focused on strategies for optimizing computing offloading, allocating computing resources, managing wireless bandwidth, and determining transmission power within a combined cloud/fog computing infrastructure. The overarching goal of these optimization strategies was to reduce both latency and energy consumption. Notably, the authors in both [16,17] employed sub-optimal methods, favoring minimal complexity, and they highlighted the significance of practical and efficient approaches.

The dynamics of energy link selection and transmission scheduling, particularly when processing applications that demanded optimal energy within a network linking SDs and MEC servers, were investigated [18]. Relying on an energy consumption model, the authors formulated an algorithm for energy-efficient link selection and transmission scheduling. An integrated algorithm that facilitated adaptive long-term evolution (LTE)/Wi-Fi link selection and data transmission scheduling was presented to enhance the energy efficiency of SDs in MCC systems [19]. Upon evaluation, the proposed algorithm outperformed its counterparts in terms of energy efficiency. Furthermore, it demonstrated proficiency in managing battery life, especially when considering the unpredictable nature of wireless channels. While these two studies prioritized energy efficiency and the proposed algorithms showed commendable performances, the studies did not address the adaptability required under varying network conditions.

The challenges of processing vast amounts of data and computational tasks using deep Q-network (DQN)-based edge intelligence within the MEC framework [20] were addressed. The authors of this study focused on the distribution of computational tasks and the allocation of resources between edge devices and cloud servers. Meanwhile, the authors [21] addressed the performance degradation and energy imbalances in SDs with a deep reinforcement learning-based offloading scheduler (DRL-OS). Noteworthy, as the tally of wireless devices skyrockets, the expenses associated with DQN-based methodologies also increase.

Several studies have leveraged actor-critic-based offloading in MEC environments to optimize service quality by analyzing agent behaviors and policies [22,23]. The authors [24] delved into the offloading challenges in multi-server and multi-user settings, whereas the authors [25] integrated the proximal policy optimization (PPO) algorithm for task offloading decisions. Implementing the PPO in practical scenarios can be challenging because of its extensive sampling requirements. Wang et al. [26] conducted a study centered on task offloading decisions using the PPO algorithm, and Li et al. [27] addressed the offloading issues within a multi-MEC server and in multi-user contexts.

Furthermore, several investigations have focused on using the deep deterministic policy gradient (DDPG) algorithm to counteract the offloading issues inherent in the MEC domain. Notably, DDPG outperforms PPO in terms of continuous action space, data efficiency, and stability, making it pivotal for reinforcement learning endeavors in the MEC space and offering effective solutions to offloading challenges. However, within specific
environments, the random-search nature of a network may pose hurdles in identifying the optimal policy. By contrast, SAC boasts greater stability than deterministic policies and exhibits excellent sampling efficiency. Modern research is now leveraging SAC to address computational offloading challenges. Liu et al. [28] enhanced the data’s efficiency and stability using SAC, where multiple users collaboratively execute task offloading in an MEC setting. Similarly, Sun et al. [29] harnessed SAC within 6G mobile networks, achieving heightened data efficiency and reliability in MEC settings. The advantages and disadvantages of some existing approaches are listed in Table 1.

Table 1. Comparison of existing approaches.

<table>
<thead>
<tr>
<th>Categorization</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor critic-based algorithm [18,19]</td>
<td>More stable and robust in dynamic environments</td>
<td>The optimization problem needs to be resolved when the environment changes</td>
</tr>
<tr>
<td>DQN-based algorithm [20,21]</td>
<td>Suitable for dynamic environments</td>
<td>Expensive when the number of wireless devices grows exponentially</td>
</tr>
<tr>
<td>PPO [25–27]</td>
<td>Good overall performance; allows discrete and continuous control</td>
<td>Low sample efficiency; unsuitable for actual application scenarios</td>
</tr>
<tr>
<td>DDPG [28]</td>
<td>High efficiency</td>
<td>Insufficient discoverability, stability, and robustness</td>
</tr>
<tr>
<td>Soft actor critic-based algorithm</td>
<td>Robust in continuous-action spaces; better exploration capabilities due to entropy regularization</td>
<td>Might be more computationally intensive due to twin Q and actor networks</td>
</tr>
</tbody>
</table>

DQN: deep Q-network; PPO: proximal policy optimization; DDPG: deep deterministic policy gradient.

Regarding MCC servers, they possess significantly greater computing capacities than those of MEC servers and are well-equipped to manage peak user request demands. Therefore, task offloading can be effectively achieved through cooperation among MEC servers. Nonetheless, tasks with high computational complexity should be delegated to cloud servers. He et al. [30] explored a multi-layer task offloading framework within the MEC environment, facilitating collaboration between MCC and MEC and enabling task offloading to other SDs. Furthermore, Akhlaqi, M.Y. [31], pointed out that the increasing use of cloud services by devices has highlighted congestion problems in centralized clouds. This situation has prompted the emergence of multi-access edge computing (MEC) to decentralize processing. Chen, Y. et al. [32], and Mustafa, E. et al. [33], address offloading decisions in MEC systems. The former focuses on assessing the reliability of multi-media data from IoT devices using a game-based approach, while the latter introduces a reinforcement learning framework for making real-time computation task decisions in dynamic networks.

3. Problem Statement

The task offloading challenge can be depicted using a DAG, which is denoted as $\xi = (V, E)$, where the vertices symbolize individual tasks and the directed edges indicate task dependencies. Consequently, a subsequent task can only commence when the preceding task has been completed. It has been posited that every task can either be offloaded to an MEC server or processed locally on the SD of a user. When a task is offloaded to an MEC server, the following three distinct phases occur: sending, executing, and receiving. Conversely, for tasks executed locally, there is no data transmission between an SD and a MEC server.

The task set $\tau$ is represented as $N = \{1, 2, \ldots, n\}$, and the set of MEC servers is denoted as $M = \{1, 2, \ldots, m\}$. The offloading destination of each task is represented by $a_i$, which indicates whether a task is processed locally or on the specific MEC server to which it is offloaded. This modeling approach streamlines the intricacies of the offloading dilemma, furnishes a mathematical schema of the situation, and aids in determining the optimal resolution.

