electronics

MDPI

Article

Joint Edge Computing and Caching Based on D3QN for the Internet of Vehicles

Geng Chen [1,*], Jingli Sun [1], Qingtian Zeng [1], Gang Jing [2,*] and Yudong Zhang [3]

[1] College of Electronic and Information Engineering, Shandong University of Science and Technology, Qingdao 266590, China; sunludada@163.com (J.S.); qtzeng@sdust.edu.cn (Q.Z.)
[2] College of Electrical Engineering and Automation, Shandong University of Science and Technology, Qingdao 266590, China
[3] School of Computing and Mathematical Sciences, University of Leicester, Leicester LE1 7RH, UK; yudongzhang@ieee.org
* Correspondence: gengchen@sdust.edu.cn (G.C.); jinggang@sdust.edu.cn (G.J.)

Abstract: With the Internet of Vehicles (IOV), a lot of self-driving vehicles (SDVs) need to handle a variety of tasks but have very seriously limited computing and storage resources, meaning they cannot complete intensive tasks timely. In this paper, a joint edge computing and caching based on a Dueling Double Deep Q Network (D3QN) is proposed to solve the problem of the multi-task joint edge calculation and caching process. Firstly, the processes of offloading tasks and caching them to the base station are modeled as optimization problems to maximize system revenues, which are limited by system latency and energy consumption as well as cache space for computing task constraints. Moreover, we also take into account the negative impact of the number of unfinished tasks in relation to the optimization problem—the higher the number of unfinished tasks, the lower the system revenue. Secondly, we use the D3QN algorithm together with the cache models to solve the formulated NP-hard problem and select the optimal caching and offloading action by adopting an e-greedy strategy. Moreover, two cache models are proposed in this paper to cache tasks, namely the active cache, based on the popularity of the task, and passive cache, based on the D3QN algorithm. Additionally, tasks which deal with cache space are updated by computing the expulsion value based on type of popularity. Finally, simulation results show that the proposed algorithm has good performance in terms of the latency and energy consumption of the system and that it improves utilization of cache space and reduces the probability of unfinished tasks. Compared to the Deep Q Network with caching policy, with the Double Deep Q Network with caching policy and Dueling Deep Q Network with caching policy, the system revenue of the proposed algorithm is improved by 65%, 35% and 66%, respectively. The scenario of the IOV proposed in this article can be expanded to larger-scale IOV systems by increasing the number of SDVs and base stations, and the content caching and download functions of the Internet of Things can also be achieved through collaboration between multiple base stations. However, only the cache model is focused on in this article, and the design of the replacement model is not good enough, resulting in a low utilization of cache resources. In future work, we will analyze how to make joint decisions based on multi-agent collaboration for caching, offloading and replacement in IOV scenarios with multiple heterogeneous services to support different Vehicle-to-Everything services.

Keywords: edge computing; cache; offloading; revenue; Dueling Double Deep Q Network (D3QN)

Citation: Chen, G.; Sun, J.; Zeng, Q.; Jing, G.; Zhang, Y. Joint Edge Computing and Caching Based on D3QN for the Internet of Vehicles. *Electronics* **2023**, *12*, 2311. https://doi.org/10.3390/electronics12102311

Academic Editor: Mehdi Sookhak

Received: 11 April 2023
Revised: 16 May 2023
Accepted: 18 May 2023
Published: 20 May 2023

1. Introduction

At present, self-driving vehicles (SDVs) in the Internet of Vehicles (IOV) are developing rapidly, and they use a variety of sensors to work together to support many autonomous tasks, for example, route planning, automatic obstacle avoidance and so on, which have greatly eased traffic pressure. The number of tasks that SDVs needs to handle is enormous, but its own computing and storage resources are limited and cannot support the handling

of many frequent computing tasks timely; moreover, due to the high speed of the vehicle, some tasks have high requirements for latency, which SDVs cannot meet, which thus increases traffic risks. The emergence of mobile edge computing (MEC) provides the possibility of solving this problem. The migration of processing and storage resources to the edge of the network near SDVs can not only meet the requirements of low latency of tasks, but also reduce the pressure of the SDVs' computing tasks [1,2], leading to the timely completion of tasks. In traditional MEC, a MEC service handles the task itself, but computing resources of a single MEC service are limited, so surrounding MEC servers are needed to help to compute tasks. The traditional caching scheme is that MEC servers and SDVs cache all computing tasks as much as possible; however, their limited computing and caching capabilities as well as the possibility of the appearance of repetitive tasks are considered. In order to alleviate the computing and caching pressure of MEC servers and SDVs, while meanwhile reducing system latency and energy consumption, some effective caching schemes must be proposed to cache different task applications and data according to different cache locations, which in turn would avoid resource waste [3]. The MEC server can directly send the calculation results to the requesting vehicle if a task is cached; otherwise, the task is left in the vehicle for local calculation or is offloaded into the MEC for offloading calculation.