The devised framework was tailored to fine-tune offloading decisions while also considering the constraints of computing resources in a setting with multiple users and
multiple MEC servers. Such decisions can expedite the processing time of tasks and reduce the energy expenditure of the system. The objective function $J$ is elucidated in Equation (1):

$$J = \beta_1 \left( T_{\text{all loc}} - \sum_{i=1}^{N} T_{\text{ofl}} \right) + \beta_2 \left( E_{\text{all loc}} - \sum_{i=1}^{N} E_{\text{ofl}} \right),$$

where $T_{\text{all loc}}$ represents the total time required for all the tasks to be executed locally, $\sum_{i=1}^{N} T_{\text{ofl}}$ denotes the total time required for the offloaded tasks, $E_{\text{all loc}}$ is the energy consumed when tasks are processed locally, and $\sum_{i=1}^{N} E_{\text{ofl}}$ represents the total energy consumed during offloading. The objective of this function is to optimize both the time and energy efficiency through offloading. By adjusting the weights $\beta_1$ and $\beta_2$ the system can decide whether to prioritize time or energy efficiency. For instance, if the battery level is low, $\beta_2$ could be elevated to emphasize energy conservation. Ultimately, the overarching aim of this approach is to maximize objective function $J$.

$$\max_{A,F} J = \beta_1 \left( T_{\text{all loc}} - \sum_{i=1}^{N} T_{\text{ofl}} \right) + \beta_2 \left( E_{\text{all loc}} - \sum_{i=1}^{N} E_{\text{ofl}} \right)$$

where $A$ denotes the offloading decision for each task, vector $F$ signifies the amount of computing resources designated for each task, and $C_1$ and $C_2$ determine whether a task is processed locally or offloaded, respectively, to an MEC server. The tasks can be partitioned and processed concurrently across multiple locations. Regarding $C_3$, it ensures that the cumulative computing resources allocated to $\tau_i$ on the $m$-th MEC server do not exceed the total available computing resources for that particular server.

4. System Architecture

In this study, we introduced a task offloading strategy for a multi-server MEC in a multi-user setting with the primary goal of minimizing service delays and terminal energy consumption by directing each task to an appropriate MEC server under centralized control. The soft actor-critic task offloading scheme (SACTOS) is a DRL-based task offloading framework built on the MEC platform, as defined by the European Telecommunications Standards Institute (ETSI) [34].

The system model presented in this paper addresses scenarios with multiple MEC servers and multi-user applications, encompassing M MEC servers and N user tasks. The
data for the tasks to be offloaded are relayed between the MEC server and the SDs via a wireless communication link. Each device can either process the task locally or offload it to an MEC server. All the SDs are situated in an area where wireless connectivity is available. Given the finite computing capacity of an MEC server, there is a constraint on the number of offloading requests that it can concurrently handle. The notations forming the mathematical representation of this system model are provided in Table 2, and Table 3 summarizes the notations.

Table 2. Mathematical notations of the system model.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho_{\text{tran}}, \rho_{\text{recv}}$</td>
<td>Transmitted and received power</td>
</tr>
<tr>
<td>$\varphi$</td>
<td>Transmission probability of a sub-channel</td>
</tr>
<tr>
<td>$\xi$</td>
<td>DAG</td>
</tr>
<tr>
<td>$\omega$</td>
<td>Parameter that determines the transmission or recrue rate</td>
</tr>
<tr>
<td>$\pi(\cdot</td>
<td>\cdot)$</td>
</tr>
<tr>
<td>$\eta_i$</td>
<td>Number of clock cycles required to process each bit of data</td>
</tr>
<tr>
<td>$H$</td>
<td>Entropy</td>
</tr>
<tr>
<td>$B$</td>
<td>Wireless channel bandwidth</td>
</tr>
<tr>
<td>$\beta_1, \beta_2$</td>
<td>Weighted coefficient of energy consumption and latency ratio</td>
</tr>
<tr>
<td>$\theta_1, \theta_2$</td>
<td>Parameters of the SACTOS</td>
</tr>
<tr>
<td>$D$</td>
<td>Replay buffer</td>
</tr>
</tbody>
</table>

DAG: Directed acyclic graph; SACTOS: soft actor–critic task offloading scheme.

Table 3. Mathematical notations of task offloading.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>Collection of MEC servers</td>
</tr>
<tr>
<td>$a_i$</td>
<td>Offloading action of task $\tau_i$</td>
</tr>
<tr>
<td>$T_{\text{loc}}, T_{\text{mec}}$</td>
<td>Local and MEC server computation latency of task $\tau_i$</td>
</tr>
<tr>
<td>$f_{\text{loc}}, f_{\text{mec}}$</td>
<td>CPU clock speed of the SD where task $\tau_i$ is located</td>
</tr>
<tr>
<td>$R_i(\varphi)$</td>
<td>Transmission rate of the receiving rate</td>
</tr>
<tr>
<td>$d_i$</td>
<td>Data size of task $\tau_i$</td>
</tr>
<tr>
<td>$\tau_{\text{complete}}$</td>
<td>Time taken to complete task $\tau_i$</td>
</tr>
<tr>
<td>$\tau_{\text{tran_complete}}$</td>
<td>Time taken to upload and download task $\tau_i$</td>
</tr>
<tr>
<td>$E_{\text{loc}}$</td>
<td>Energy consumed by a local computing model for task $\tau_i$</td>
</tr>
<tr>
<td>$E_{\text{mec}}$</td>
<td>Energy consumed in computational offloading for task $\tau_i$</td>
</tr>
<tr>
<td>$E_{\text{total}}$</td>
<td>Total energy consumed in computational offloading for task $\tau_i$</td>
</tr>
<tr>
<td>$\beta_1, \beta_2$</td>
<td>Weighting coefficients</td>
</tr>
</tbody>
</table>

CPU: Central processing unit; SD: smart device.

4.1. MEC Architecture

The proposed system architecture, which is shown in Figure 1, incorporates both an SD and a graph parser (GP) at the user device level. User requests take the form of a graph that encapsulates the dependencies among the tasks. The GP evaluates the dependent tasks to determine the feasibility of offloading.
4.1. MEC Architecture

The proposed system architecture, which is shown in Figure 1, incorporates both an SD. The trained neural network of the device determines the offloading decision through forward propagation. Based on this decision, the task is either processed by MEC or executed locally. Such an architecture ensures the streamlined processing of user requests, graph-centric task distribution, and resource optimization via offloading. Continuously refining the algorithm through DRL will contribute to consistent enhancements in system performance. This approach is intended to efficiently utilize resources in a smart-device-centric MEC system.

4.2. System Model

Different SDs execute different tasks, each of which results in various delays. In this study, we considered two scenarios: the local execution of the current task and its offloading to the MEC server. Our strategy determines offloading decisions for tasks with dependencies, meaning that the completion time of the preceding tasks influences subsequent processing. This sub-section provides the definitions for computation delay, transmission delay, and task completion time. First, the computational delay for both local and MEC servers, as shown in Equation (3):

\[
T_{loc}^{\tau_i} = \frac{\chi_i(d_i \times \eta_i)}{f_i}, \quad T_{mec}^{\tau_i} = \frac{(1 - \chi_i)(d_i \times \eta_i)}{f_i},
\]

where \(T_{loc}^{\tau_i}\) denotes the local computation delay of task \(\tau_i\); \(T_{mec}^{\tau_i}\) signifies the computation delay of the task when processed by the MEC server; and \(f_i\) represents the CPU clock rates of the MEC server and the SD, respectively, with \(m\) and \(n\) as identifiers. For scenarios involving partial offloading, a portion of the task is processed locally, and the remainder is offloaded to the MEC server. The weight \(\chi_i\) is introduced to quantify this distribution. This weight, which ranges from 0 to 1, indicates the fraction of data processed locally for task \(\tau_i\). Conversely, \((1 - \chi_i)\) represents the proportion processed by the MEC server. Moreover, \(\tau_i\) refers to the number of clock cycles necessary to process each data bit of the task, and \(\eta_i\) denotes the total number of clock cycles required for the entire task \(\tau_i\). When addressing SDs, Shannon’s theory, which is delineated in Equation (4), needs...
to be employed before computing the data offloading delay. This equation is crucial for estimating the maximum channel transmission rate.

\[
R_i(\varphi) = B \times \log_2\left\{1 + \frac{[\varphi \rho_{\text{tr}} + (1 - \varphi) \rho_{\text{rec}}] \times \theta^2}{\varphi^2} \right\},
\]

where \(B\) denotes the radio channel bandwidth between the SD and the MEC server, signifying the bandwidth available for use during wireless communication; and the variable \(\varphi\) represents the transmission probability of each sub-channel and is characterized as a discrete random variable with the possible values \(\varphi \in \{0, 1\}\). When \(\varphi = 1\) and 0, \(R_i(\varphi)\) represents the data transmission and reception rates, respectively. This suggests that the data rate for a given sub-channel can fluctuate depending on whether data are transmitted or received. The term \(\theta^2\) designates the wireless channel gain between the SD and the BS at the time slot \(t\); \(\varphi^2\) represents the noise power; the variables \(\rho_{\text{tr}}\) and \(\rho_{\text{rec}}\) correspond to the power expended for data transmission and reception, respectively; \(\rho_{\text{tr}}\) is the transmission power utilized to relay data from an SD to an MEC server; and \(\rho_{\text{rec}}\) is the reception power employed by an SD to obtain data from an MEC server. The latency associated with data transmission is delineated according to Equation (5):

\[
T_{\text{tran}}^i(\varphi) = \frac{d_i}{R_i(\varphi)},
\]

where \(d_i\) denotes the data size of task \(\tau_i\), \(R_i(\varphi)\) represents the data transmission rate during data transmission, and \(\varphi\) is a variable accounting for various parameters or conditions that may influence the transmission speed, enabling the requisite time for data transmission to be calculated.

Given the constraints of limited computing resources, we postulate that an SD cannot finalize a computing task within a designated timeframe, necessitating offloading in task processing. Under these circumstances, the agent scrutinizes the task and decides on its execution within the MEC server to minimize latency. Beyond merely determining the processing location for the task, the agent also assists in selecting the most suitable MEC server based on the requirements of the SD. The cumulative latency when offloading a task to an MEC server is significantly influenced by computational and uplink transmission latency [31]. The criteria for task processing are expressed in Equation (6):

\[
T_i^{\text{complete}} = \begin{cases} 
T_i^{\text{loc}} & \text{if } a_k = 0 \\
T_i^{\text{tran}} & \text{if } a_k \in 1, 2, \ldots, |M|
\end{cases}
\]

where \(T_i^{\text{complete}}\) denotes the completion time of task \(\tau_i\); \(T_i^{\text{loc}}\) represents the processing time of the task when executed locally; \(T_i^{\text{tran}}\) represents the processing duration of a task that has been offloaded to an MEC server; and \(a_k\) denotes the offloading location of the task, where \(a_k = 0\) if the task is processed locally and otherwise indicates the designated number of MEC servers from which the task is offloaded. Subsequently, the completion time of task \(\tau_i\), i.e., \(T_i^{\text{complete}}\), is the summation of the local processing time and waiting period before initiation, as described by Equation (7):

\[
T_i^{\text{complete}} = T_i^{\text{loc}} + T_i^{\text{loc\_wait}}
= T_i^{\text{loc}} + \max_{(\tau_j, \tau_i) \in \varepsilon} \left(T_j^{\text{loc\_complete}}, T_j^{\text{loc}}\right),
\]

where \(T_i^{\text{loc\_wait}}\) represents the greater of two values: the local execution time \(T_i^{\text{loc}}\) of task \(\tau_i\) and the longest completion time among all the preceding tasks \(\tau_j\). It is ensured that \(\tau_i\) does not commence until all preceding tasks have been completed. The offloading of a task to the MEC server involves three primary stages that capture the time required to relay it to
the server. The completion time during this stage is the aggregate of the upload time and the waiting period for the upload. Notably, the waiting time for the upload is contingent on the larger value between the maximum completion times of the preceding tasks and the local completion time of the current task. This phase involves the time required to process a task on an MEC server, with the duration varying depending on the complexity of the task and the computational capacity of the MEC server. This phase considers the time required to retrieve the processed task results from the MEC server. The associated completion time is expressed in Equation (8):

\[
T_{i}^{\text{tran,complete}} = T_{i}^{\text{tran}} + T_{i}^{\text{tran,wait}} \\
= T_{i}^{\text{tran}} + \max\left(T_{i}^{\text{loc}}, \max_{(\tau_{j}, \tau_{k}) \in \mathcal{E}} T_{j}^{\text{complete}}\right).
\]