Recently, with the rapid development of mobile communication technology, MEC caching has received more and more attention. It reduces network burden by pushing computing or storage resources closer to user devices. Moreover, MEC avoids the long latency caused by transferring data from mobile devices to the Cloud and is thus able to support IOV applications that are crucial for latency. Based on the popularity and freshness of content, [4] proposed a new and original content caching strategy to reduce latency, demonstrating it could estimate locally. This strategy implements a coordinated cache in the edge domain and an autonomous cache in the core domain. The popularity based on Zipf's Law is used to cache data of the Internet of Things, and freshness is used to identify and replace (or avoid caching) content that is about to expire. In order to improve the utilization of storage and computing resources, [5] designed an iterative algorithm based on Gibbs sampling. Specifically, the authors designed a double iterative cache update algorithm, with the outer layer designing reasonable conditional probabilities based on Gibbs sampling, deriving the optimal edge caching strategy, and then iteratively updating the edge caching strategy. The inner layer optimizes the workload-scheduling policies. The problem of storage allocation and placement of content is modeled as a joint optimization problem in [6], where global and local popularity were used to cache content, and an allocation algorithm based on traffic was proposed to allocate storage space proportionally. In general, the popularity of content follows the Zipf distribution in most articles [7,8]. An Agent of Deep Reinforce Learning (DRL) can achieve specific goals in uncertain environments of the IOV, such as minimizing system latency and energy consumption [9]. There have been many examples from the literature proposing caching methods based on DRL. There are two kinds of tasks in [10], edge cache and resource allocation, which realize the optimal cache and allocation decision of all MEC servers based on the proposed DRL algorithm. A DRL-based caching strategy was proposed in [11] to improve the efficiency of the cache content when the content popularity is dynamic and unknown by using a deep Q neural network to approximate the Q action value function and optimizing parameters of the network. In order to improve the long-term profit of MEC and meet the low latency requirements of users, a joint optimization strategy with offloading calculation and resource allocation based on a DDQN was proposed to improve service quality in [12]. A distributed algorithm based on long short-term memory, the Dueling Deep Q Network (D2QN) and the Double Deep Q Network (DDQN) was proposed in [13] to determine the offloading decision without knowing the task model for minimizing the long-term average cost of the system. The content popularity prediction algorithm based on an offline spatial–temporal process was introduced to predict the time-varying content popularity and consider the relationship between content and time and space

in [14]. In addition, the strategy involved in the offloading calculation and the caching of tasks, and even resource allocation, can be optimized by utilizing the DDPG algorithm. An air-assisted vehicular caching scheme based on a DQN was proposed in [15], which uses unmanned aerial vehicles to cache content related to vehicle driving safety and with high request probability. Additionally, the DQN was used to cache and calculate by learning the historical content of vehicle users. A joint push caching strategy based on hierarchical reinforcement learning was proposed in [16], which took into account the long-term file popularity and short-term temporal correlation. The author also proposed five cache reference schemes. The first scheme is Least recently used (LRU), which replaces the files that have not been used for the longest time with new files. The second is Least often used (LFU), which replaces the files that have the least number of requests with new files. The third is The Most popular (MP) scheme, which calculates the stable request probability for each file. The fourth is the Local Most Popular (LMP) scheme, which exploits the long-term popularity and short-term correlation of files to cache the files with the highest request probability. The fifth is the Threshold Local Most Popular (TLMP) scheme, which uses the Zipf distribution to cache the most popular files. Most of the scenarios in the above articles discussed how users receive content from the server through the downlink. To our knowledge, few articles discuss computing-requested tasks and caching tasks through the uplink. There are few caching schemes that combine caching models with a Dueling Double Deep Q Network (D3QN) for multi-objective optimization of offloading and caching in order to reduce system costs and the latency and energy consumption of IOVs, as well as improve the utilization of cache resources in MEC systems. Therefore, we propose a joint edge computing and caching method based on the D3QN in this paper, which uses an active caching model based on the popularity of the task, passive caching model and D3QN algorithm to make joint decisions for task caching and offloading. Active caching is the process of caching tasks that are the most frequently requested by users by calculating their popularity. Passive caching uses the D3QN to select caching tasks.

The main contributions of this paper are as follows:

1. The distributed MEC is proposed to reduce the average latency and energy consumption of the system through the collaboration of MEC servers and takes full advantage of the limited computing and caching resources of the system.
2. An active cache and passive cache are added to the MEC system. They use popular schemes and the D3QN to cache tasks, respectively, which improves the utilization rate of the cache space.
3. A joint edge computing and caching method based on the D3QN is proposed, greatly improving system latency, energy consumption and revenue.

The structure of the article is as follows. In Section 2, the system model is established, and the optimization problem is formulated. Section 3 introduces the basics of the D3QN as well as a joint edge computing and caching method based on the D3QN. The simulation parameters and results are discussed in Section 4. Section 5 summarizes the work of this paper.

2. System Model and Problem Formulation

Figure 1 illustrates the caching and offloading MEC system in the IOV, in which SDVs and roadside units (RSUs) have computing capability, with only RSUs having caching capability; however, their resources for computing and storage tasks are very limited. The system includes N SDVs v driving on a straight road, with the SDVs set as $\mathbf{V} = \{v_1, v_2, \ldots, v_N\}$, $v \in \mathbf{V}$; M RSUs are found along the road, and each RSU m is equipped with a MEC server and has the ability to calculate and store tasks, with the RSUs set as $\mathbf{M} = \{m_1, m_2, \ldots, m_M\}, m \in \mathbf{M}$. The MEC server that can communicate directly with SDVs is called an access MEC (aMEC) server, and the MEC server that can communicate indirectly with SDVs is called a collaborating MEC (aMEC) server [17]. This paper discusses the caching and offloading of task k in time slot t; the time set is $\mathbf{T} = \{t_1, t_2, \ldots, t_T\}, t \in \mathbf{T}$.