In the execution phase of an MEC server, two primary scenarios arise regarding the previous state of a task. The first scenario involves tasks for which the transfer and upload phases have already been completed, and the second scenario involves tasks executed on the same MEC server as the current task. Based on this, the completion time for the previous task is defined in Equation (9):

\[
T_{i}^{\text{pre,complete}} = \begin{cases} T_{i}^{\text{complete}} & \text{if } (\tau_{j}, \tau_{k}) \in \mathcal{E} \text{ and they on the same MEC server} \\ T_{i}^{\text{tran}} & \text{if task has completed the transmission phase} \end{cases}.
\]

The total time to complete a task on an MEC server is the cumulative sum of the execution and waiting times. The waiting time encompasses both the quickest task execution time specific to an MEC server and the waiting duration for the completion of the preceding task. This waiting time is influenced by the completion times of other tasks on the same MEC server, and it is defined in Equation (10):

\[
T_{i,m}^{\text{MEC}} = T_{i,m}^{\text{MEC,exec}} + T_{i,m}^{\text{MEC,wait}} \\
= T_{i,m}^{\text{MEC,exec}} + \max\left(T_{i,m}^{\text{MEC,wait}}, T_{i,m}^{\text{complete,pre}}\right).
\]

Regarding the download and reception processes, the completion time comprises two parts: the completion time on the MEC server from the preceding step of the task and the quickest available download link. The download-completion time of a task that is processed on an MEC server is defined in Equation (11):

\[
T_{i,m}^{\text{down}} = T_{i,m}^{\text{tran}} + T_{i,m}^{\text{down,wait}} \\
= T_{i,m}^{\text{tran}} + \max\left(T_{i,m}^{\text{down,avail}}, T_{i,m}^{\text{complete,MEC}}\right),
\]

where \(T_{i,m}^{\text{down}}\) denotes the completion time of the download and reception phase for task \(\tau_{i}\), \(T_{i,m}^{\text{down,avail}}\) indicates the quickest available download link time, and \(T_{i,m}^{\text{complete,MEC}}\) represents the processing completion time on MEC server \(m\).

Energy consumption is a critical factor for SDs. The overall energy consumption for a task can be categorized into two components: computation and transmission. If a task operates locally, only computational energy costs come into play because no transmission is involved. Conversely, when a task is offloaded to an MEC server, an SD incurs transmission energy costs for both the upload and download. Hence, the cumulative energy consumption of an SD can be expressed as follows:

\[
E_{i,j}^{\text{total}} = \sum_{\tau_{i} \in \mathcal{V}, \alpha_{j} = 0} E_{i}^{\text{loc}} + \sum_{\tau_{i} \in \text{Task}, \alpha_{j} = \{1, 2, \ldots, m\}} E_{i}^{\text{tran}} \\
= \sum_{\tau_{i} \in \text{Task}, \alpha_{j} = 0} K_{n} \times \left(j_{n}^{\text{loc}}\right)^{2} \times T_{i}^{\text{loc}} + \sum_{\tau_{i} \in \text{Task}, \alpha_{j} = \{1, 2, \ldots, m\}} \left(P_{\text{tran}} \times T_{i}^{\text{up}} + P_{\text{recov}} \times T_{i}^{\text{down}}\right),
\]

\(E_{i}^{\text{loc}}\) denotes the energy consumption of the local computing process. The energy consumption of the transmission phase is divided into two parts: the transmission time and the time it takes to complete the transmission. The energy consumption of the upload phase is expressed as follows:

\[
P_{\text{tran}} \times T_{i}^{\text{up}}
\]

and the energy consumption of the download phase is expressed as follows:

\[
P_{\text{recov}} \times T_{i}^{\text{down}}
\]
where \( K_u \) represents the energy constant \([35]\), \( f_{loc}^{\text{loc}} \) is the clock frequency of the local processor, \( T_{loc}^{\text{loc}} \) denotes the time required to process a task locally, \( P_{\text{tran}} \) is the power utilized during transmission, \( T_{\text{up}}^{\text{loc}} \) indicates the upload time, \( P_{\text{recv}} \) is the power consumed during reception, and \( T_{\text{down}}^{\text{loc}} \) is the download time.

5. Proposed Task Offloading Scheme

This section discusses the implementation methodology of the proposed SACTOS. First, the SAC algorithm, which manages a continuous action space, is introduced. Subsequently, an offloading model that focuses on the optimization problem is examined. Finally, the steps involved in implementing the algorithm are involved.

5.1. SAC Algorithm for Continuous Actions

The SAC algorithm is a DRL algorithm that leverages the maximum entropy in a continuous action space. While most online algorithms estimate gradients using new samples at each iteration, this algorithm is different in that it reuses past experiences. Grounded in a maximum entropy framework, it aims to maximize action diversity while also maximizing the expected reward. Consequently, in terms of discoverability, stability, and robustness in continuous action spaces, SAC tends to outperform deterministic policies \([35]\).

Typically, SAC is tailored to continuous action spaces. This continuity facilitates nuanced action choices within search spaces, thereby paving the way for agile responses under various environmental conditions. The primary distinction between continuous and discrete actions under SAC is their respective outputs and representations. In the continuous action space of SAC, the policy \( \pi_{\Phi}(a_t, s_t) \) is represented as a density function, whereas in a discrete action space, it is represented as a probability. Regarding entropy, it represents the uncertainty in a random variable \([36]\). The entropy value of policy \( \pi_{\Phi}(a_t, s_t) \) is determined using Equation (13):

\[
H(\pi(\cdot|s_t)) = E[-\log \pi(\cdot|s_t)],
\]

(13)

where \( E(\cdot) \) is the expected value, \( -\log \pi(\cdot|s_t) \) represents the log probability of an action in state \( s_t \), and the inverted expected value of this log probability is the entropy. Equation (14) determines the policy that maximizes the balance between the expected reward \( R(s_t, a_t) \) and the policy entropy \( H \) \([36]\):

\[
\pi^* = \arg \max_{\pi} E_{(s_t,a_t) \sim \rho_{\pi}} \left[ \sum_t R(s_t, a_t) - \alpha H(\cdot|s_t) \right],
\]

(14)

where \( E_{(s_t,a_t) \sim \rho_{\pi}} \) denotes the expected value of the state-action pair \( (s_t, a_t) \) as determined by the policy \( \pi \), and the coefficient \( \alpha \) modulates the balance between reward and entropy. This formula aims to maximize the expected return in each state while preserving the diversity or exploratory nature of the policies. Moreover, instead of fixing \( \alpha \) as a static hyperparameter, it can be adaptively fine-tuned using a neural network by backpropagating the error in the entropy normalization coefficient \([36]\). The objective of the entropy normalization coefficient is given by Equation (15):

\[
J(\alpha) = \pi_t(s_t)^T \left[ -\alpha \left( \log(\pi_t(s_t) + \underline{H}) \right) \right],
\]

(15)

where \( \pi_t(s_t) \) represents the probability distribution for state \( s_t \) at time \( t \); and \( -\alpha \left( \log(\pi_t(s_t) + \underline{H}) \right) \) represents the log probability of a policy, where \( \underline{H} \) represents the target entropy. The disparity between these two values influences the increase or decrease in entropy. The accumulated reward for entropy in state \( s_t \) is characterized by a soft state value function. Owing to the continuous nature of the task set, estimating this value
requires intricate computations. To address this issue, value estimation is performed using sampling techniques, such as the Monte Carlo technique; it is defined in Equation (16):

\[
V(s_t) := \pi_t(s_t)^T [Q(s_t) - a \log (\pi_t(s_t))]^2.
\] (16)

This equation integrates both the anticipated value of future rewards and the entropy of the policy through a soft-state value function. Given the continuous nature of tasks, it is possible to represent actions in intricate environments. This incorporation permits a richer inclusion of information and fosters the application of exploratory policies.

\[
Q_\theta(s_t, a_t) = r(s_t, a_t) + \gamma E_{s_{t+1} \sim \rho} \left[ \max_a (Q(s_{t+1}, a) - a \log (a|s_{t+1})) \right] + \alpha E_{\pi \sim \rho} \left[ \log \pi(a|s_{t+1}) \right].
\] (17)

At this juncture, incorporating the entropy normalization term reflects the uncertainty in action selection and promotes the use of explorative policies. The approach for maximizing the objective function using the soft policy gradient method is defined as follows (refer to Appendix A for details):

\[
J(\theta) = E_{(s_t, a_t) \sim D} [\alpha \log (\pi(a|s_t)) + Q_\theta(s_t, a_t)].
\] (18)

This method employs a Q-function to estimate the value of an action and promotes exploration using an entropy term. Gradient-based optimization methods are utilized to maximize the objective function, as follows (refer to Appendix B for details):

\[
\mathcal{L}(\theta_t) = E_{(s_t, a_t) \sim D} [Q_\theta(s_t, a_t) - r(s_t, a_t)] + \alpha E_{\pi \sim \rho} \left[ \max_a (Q_\theta(s_{t+1}, a) - a \log (a|s_{t+1}))^2 \right].
\] (19)

This equation denotes the loss function of the Q function. By employing gradient descent to minimize this loss function, a gradient akin to the aforementioned estimated gradient can be obtained [36].

\[
J(\Phi) = E_{(s_t) \sim D} \left[ \pi_t(s_t)^T (\alpha \log (\pi_t(s_t)) - Q_\Phi(s_t)) \right].
\] (20)

When SAC is applied to continuous action spaces, the reparameterization technique is utilized to minimize the policy loss \(J(\Phi)\). This method was introduced to address the gradient instability arising from stochastic behavioral sampling in the continuous action space of SAC. The reparameterization technique stabilizes the gradient calculations by decoupling the generation of a stochastic action from an independent noise variable; owing to this, SAC demonstrates superior performance even in intricate environments.

5.2. Markov Decision Process

Markov prediction models are based on memory-less, discrete-time stochastic processes known as Markov chains [37]. The essential feature of Markov chains is their memory-less nature, meaning that the next state relies solely on the current state and not on the sequence of states that occurred before it. This aspect is especially relevant in the context of task offloading. Here, the choice to offload a specific task does not necessarily depend on the historical sequence of tasks but rather on the current system state and the task’s inherent characteristics.

Given these properties, we recognized that the task offloading challenge closely aligns with the dynamics portrayed by Markov chains. To systematically and efficiently tackle this challenge, we framed it as an MDP characterized by states (s), actions (a), and rewards (r). This approach empowered us to leverage the structured decision-making abilities of MDPs, enabling strategic offloading of decisions grounded in current states to maximize cumulative future rewards. The specific definitions of the state space, action space, and reward function for the MDP are elaborated below.
The decision to use MDP offers a structured framework for making sequential decisions amidst uncertainties, aligning seamlessly with the stochastic nature of the task offloading problem. It establishes a systematic approach to determining the optimal offloading strategy, considering not only the immediate reward but also the long-term cumulative reward.

5.2.1. State Space

A state space is defined as a combination of the task DAG and the associated offloading decisions. Regarding the parameters in a state space, \( \Omega_1 \) denotes the offloading choices from the initial task up to the current one; and \( \zeta \), which comprises five vectors \([P, Q, U, S, \text{and } T]\), encapsulates the encoding of a DAG. Here, \( P \) represents the characteristics of a task and its current state, containing information such as the type of task, estimated execution time, required resources, and estimated time to completion; \( Q \) is a metric that indicates the priority or importance of a task; the vector \( U \) reflects the current state of system resources, including information such as CPU utilization, memory status, input/output (I/O) latency, and network latency; \( S \) represents the current status of the offload queue and includes data such as the length of the queue, the average waiting time, and the number of recently offloaded jobs; and the vector \( T \) represents the information related to the task load of the system, including the average processing time, processing speed, and failure rate of the offloaded tasks. Therefore, the state space can be expressed as:

\[
M' := m' \big| m' = (\sigma, O_1 : j, R_1 : k). \quad (21)
\]

5.2.2. Action Space

Tasks can be executed locally on a device or offloaded to an MEC server, with each choice influenced by various factors. Local execution may offer faster response times but can be limited by resource constraints, rendering it unsuitable for complex operations. In contrast, offloading to an MEC server relieves the local device’s burden and leverages greater computing capabilities. However, this approach can come with potential drawbacks, such as network delays or data transmission costs.

The task’s location is determined by the variable \( a_i \). If \( a_i = 0 \), the task is executed locally. Conversely, any value other than zero identifies an MEC server, indicating the task should be processed there. Different MEC servers may have distinct performance metrics and available resources. Consequently, the offloading decision must consider multiple factors, such as server status, task requirements, and network conditions. The action space, denoted as \( A = \{0, 1, 2, \ldots, m\} \), with \( m \) representing the number of accessible MEC servers and 0 indicating local execution, plays a crucial role. To ensure optimal performance, this action space requires dynamic management through the integration of diverse offloading strategies and optimization methods.