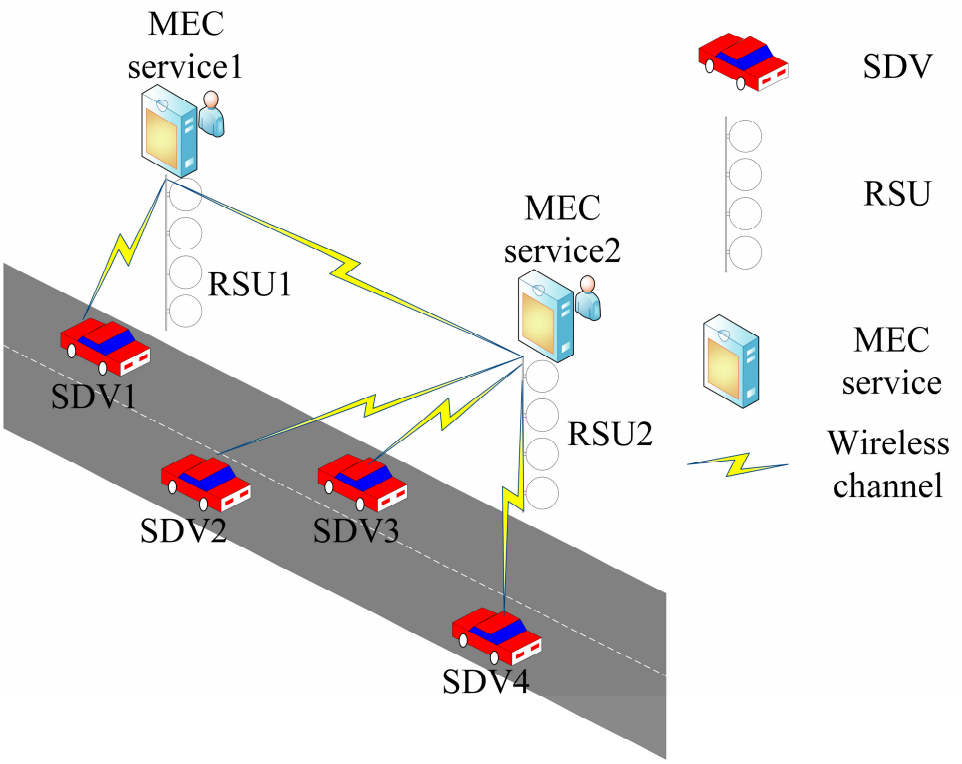

Figure 1. Multi-task computation offloading and caching of MEC system in the IOV.

Suppose N SDVs generate many assisted driving tasks at slot t, with some tasks with the smaller required computing resources and size of task being kept locally for calculation, and some tasks with higher required computing resources being offloaded to and calculated based on the RSUs. In addition, if the request task has already been cached in the RSU, that is, a cache hit task occurs, the SDV can directly obtain the calculation results of the task, with the following caching model then needing to be used. Otherwise, SDVs must offload tasks that are not cached in the RSU to the target RSU for computation using a wireless channel or a CPU to compute locally; this process requires the following computation and caching models. Intuitively, collaborative task offloading through the caching mechanism not only reduces the latency and energy consumption of the system, but also makes full use of the cache space of the RSU.

2.1. Task Model

Current SDVs can do a lot of things; for example, they can accurately map their surroundings and monitor the location of nearby vehicles, traffic lights, pedestrians and lane markings, all in real-time. In general, they need to complete a variety of multi-task calculations in a very short time.

Suppose that N SDVs generate W tasks k that request the computation in slot $t \in \mathbf{T}$, and the task set is set as $\mathbf{K} = \{k_1, k_2, \ldots, k_W\}, k \in \mathbf{K}$, and we assume that there are repetitive tasks in these W tasks. Each task $k \in \mathbf{K}$ has four different task characteristic parameters: $T_{k,t}^{\mathrm{m}}$ (in seconds) represents the maximum tolerance delay of the computing task and is used to constrain the computation latency of the task, and also represents the maximum time that can be accepted for the completion of the task k in slot t; $\psi_{k,t}$ represents the computing resources required for computing task k in slot t. In order to calculate the task, the computing resources allocated by the SDV or MEC server must be greater than or equal to it; $I_{k,t}$ represents size of the task k in slot t. Assuming that the size of the task k is known to the MEC server, the MEC server can allocate cache space based on the size of the task; k_{task} represents the type of the task k. Assuming that the type of the task is known to the

MEC server, the MEC server can assign the task to the corresponding type area of the cache area, and there are L task types.

2.2. Caching Model

Caching tasks can greatly reduce the overall system latency and energy consumption, meanwhile improving quality of service because cached tasks do not need to be repeatedly calculated and transmitted. However, based on the popularity and size of each task k, what kind of task to cache and how to cache it are challenging problems.

Here, two caching models are presented: active caching and passive caching.

Passive cache: The general process is that the D3QN generates caching decisions based on the state of the system to cache tasks in the appropriate place. By using the advantage function, the D2QN can estimate the Q value more accurately and select more firm actions after collecting the data of only one discrete action. The DDQN selects the target Q value through the action of target Q value selection, so as to eliminate the problem of overestimation of the Q value. Additionally, the D3QN combines the advantages of the D2QN and DDQN, which can make better decisions and execute more appropriate actions.

Active cache: It determines whether to cache a task by calculating the popularity of the task. In general, the most popular tasks are the ones that have the most requests. Additionally, we not only calculate the popularity of a single task k, but also the class popularity of task types. We determine whether the tasks belong to the same type, then treat tasks of the same type as a whole and calculate the entire popularity. The purpose of computing the popularity of the task type is to reduce the frequency of deletion tasks. When the cache area is full, if only one task is deleted, and then later tasks are cached in the cache area, the cache area is likely to fill up again, and the task will continue to be deleted. In this way, continuously caching and deleting tasks in a slot will increase the load of the RSU, because the RSU needs to allocate part of the computing resources to calculate and delete tasks. The tasks of the same type will be deleted based on the popularity of the task type, so that the remaining cache area space will be increased, making it easier for subsequent tasks to be cached, which does not cause the task to be deleted many times in a slot. So, from this point of view, calculating the popularity of the task type is better than calculating individual task popularity.

In general, the popularity of task k follows the Zipf distribution that is a statistical rule of thumb and describes the theorem that only a few tasks are used frequently and most tasks are used infrequently, which is expressed as [17]

$$P_k = \mathrm{Zipf}_k(\lambda, W) = \frac{1}{k^{\lambda} \sum_{k \in \mathbf{K}} k^{-\lambda}} \tag{1}$$

where $\lambda \in (0, 1)$ is the Zipf slope and controls the degree of deviation from popularity; W is the number of tasks generated in slot t.

Some cached tasks in RSUs need to be updated in real-time due to the limited cache resources of RSUs, so some deletion rules for tasks need to be set. The expulsion value of a task type is used for updating tasks of the cache space and it deletes all the tasks falling into the same type based on the eviction value. Here we present the formula for the class expulsion value of the task:

$$P_{\mathrm{exp}} = \mathrm{argmin}\left(\frac{\sum_{k \in \mathbf{K}_{\mathrm{A}}} I_k}{q \times P_{\mathrm{mean}}} \right) \tag{2}$$

where q is the number of tasks belonging to the same type and $\mathbf{K}_{\mathrm{A}} = \{k_1, k_2, \ldots, k_q\}$ is the set of tasks in the cache area. This formula implies that the smaller the type of popularity, the larger the size of the task, and the easier the task is to delete. P_{mean} is the type of

popularity that indicates the popularity of the task class and is the average popularity of all tasks of the same type.

$$P_{\text{mean}} = \frac{\sum_{k \in \mathbf{K}_A} P_k}{q} \tag{3}$$

where Equation (3) must satisfy $L \times q \leq K$, which represents that the number of tasks cached into the cache area cannot exceed the number of tasks requested for computation generated by SDVs. If the cache resources of the RSU have been exhausted, all corresponding tasks belonging to the same type will be removed based on the maximum expulsion value calculated.

There are two related decisions: the caching decision $h_{k,t}$ and the deletion decision $H_{k,t}$. $h_{k,t} = \{0, 1\}$ represents the caching decision for task k in slot t.

$$h_{k,t} = \begin{cases} 1, \text{if required task } k \text{ is cached} \\ 0, \text{ otherwise} \end{cases} \tag{4}$$

if $h_{k,t} = 1$, task k will be cached in the RSU; otherwise, it will not be cached. The caching decision must meet the following condition due to the limited cache resources of RSUs.

$$\sum_{k \in \mathbf{K}} \zeta(h_{k,t} = 1) I_{k,t} \leq C \tag{5}$$

where $\zeta(\bullet)$ represents the indicator function; $\zeta(\bullet) = 1$ if the event $\bullet$ is true, otherwise $\zeta(\bullet) = 0$; C indicates size of the cache space in the RSU.

$H_{k,t} = \{0, 1\}$ represents the deletion decision of the task type in slot t.

$$H_{k,t} = \begin{cases} 1, \text{ if task in task type is deleted} \\ 0, \text{ otherwise} \end{cases} \tag{6}$$

if the cache area is full and the task type to which task k belongs has the largest expulsion value, $H_{k,t} = 1$, all tasks belonging to the same class that correspond to the maximum expulsion value will be deleted to facilitate new tasks being cached in the cache area.

The caching process for tasks is shown in Figure 2. For the task that requests computation, its popularity needs to be calculated firstly. If task k is the most popular task, that is it has the highest popularity, then task k will be cached in the cache area of the RSU. If task k is not the most popular task, the D3QN algorithm is run. If the agent performs the action of the caching task, task k will be cached in the cache area of the RSU. Otherwise, the task will not be cached. Secondly, the type of the task k needs to be judged when the task is cached. For example, if the type of task k is task type1, it will be cached in the cache area of task type1. Finally, each time a task is cached, it is determined whether the cache space of the RSU is full. If the cache space is full, the expulsion value of the task type is calculated, and the related tasks belonging to the same class with the largest expulsion value are removed from the cache area.

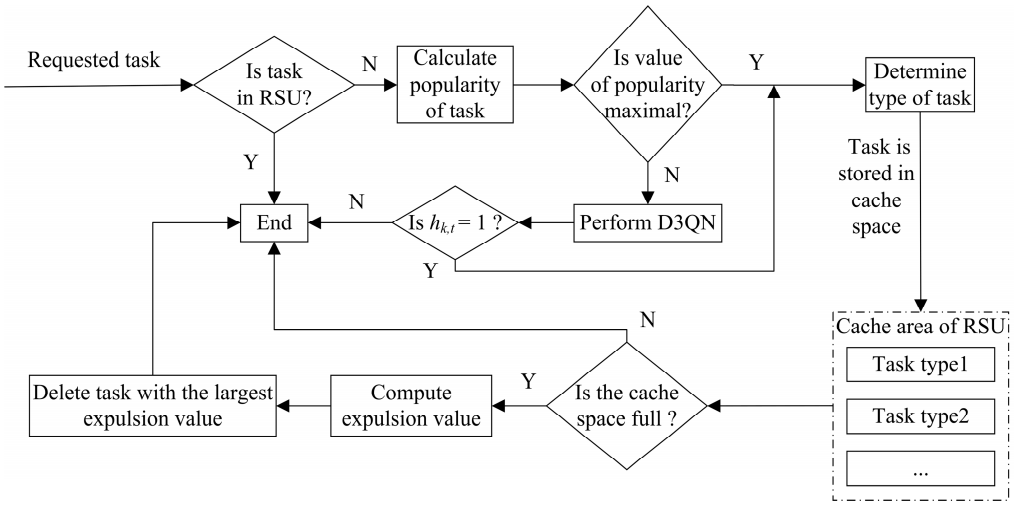

Figure 2. Caching and deletion process of tasks.

2.3. Communication Model

Generally, it is assumed that the network topology of IOVs is constant in slot t. For the available RSU, the SDV transmits task data through the uplink and obtains its calculation results. Let the transmission rate between SDV v and RSU m in slot t be v_t, denoted as [18]

$$v_t = b_{vm,t} \log_2 \left(1 + \frac{\rho G_{vm,t}}{\xi^2}\right) \tag{7}$$

where $b_{vm,t}$ represents the orthogonal distribution bandwidth between RSU m and SDV v in slot t, and we assume that $b_{vm,t}$ is a constant over a time slot t; ρ represents the transmitted power of the SDV; ξ^2 shows the Gaussian white noise power; $G_{vm,t}$ represents the channel gain between SDV v and RSU m in slot t.

2.4. Computation Model

To determine whether task k has occurred a cache hit when slot t arrives, that is whether task k can be found in the RSU. If a cache hit occurs, the results of task k will be transmitted to the SDV. Otherwise, the SDVs will calculate the tasks locally in the vehicle or offload them to the MEC server for offloading calculation. So, there are three kinds of calculations: direct transmission result, the computation local and the computation offloading.