5.2.3. Reward Function

Several strategies can improve QoS, with adjustments based on key variables. Notably, system latency and energy consumption hold significant importance. In our study, we introduced a more intuitive and transparent reward function. We refined the formula calculations to precisely capture real-time changes in both delay and energy consumption at each step.

In existing strategies, the reward function primarily focuses on system delay and energy consumption as key factors for optimizing QoS. However, this approach is both computationally demanding and fails to capture real-time changes in delay and energy consumption. To address this, we integrated increments that accurately represent variations in actual delay and energy consumption at every stage.

\[
\Delta T_{off}(N) = T_{off}(N) - T_{off}(N - 1), \quad (22)
\]
\[ \Delta E_{ofl} = E_{ofl}(N) - E_{ofl}(N-1), \] (23)

where \( \Delta T_{ofl} \) denotes the variation in system delay between successive time intervals, and \( \Delta E_{ofl} \) signifies the alteration in energy consumption over the same period. These incremental changes are pivotal in promptly discerning shifts in the performance and efficacy of a system and serve as central metrics in both the optimization and evaluation processes.

The reward function imposes penalties on incremental delays and increases energy consumption at each step. By normalizing these values against the overall local delay and energy consumption, the function ensures a balance between penalizing inefficiencies and assessing system performance; it is defined as follows:

\[ R(s_i, a_i) = \beta_1 \times \left( \frac{\Delta T_{ofl}}{T_{all}^{loc}} \right) + \beta_2 \times \left( \frac{\Delta E_{ofl}}{E_{all}^{loc}} \right), \] (24)

where \( \beta_1 \times \left( \frac{\Delta T_{ofl}}{T_{all}^{loc}} \right) \) normalizes the increase in delay relative to the overall local delay, and \( \beta_2 \times \left( \frac{\Delta E_{ofl}}{E_{all}^{loc}} \right) \) normalizes the increment of energy consumption relative to the total local energy consumption. Both elements are structured to mirror the efficiency and responsiveness of the system.

5.2.4. Design of SACTOS

The offloading process comprises three steps. In step 1, tasks within a DAG are topologically sorted and then arranged in descending order based on their rank values, resulting in the following sequence:

\[ \text{Score}(v_i) = \begin{cases} T_{total_i} & \text{if } v_i \in \text{exit} \\ \max_{(v_i, v_j) \in \epsilon} \left( \text{Score}(v_j) + T_{total_i}, v_i \right) & \text{otherwise} \end{cases}, \] (25)

where \( T_{total_i} \) represents the total time required to complete processing within the defined set. In step 2, the tasks are transformed into a sequence of vectors that serve as inputs to the neural network. During this transformation, feedback, such as network and device status (e.g., battery level and CPU usage) is incorporated to enhance offloading efficiency. In step 3, the decision to offload a task is made by selecting the most likely offloading action based on probability. Then, MEC servers collaborate to complete selected tasks in accordance with this action. Moreover, potential overloads are distributed through cooperation between multiple MEC servers or SDs, thereby optimizing resource utilization. The comprehensive procedure for the proposed SACTOS algorithm is detailed in Algorithm 1.

The SACTOS algorithm optimizes task offloading for the MEC server. After initializing the neural network and memory buffers, it prioritizes tasks and transforms them into neural network inputs. The algorithm continually selects and executes tasks based on the offloading probability, ensuring collaboration between the MEC server and SD to accomplish these tasks.
Algorithm 1: SACTOS

**Input:** Episode, environment, batch size, replay buffer size

1. Initialize neural network parameters $\theta$
2. Initialize Q-function with random weights $\theta_Q$
3. Initialize policy $\pi$ with random weights $\theta_\pi$
4. Initialize replay buffer: $R$

for each episode do:

6. Topological sorting of the DAG:
7. Sort tasks in descending order based on their priority values (computed based on $T_{total}$)
8. Convert tasks to a neural network input:

for each task in DAG do:

10. Convert the task to a vector sequence
11. Incorporate feedback from the network and device state (e.g., battery level, CPU usage)

end for

14. Offloading the decision:
15. while task not done do:
17. Select action $a$ with the highest offloading probability from policy $\pi$ given the current state $s$ using:
18. $a \sim \pi(. | s; \theta)$
19. Execute action $a$ and observe the next state $s'$ and reward $r$
20. Store $(s, a, r, s')$ in $R$
21. Sample a random mini-batch from $R$
22. Update the Q-function using the SAC loss
23. Update policy $\pi$ using the SAC objective
24. Update the target networks

end while

26. Collaborate between the smart MEC server and selected task based on action $a$ to complete the task.
27. Distribute the load and optimize the resource usage among multiple MEC hosts or SDs.

end for

In Figure 2, we present the detailed structure of the SAC scheme, a cutting-edge deep reinforcement learning algorithm known for its adaptability and efficiency. The SAC architecture uniquely combines an off-policy method with the advantages of actor-critic paradigms and a maximum entropy framework. At its core, the actor network generates actions based on the input state. Unlike deterministic strategies, SAC’s actor produces actions from a stochastic policy, facilitating a comprehensive exploration of potential actions. SAC stands out with its dual-critic system. Each critic network independently predicts the Q-value for a given state-action pair, mitigating overestimation biases—common in deep Q-learning algorithms. For policy optimization, SAC uses the smaller Q-value from its two critics, ensuring a cautious and robust policy that accounts for the variability inherent in single-critic evaluations. SAC’s distinctive feature is its entropy regularization, which encourages extensive exploration by introducing an entropy term to the reward. This nudges the actor towards more exploratory action choices. Lastly, the critics are updated using a ‘soft’ Bellman equation, which blends estimated and target Q-values, greatly enhancing learning stability. Overall, Figure 2 provides a comprehensive visual overview of SAC’s intricate architecture, showcasing its ability to derive optimal policies in complex environments by effectively balancing exploration and exploitation.
6. Performance Evaluation

This section discusses the experimental results of SACTOS and the methodology used to assess its performance. First, various parameters employed in the DAG generator are presented. Subsequently, the simulation environment and hyperparameters are described. Subsequently, the analysis of the average reward value to gauge the convergence attributes of SACTOS is presented. Finally, a comparison of SACTOS with four other benchmark algorithms, meant to highlight its efficacy, is presented.