2.4.1. Direct Transmission Result

If the requested task k can match the task in the cache area, then the task does not need to be computed and transmitted, and only the result needs to be transmitted. Otherwise, the task k will be offloaded and computed.

$z_{k,t} = \{0, 1\}$ represents the matching decision of task k in slot t.

$$z_{k,t} = \begin{cases} 1, \text{if required task } k \text{ is matched} \\ 0, \text{ otherwise} \end{cases} \tag{8}$$

if the requested task k can be found in the cache area of the RSU, with $z_{k,t} = 1$, the SDV can obtain the processing results of the task directly. Otherwise, $z_{k,t} = 0$. The energy consumed by the computing task is actually the amount of computing resources consumed by the SDV or RSU computing task. Additionally, energy consumed generated through transmission is the amount of resources consumed by transmitting the task data. Generally, computing resources required by the task will be smaller than that of the server, otherwise the server will be unable to process the task. There will be calculation latency and calculation energy consumption when task k is calculated. If task k needs to be offloaded, there will also

be transmission latency and transmission energy consumption during the transmission process. However, unlike computing tasks, transmitting results does not need to transfer task data and programs, but only the transmission results; therefore, its latency and energy consumption are very small, and can also be ignored and both set to zero.

If $z_{k,t} = 0$, the task k needs to be computed. This requires an offloading decision; $x_{k,t} = \{0, 1, 2\}$ determines where to offload task k for computation in slot t, which represents the offloading decision.

$$x_{k,t} = \begin{cases} 2, \text{ if task } k \text{ is offloaded to the cMEC server} \\ 1, \text{ if task } k \text{ is offloaded to the aMEC server} \\ 0, \text{ if task } k \text{ is computed locally} \end{cases} \tag{9}$$

2.4.2. Computation Local

When the computational resource required $\psi_{k,t}$ of the task k is not very large in slot t, the computational resource of the SDV is greater than it, and the SDV can complete the calculation of the task by itself; this is known as local computation.

If $x_{k,t} = 0$, the SDV allocates its own computing resources to compute tasks, also known as local computation, and its latency can be expressed as

$$T_{k,t}^{\mathrm{L}} = \frac{\psi_{k,t}}{f_{v,t}} \tag{10}$$

and the corresponding calculation energy consumption is

$$E_{k,t}^{\mathrm{L}} = \sigma \psi_{k,t} \tag{11}$$

where $\psi_{k,t}$ represents the computing resources required for computing task k in slot t; $f_{v,t}$ represents the computing resources allocated by the SDV v to compute task k in slot t, assuming that all SDVs have the same computing resources; σ represents the energy consumption per unit computing resource.

2.4.3. Computation Offloading

If $x_{k,t} \neq 0$, tasks will be offloaded to the RSU for calculation, which is involves three steps. Firstly, the SDVs offload the data and programs of the tasks to the aMEC server through the uplink, which generates the transmission latency and transmission energy consumption. If there are a large number of tasks offloaded to the RSU, aMEC will allocate some tasks to cMEC for auxiliary calculations. Secondly, the MEC server allocates computing resources to tasks for computing. This process will generate computational latency and energy consumption. Thirdly, the MEC server returns the calculation results to the SDV; because transmission latency and energy consumption are lower, they are ignored.

If $x_{k,t} = 1$, task k is offloaded to the aMEC server that communicates directly to compute; if $x_{k,t} = 2$, the number of tasks may be too large, and thus the neighboring MEC server is required to compute the tasks, task k is offloaded to the cMEC server to compute, and their computation method remains the same, meaning that both go through the communication and computation processes.

The transmission latency for transferring task data and programs from SDV v to the MEC server in slot t can be expressed as

$$T_{k,t}^{\mathrm{tra}} = \frac{I_{k,t}}{v_t} \tag{12}$$

The corresponding energy consumption generated by transmitting the task data to the MEC server is

$$E_{k,t}^{\mathrm{tra}} = P(t) T_{k,t}^{\mathrm{tra}} \tag{13}$$

The corresponding calculation latency generated by the MEC server to compute task k is denoted by

$$T_{k,t}^{\mathrm{com}} = \frac{\psi_{k,t}}{f_t} \tag{14}$$

The energy consumption generated due to task k being offloaded by the the MEC server in slot t is given by

$$E_{k,t}^{\mathrm{com}} = \sigma \psi_{k,t} \tag{15}$$

where $I_{k,t}$ represents size of task k in slot t; v_t indicates the transmission rate between SDV v and RSU m; $P(t)$ means the average transmission rate between SDV v and the MEC server; f_t represents the computing resources assigned to tasks by the MEC server in slot t; $\psi_{k,t}$ denotes the computing resources required for computing task k.

It is worth noting that the transmission rate between the SDV and the directly connected RSU is different from that of the SDV and the neighboring RSU; the former is smaller than the latter because the distance is smaller. So, the transmission latency and energy consumption generated by the transmission of the task data and programs from the SDV to the aMEC server are both less than that of the task data transferred to the cMEC server.

2.5. Problem Formulation

If task k has already been calculated, the latency and energy consumption of the task are the duration and consumed energy from the generation of task k in achieving the result is obtained using the SDV. The $x_{k,t}$ and $z_{k,t}$ represent the offloading decision and caching decision of task k, respectively; therefore, the energy consumption and latency required to complete task k in slot t, respectively, can be represented in the following forms:

$$E_{k,t} = \zeta(z_{k,t} = 1)0 + \zeta(z_{k,t} = 0)\left[\zeta(x_{k,t} = 0)E_{k,t}^{\mathrm{L}} + \zeta(x_{k,t} \neq 0)\left(E_{k,t}^{\mathrm{tra}} + E_{k,t}^{\mathrm{com}}\right)\right] \tag{16}$$