6.1. Fundamental Approaches

This section describes the assessment of the performance of the proposed offloading scheme under various parameter settings to validate both its effectiveness and convergence. The following four offloading schemes were evaluated:

- Local-only: All the computational tasks were performed exclusively on local devices.
- Random: Tasks were offloaded based on a random selection [38].
- The PPO algorithm is an advancement of the policy gradient method; however, it still has constraints in terms of sampling efficiency [39].
- Dueling double (D3QN) deep-Q network-based scheme: This method combines the strengths of both the dueling DQN and double DQN, further enhancing the learning algorithm and structure of the conventional DQN.

6.2. Simulation and Results

Although real-world application programs can be depicted using DAGs with diverse topologies, currently available datasets provide limited application information [34]. To implement the proposed offloading method, a graph theory-based approach is required, even for applications for which the topological information is not present in an actual dataset. By analyzing the interactions and data flows between applications, a graph theory-based approach can facilitate more efficient offloading decisions.

For the simulation experiment, we considered both the channel gain and transmission rate based on the distance between the SD and the MEC server. The baseline transmission rate was assumed to be 100 Mbps.

The CPU clock frequency of the local device, i.e., $f_{loc}^{loc}$, was set to 2 GHz, while that of the MEC server, i.e., $f_{mecc}^{mecc}$, was set to 8 GHz. The transmission and reception powers, $P_{tran}$ and $P_{recv}$, respectively, were set to 2.5 and 1.8 W, respectively. The task sizes ranged from 10 to 50 KB. Additionally, the number of clock cycles required for a single task ranged between $10^7$ and $10^9$ cycle/sec. Each agent consisted of an actor and a critic. Both contained...
two hidden, fully connected layers, each with 256 neurons. The essential hyperparameters for SAC implementation are listed in Table 4.

Table 4. Hyperparameters for algorithm implementation.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replay memory size</td>
<td>5000</td>
</tr>
<tr>
<td>Minibatch size</td>
<td>128</td>
</tr>
<tr>
<td>Discounted factor</td>
<td>0.99</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Adam</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.001</td>
</tr>
<tr>
<td>Initial value of the temperature parameter</td>
<td>0.2</td>
</tr>
<tr>
<td>Learning rate of the temperature parameter</td>
<td>0.001</td>
</tr>
<tr>
<td>Update interval of the target network</td>
<td>1000</td>
</tr>
<tr>
<td>Epsilon decay rate</td>
<td>0.99</td>
</tr>
</tbody>
</table>

The results depicted in Figure 3 show that the rapid convergence and impressive average reward of SAC are immediately noticeable. While the initial performances of all the algorithms appeared comparable, SAC began to gradually distinguish itself via superior performance enhancements over time. Both D3QN and PPO exhibited faster convergence rates than that of SAC in the initial stages, but SAC surpassed them at a convergence speed midway through the experiments. This trend highlighted the more efficient learning strategy of SAC in complex offloading scenarios compared to the strategies of the other two methods. Regarding D3QN, its performance steadily enhanced, ultimately settling at a reward lower than that of SAC and PPO. On the other hand, PPO displayed a rapid improvement in the initial phase but exhibited a decelerated growth rate in the later stages, finally resulting in a performance less than that of SAC. These findings underscored the effectiveness of SAC as a primary reinforcement learning algorithm in handling intricate offloading settings.

Figure 3. Convergence analysis. D3QN: Dueling double DQN.

Figure 4 presents a comparison of the time and energy conservation scales (TECS) across the four offloading methods as the task count increased. The TECS quantifies the reduction in local task execution costs relative to the total post-offloading costs, effectively highlighting the trade-off between latency and energy consumption. For the experiment, the aspect ratio of the graph was fixed at 0.45, three MEC servers were considered, and a communication-to-computation ratio (CCR) of 0.5 was used. Compared to the improvements exhibited by PPOS, D3QNS, and Random, SACTOS exhibited an
average improvement in TECS of 2.24, 52.52, and 74.25%, respectively. This indicated the superior optimization of SACTOS for offloading tasks. Although an increase in task count invariably led to an increase in delay and energy consumption, the gradual increases in post-offloading costs led to higher TECS values for SACTOS.

Figures 5 and 6 provide an in-depth view of the service waiting time and energy consumption for each offloading strategy, influenced by the progressive increase in the number of tasks. As the workload increased, we observed a corresponding rise in service waiting time and energy consumption across all strategies. This consistent increase highlights the inherent challenges associated with managing higher workloads in offloading systems. Notably, the ‘local-only’ strategy exhibited a significant spike in both latency and energy consumption. Such a rapid increase can be interpreted as indicating inherent scalability limitations in this strategy, possibly due to the absence of an adaptive mechanism for efficiently managing and distributing growing tasks.

![Figure 4. Impact of the number of tasks on the time and energy conservation scales (TECS).](image1)

![Figure 5. Impact of the number of tasks on the latency.](image2)
On the other end of the spectrum, DRL-based offloading strategies such as PPO, D3QN, and SACTOS demonstrated a relatively stable trend. Their consistent performance, even as the number of tasks increased, suggests their ability to effectively balance the dual objectives of conserving energy and reducing waiting time. Among these strategies, SACTOS emerged as a top performer. It significantly reduced standby time by an average range of 1.6–62.19% and demonstrated commendable energy efficiency, reducing consumption by 6.7–88.23%. These statistics highlight SACTOS’s robust learning mechanism, which continuously refines its offloading decisions in response to changes in task volumes.

It is important to note the growth pattern exhibited by the DRL-based strategies. Unlike the sudden increase observed with the ‘local-only’ approach, these strategies demonstrated a more gradual growth. This pattern suggests their capacity to adapt and allocate resources strategically, preventing spikes in demand from causing disproportionate rises in waiting times or energy expenses.

Essentially, while all strategies encountered challenges as the number of tasks increased, the DRL-based approaches, particularly SACTOS, demonstrated superior capabilities in handling these challenges. Their performance not only surpassed that of other strategies, but also highlighted the significance of adaptive learning in optimizing offloading decisions, thereby reducing both latency and energy consumption.