$$T_{k,t} = \zeta(z_{k,t} = 1)0 + \zeta(z_{k,t} = 0)\left[\zeta(x_{k,t} = 0)T_{k,t}^{\mathrm{L}} + \zeta(x_{k,t} \neq 0)\left(T_{k,t}^{\mathrm{tra}} + T_{k,t}^{\mathrm{com}}\right)\right] \tag{17}$$

where $\zeta(z_{k,t} = 1)0$ represents task k matching the task in the cache area, so the latency and energy consumption of the task during the cache hit are zero. $E_{k,t}^{\mathrm{L}}$ and $T_{k,t}^{\mathrm{L}}$ represent the energy consumption and the latency of the task in slot t through the local computation, respectively. $E_{k,t}^{\mathrm{tra}}$ and $T_{k,t}^{\mathrm{tra}}$ represent the energy consumption and latency of the transmission task k in slot t, respectively. $E_{k,t}^{\mathrm{com}}$ and $T_{k,t}^{\mathrm{com}}$ represent the calculation energy consumption and latency of task k in slot t, respectively.

System latency and energy consumption are two critical parts of system cost, and are the core problems of edge caching. Since there exists a magnitude gap between latency and energy consumption, we need to set two coefficients, the coefficients of latency and energy consumption, to eliminate the magnitude gap between them. We chose index form as the form of a cost function because of Equation (20) in this paper. In general, the lower the cost, the higher the system revenue, so we need an inverse function to represent the relationship between the cost and revenue of the system, and the common inverse function is generally the trigonometric function and exponential function; here, however, we take an exponential function to calculate the system revenue. Therefore, the cost of computing task k in slot t can be represented by an exponential form of the weighted sum of delay and energy consumption; the system cost is denoted by

$$J_t = \sum\nolimits_{k \in \mathbf{K}} e^{-(\omega_E E_{k,t} + \omega_T T_{k,t})} \tag{18}$$

where ω_{E} and ω_{T} are the coefficients of energy consumption and latency; $E_{k,t}$ and $T_{k,t}$ are the total energy consumption and latency required to calculate task k in slot t, respectively, $k \in \mathbf{K}$. The cache capacity occupied by all tasks in the cache area of RSU in slot t is

$$c_t = \sum\nolimits_{k \in \mathbf{K}} \zeta(h_{k,t} = 1) I_{k,t} \tag{19}$$

where $I_{k,t}$ denotes the size of task in cache area in the slot t.

From the perspective of cache, the cache capacity occupied by all tasks in the cache area is also a kind of caching cost, so it is defined as system revenue by making a weighted sum of the reciprocal of the above system cost and the cache capacity of all cached tasks. The average system revenue is given by

$$P_t = \mathbb{E}\left[\lim_{|\mathbf{T}\to\infty|} \frac{1}{|\mathbf{T}|} \sum\nolimits_{t\in\mathbf{T}} \left(J_t^{-1} + \omega_c c_t\right)\right] \tag{20}$$

s.t.

$$h_{k,t} = \{0, 1\} \tag{21}$$

$$x_{k,t} = \{0, 1, 2\} \tag{22}$$

$$z_{k,t} = \{0, 1\} \tag{23}$$

$$\sum\nolimits_{k\in\mathbf{K}} \zeta(h_{k,t} = 1) I_{k,t} \le C \tag{24}$$

$$T_{k,t} \le T_{k,t}^{\mathrm{m}} \tag{25}$$

$$\sum\nolimits_{k\in\mathbf{K}} \zeta(x_{k,t} = 0) \psi_{k,t} \le \psi_{v,t} \tag{26}$$

$$\sum\nolimits_{k\in\mathbf{K}} \zeta(x_{k,t} \ne 0) \psi_{k,t} \le \psi_{m,t} \tag{27}$$

$$\sum\nolimits_{k\in\mathbf{K}} [\zeta(x_{k,t} = 0) + \zeta(x_{k,t} \ne 0)] \le K \tag{28}$$

where ω_{c} represents the coefficient of the cache cost, which is used to eliminate the magnitude gap between the system cost and caching cost; $\psi_{m,t}$ indicates the total computation resource owned by RSU m; $\psi_{v,t}$ represents the total computation resource of SDV v in slot t.

The problem of Equation (20) is composed of the calculation cost J_t and cache cost c_t of the system. Within the constraint range, the smaller J_t and the larger c_t, the higher the completion rate of the computing tasks and the optimal offloading and cache decisions of the tasks. Therefore, the key to minimizing the overall cost of the system is to maximize the problem of Equation (20). The purpose of this paper is to maximize the average revenue of the system; in other words, maximize Equation (20), which is however subject to constraints. Here, Equation (21) represents the offloading decision, which decides where the task will be offloaded to; Equation (22) represents the caching decision, which determines whether the task will be cached or not; Equation (23) represents the matching decision, which determines whether the task is matched. It is noted that $h_{k,t}$ and $z_{k,t}$ are binary variables. Equation (24) means that all tasks cached to the RSU occupy a cache size smaller than the size of the cache resources of the RSU; Equation (25) shows that the latency of computation task k is less than its tolerance latency; Equation (26) indicates that the total computing resources required by the tasks to be offloaded are less than the total computing resources of the RSU; Equation (27) means that the total computing resource required by the tasks requiring local computing is less than the total computing resources of the SDV; Equation (28) indicates that the number of tasks to be calculated is not greater than the total number of tasks generated by users.

3. Solution Based on the D3QN

From a caching perspective, there are only two possibilities for a task: cache or do not cache. From the perspective of offloading, there are generally two types: offloading or local. If a task can be matched, then its energy consumption and latency are zero; if not, then

there is energy consumption and latency. If a task needs to be computed, there are only two general scenarios: local computation and offloading computation. In other words, it does not matter if it's from the perspective of caching or offloading—the energy consumption and latency of the task can only be 0 or 1. This 01 program happens to be a special form of integer programming. The latency and energy consumption of each task depend on the characteristics of the task itself, and the tasks generated by each time slot are different, so the latency and energy consumption do not change linearly. For the variable task, it can only be an integer, so the cost function composed of energy consumption and latency is a mixed-integer nonlinear function. As for the cache cost, each cached task is different, so the cache cost is also different. It only changes with the task and the policy. However, the generation of the task is random, and the generation of the policy is based on learning, so the cache cost function is a nonlinear function. Only when the ejection value meets the condition does the cache cost change significantly, such as in the form of a large reduction, otherwise it simply keeps increasing. Additionally, this increase can only be 1 or 0, so it increases when $h_{k,t} = 1$ and it does not increase when $x_{k,t} = 0$. The cache cost is the sum of the size of each task, and the number of tasks can only be an integer, so the cache cost function is also a mixed-integer nonlinear function. So, the optimization problem of these two combinations is a mixed-integer nonlinear programming problem. The mixed-integer nonlinear programming problem of Equation (20) has an NP-hard property [19] and is difficult to solve directly. The problem is addressed by designing an online solution that allows the agent to interact with the system environment in real-time to make optimal decisions about caching and offloading. Therefore, we propose a joint edge computing and caching method based on the D3QN to find a joint optimal strategy for x_k and h_k rather than using traditional methods for solving a nondeterministic polynomial problem.

The D3QN introduces the idea of a DDQN based on a D2QN. The action corresponding to the optimal action value in the next state is obtained by using the evaluation network, then the target network is used to calculate the action value to obtain the target value. The problem of overestimation is effectively avoided through the interaction of the two networks.

3.1. D3QN Algorithm

Reinforcement learning (RL) refers to the guiding behavior where an agent learns a policy π to obtain a reward during its interaction with the environment, with the purpose of making the agent obtain the maximum reward. Actually, the policy π is a process; it refers to which action is selected to execute in state s_t at time step t [20]. After the action is executed, the agent receives the next state and the immediate reward. RL obtains learning information and updates model parameters by receiving feedback from the environment and directing it to the action. State s' at time $t + 1$ is determined only by the state s_t and the action a_t at time t. At each time step t, the purpose of the agent is to obtain the maximum return on future discounts, as shown below [19]:

$$R_t = \sum_{i=0}^{n} \mu^i r_{t+i} \tag{29}$$

where μ denotes the discount factor and r_{t+i} indicates the reward in step $t + i$.

Generally, Q-learning is widely applied to small state–action space problems. It will train an agent to find the action with the best action value function $Q^*(s, a)$. The Behrman optimality equation is used to express the best action value function in the following form [21]:

$$Q^*(s, a) = \mathbb{E}\left[r + \mu \max_{a'} Q^*\left(s', a'\right) \mid s_t = s, a_t = a\right] \tag{30}$$

where a' is an action of the next time step $t + 1$, and its best action value is $Q^*(s', a')$; $\mathbb{E}[*]$ represents the expectation function.

Q-learning is suitable for the low-dimensional state–action space value, but not suitable for the high state–action space value, therefore, DQN is proposed.

In the DQN algorithm, at each slot t, the environment represented as a four-tuple $e_i = (s_t, a_t, r_t, s_{t+1})$ is stored into the experience replay pool $D = (e_1, e_2, \ldots, e_n)$. After a period of training, the DQN will randomly sample tuples of batch size B from the replay memory pool D. These batch tuples are input into the neural network (NN) to train the agent to approximate the action value function; that is, an action value function is derived from $Q(s, a; \theta_i)$ to approximate $Q^*(s, a)$, where θ_i is the training weight of the Q network in each iteration i. The DQN has two NNs: the target Q network F_t and the estimated Q network F_e; the former is used to calculate the target Q value Q_t, and the latter is used to estimate the Q value Q_e in the current state. They have the same network structure. The purpose of the DQN is to train appropriate weights to minimize the loss $L_i(\theta_i)$ [19] of Q_t and Q_e, as shown in Equation (31).

$$L_i(\theta_i) = \mathbb{E}\left[r + \mu \max_{a'} Q_t(s', a'; \theta_i^t) - Q_e(s, a; \theta_i))^2\right] \tag{31}$$

where r is the reward in slot t and θ_i^t is the training weight of the target Q network in each iteration i.

For every J episode, the parameters of F_e are used to update the training parameters of F_t. In the update process, only the weight of the estimated network is updated, but not the weight of the target network. After updating a certain number of times, it is set as 3000 times in this article, a learning rate γ is introduced and the weighted average value of the old target network parameters and the new estimated network parameters are directly assigned to the target network, and then the next batch of updates are carried out, so that the target network can also be updated. Such updates are called soft updates, represented by

$$\theta_t \leftarrow \gamma\theta_e + (1 - \gamma)\theta_t \tag{32}$$

where θ_t represents the weight of the target network and θ_e represents the weight of the estimated network. Then, the DQN optimizes the loss function using stochastic gradient descent. The agent has two choices of action: selecting random actions with probability ε, or selecting the action with the highest action value.

However, the disadvantage of the DQN and Q-learning is that they select and evaluate Q-valued functions in each iteration by using the same action values, which sometimes leads to an overestimation problem. To solve this problem, the DDQN is proposed. Its advantage is that it selects the action with the highest Q value from F_e instead of selecting the same action value and then calculating Q_t in F_t, which reduces overestimation to some extent and makes the Q value closer to the true value, as in the following [22]:

$$L_i(\theta_i) = \mathbb{E}\left[\left(\mu \max_{a'} Q_t(s', \operatorname{argmax}_{a'} Q_e(s', a; \theta_i); \theta_i^t) + r - Q_e(s, a; \theta_i)\right)^2\right] \tag{33}$$

The D2QN uses alternative and complementary methods to reconstruct the network; that is, the input of the D2QN algorithm is also the state information of the environment, but its output consists of two branches, which are the state value V (scalar) of the state and the advantage value A (vector with the same dimension as the action space) of the state. The advantage function represents the dominance of the average value of an action a over the average value in state s and is used to normalize the Q value, as follows:

$$Q^*(s, a) = V^*(s) + A^*(s, a) \tag{34}$$

where $V^*(s)$ represents the best state value function that estimates the importance of the state to the Q value and $A^*(s, a)$ represents the best advantage function that estimates the importance of the best action to Q value compared to other actions. They are computed separately.