Figure 7 illustrates the variation in the TECS with respect to the CCR. The figure clearly shows that the optimization performance of SACTOS increased when the CCR was low, indicating the dominance of computational tasks. These results also highlighted the effectiveness of SACTOS in environments where the workload was more computationally intensive. Moreover, SACTOS outperformed the other methods, exhibiting a performance that was 6.36, 40.2, and 48.6% superior to that of PPO, D3QN, and random, respectively. Regarding PPO, D3QN, and random, they displayed inconsistent performance shifts with varying CCRs. This observation underscored the necessity for an offloading strategy adept at handling the surges in latency and energy expenditure associated with localized computing in a computation-centric setting. The exceptional efficiency of SACTOS is pivotal, especially in intricate scenarios where a proper balance between communication and computation is essential.
To adeptly navigate the ever-changing MEC landscape, we incorporated a collaborative architecture among MECs, framing it within an MDP context. We articulated the dependent tasks using a DAG and honed the MEC system using a DRL. The proposed SACTOS, which operates under centralized control, significantly reduces service delays and conserves energy. The experimental outcomes confirmed that SACTOS not only converged swiftly and stably but also outperformed existing methodologies such as local-only, random, D3QN, and PPO.

Currently, our future research trajectory points toward the integration of multi-agent reinforcement learning. Our goal is to increase task offloading efficiency by judiciously tapping into the shared resource reservoir inherent in the MEC setup, wherein each SD is perceived as an autonomous agent, and to explore the synergies of multi-agent collaboration to reduce costs and increase the QoS values of systems.

Author Contributions: Conceptualization, D.L.; Methodology, D.L. and I.J.; Software, D.L.; Validation, D.L.; Formal Analysis, D.L.; Investigation, D.L.; Resources, I.J.; Data Curation, D.L. and I.J.; Writing—Original Draft Preparation, D.L.; Writing—Review and Editing, D.L. and I.J.; Visualization, D.L.; Supervision, I.J.; Project Administration, I.J.; Funding Acquisition, I.J. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Institute of Information & communication Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2020-0-00107, Development of the technology to automate the recommendation for big data analytic models that define data characteristics and problems).

Data Availability Statement: The data that support the findings of this study are available on request from the corresponding author, I.J. The data are not publicly available because they include information that could compromise the privacy of the study participants.

Conflicts of Interest: The authors declare no conflict of interest.
Abbreviations
The following abbreviations are used in this manuscript:

AR | Augmented Reality
BS | Base Station
CRRM | Collaborative Radio Resource Management
CRSP | Collaborative Signal Processing
DAG | Directed Acyclic Graphs
D3QN | Dueling Double Deep-Q Network
DRL | Deep Reinforcement Learning
IoT | Internet of Things
MR | Mixed Reality
PPO | Proximal Policy Optimization
SAC | Soft Actor-Critic
SD | Smart Device
TECS | Time and Energy Conservation Scale
MDP | Markov Decision Process
MCC | Mobile Cloud Computing
MEC | Multi-Access Edge Computing
VR | Virtual Reality

Appendix A
It seems to represent an objective function within the context of entropy-regularized reinforcement learning. Specifically, it combines the expected Q-value $Q(\theta)_{st,at}$ with an entropy term $\alpha \log \pi(a_{t}|s_{t})$, where $\alpha$ serves as a temperature coefficient. This combination aims to promote effective exploration of the state-action space while optimizing expected rewards. However, this objective is derived from the maximum entropy reinforcement learning framework, where the goal is to find a policy that maximizes both expected return and entropy. In traditional reinforcement learning (RL), the standard objective is to maximize the expected return, often represented by the Q-value: $E(\pi_{\theta})_{st,at}[Q(\theta)_{st,at}]$. To encourage more exploration in the state-action space, an entropy-regularization term can be introduced. This entropy term, denoted as $H(\pi) = -E_{\pi}[\log \pi(a)]$, measures the randomness of a policy distribution $\pi$, thus incentivizing a more exploratory policy. By combining these objectives, we derive the entropy-regularized objective $J(\theta) = E(\pi_{\theta})_{st,at}[\alpha \log \pi(a_{t}|s_{t}) + Q(\theta)_{st,at}]$. This composite objective ensures that the policy strikes a balance between efficient exploration (governed by the entropy term) and leveraging learned strategies (represented by the Q-value), with the coefficient $\alpha$ serving as the balancing factor.

Appendix B
The function $L_{Q}(\theta_{i})$ represents the loss function of the Q-network parameterized by $\theta_{i}$ and its main objective is to minimize the difference between the predicted and target Q-values.

- **Predicted Q-value**

  $Q_{\theta}(s_{t},a_{t})$

  This term represents the predicted Q-value for the current state-action pair $s_{t},a_{t}$ using the current Q-network parameters $\theta$.

- **Target Q-value**

  $r(s_{t},a_{t}) + \gamma E(\pi)_{s_{t+1}}[\min(Q_{\theta_{1}}(s_{t+1},\pi(s_{t+1})), Q_{\theta_{2}}(s_{t+1},\pi(s_{t+1})))]

  This term captures the expected cumulative rewards. It consists of the immediate reward $r(s_{t},a_{t})$ observed after taking action $a_{t}$ in state $s_{t}$, and the discounted future reward,
which is the expected value of the minimum Q-value between two Q-functions $Q_{\theta_1}$ and $Q_{\theta_2}$ for the next state $s_{t+1}$ and action chosen by policy $\pi$.

- Entropy Regularization

$$-\alpha \log \pi(a|s_{t+1})$$

This term encourages exploration. Here $\pi(a|s_{t+1})$ represents the probability of executing action $a$ in state $s_{t+1}$ according to policy $\pi$. The term $\alpha$ is the temperature parameter that balances the importance of the entropy term against the reward.

Combining all these terms, we get:

$$L_{Q}(\theta) = E_{(s_t, a_t)} \sim D[Q_{\theta}(s_t, a_t) - r(s_t, a_t)] + \gamma E_{s_{t+1}}[\min Q_{\theta_1}(s_{t+1}, \pi(s_{t+1})), Q_{\theta_2}(s_{t+1}, \pi(s_{t+1}))]$$

$$- a \log \pi(a|s_{t+1})^2$$

This equation aims to minimize the mean squared difference between the predicted and target Q-values using data sampled from replay buffer $D$.

References
11. Samanta, A.; Chang, Z. Adaptive service offloading for revenue maximization in mobile edge computing with delay-constraint. *IEEE Internet Things J.* 2019, 6, 3864–3872. [CrossRef]


37. Wang, Y.; Friderikos, V. A survey of deep learning for data caching in edge network. *Informatics* 2020, 7, 43. [CrossRef]


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.