The features are extracted through a convolutional neural network (CNN) layer, value function $V(s; \theta, \beta)$ and the advantage function; the $|A|$-dimensional vectors $A(s, a; \theta, \alpha)$ are output through two Fully Connected Neural network (FCNN) streams, θ represents the

parameters of the CNN layer and β and α are the parameters of two flows of the FCNN layer. The D2QN uses Equation (35) to calculate the Q value [23].

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left[A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right] \tag{35}$$

Because the D2QN has a better network structure, it performs better.

The D3QN integrates the DDQN and D2QN and solves the problems of overestimation, instability and convergence difficulty well. The only difference between the D3QN and D2QN algorithm is the method that they use to calculate the target value. In the D3QN algorithm [24], the method of calculating the target value is

$$y_t = r' + \mu Q\left[s', \text{argmax}_a Q(s', a; \theta_e); \theta_t\right] \tag{36}$$

where r' represents the reward of the next time step $t+1$. We use the D3QN as the basis to solve the proposed NP-hard problem in this paper.

3.2. Joint Computing and Caching Based on the D3QN

A common model of RL is the standard Markov decision process (MDP), in which the state information contains previous historical information. Here, we model the proposed problem as an MDP and solve this MDP problem using a joint computation and caching algorithm based on the D3QN. The MDP is defined as four-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R},)$, where $\mathcal{P}$ is the state transition probability function, $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space and $\mathcal{R}$ is the reward function.

3.2.1. States

The system state s_t represents the environment information perceived by the agent and the changes brought by its own action, which can be expressed as a set of information parameters. System state space $\mathcal{S}$ is the basis for the decisions the agent makes and evaluates their long-term benefits, as well as includes the states of the SDVs and tasks and the available resources of the RSUs. The agent needs to offload tasks according to the number of requested tasks; when the number is large, tasks can be offloaded to the neighboring cMEC for calculation. The popularity of the task also must be cached with the task as well as the size of the task in order to calculate the cache space occupied by the task, and deletion of the task is thus performed. Therefore, the system state s_t in slot t can be expressed in the following form:

$$s_t = [k_{1,t}, k_{2,t}, \ldots, k_{W,t}, P_{1,t}, P_{2,t}, \ldots, P_{W,t}, z_{1,t}, z_{2,t}, \ldots, z_{W,t}, I_{1,t}, I_{2,t}, \ldots, I_{W,t}, c_{1,t}, c_{2,t}, \ldots, c_{W,t}]^\top \tag{37}$$

where $k_{i,t}$ represents i-th task k in slot t, $P_{i,t}$ represents the popularity of task k computed by using Equation (1); $I_{i,t}$ represents the data size of i-th task k; $z_{i,t}$ denotes the matching decision of task k_i; $c_{i,t}$ denotes the cache capacity for all tasks that are cached to the RSU; $k_i \in \mathbf{K}$ is i-th task, $i \in \{1, 2, \ldots, W\}$.

3.2.2. Action

Based on the observed state s_t of the environment, the agent generates a better caching policy to cache tasks and a better offloading policy to compute tasks. This operation consists of two issues that need to be resolved: where tasks should be offloaded to compute and whether tasks should be cached. So, the action a_t in any slot t is

$$a_t = [x_{1,t}, x_{2,t}, \ldots, x_{W,t}, h_{1,t}, h_{2,t}, \ldots, h_{W,t},]^\top \tag{38}$$

where $x_{i,t}$ denotes the offloading decision of i-th task k_i; $h_{i,t}$ represents the caching decision of i-th task k_i, $k_i \in \mathbf{K}, i \in \{1, 2, \ldots, W\}$.

3.2.3. Reward

After the environment is set up, the next step is to consider what actions the agent will take and how we will reward it once it is completed. This requires designing a reward function that allows the algorithm to understand when the strategy becomes better, ultimately leading to the desired outcome. Reward r_t is a scalar obtained when an agent takes action a_t based on the s_t of the environment. Maximizing the average system revenue is our goal, which means that actions that result in lower costs should have higher immediate rewards. Meanwhile, the reward function should be associated with Equation (20). Here, there are three types of rewards. If latency generated by computing task k is less than the tolerated latency of task k, the agent will be rewarded with system revenue; if latency generated by computing task k is greater than the tolerated latency of task k, the agent will be punished, and the reward is a negative system revenue; if the task has a cache hit, the reward is the maximum reward value defined.

$$r_t = \begin{cases} P_t, \text{ if } T_{k,t} \leq T_{k,t}^{\mathrm{m}} \\ -P_t, \text{ if } T_{k,t} > T_{k,t}^{\mathrm{m}} \\ \mathrm{m\_} \text{ reward, if } z_{k,t} = 1 \end{cases} \tag{39}$$

where m_reward is the maximum reward value defined and is equal to 2.

3.2.4. Transition Probability

$P(s, s')$ represents the transition probability that action $\mathcal{A}$ of state s of slot t leads to state s' of slot $t + 1$ [19].

$$P(s, s') = P_r\{s(t+1) = s' \mid s(t) = s, \mathcal{A}(t) = \mathcal{A}\} \tag{40}$$

Our proposed algorithm is a combination of cache models and the D3QN algorithm. For each episode, the agent constantly interacts with the MEC-enabled environment in order to find the action that maximizes the reward value. There are two parts to this process; if the requested tasks can be matched, then the agent can obtain the maximum reward directly, so the caching models and D3QN algorithm are not needed; otherwise, the functions of caching models and the D3QN algorithm are executed.

3.2.5. Cache Model

Two caching models are proposed in this paper, and they implement different caching strategies based on different parameters of tasks. The first is active cache, which takes advantage of the popularity of task k in caching the task. There are two parts to this process: firstly, we determine whether the states' parameters $z_{i,t}$ are 1. If they are not, this means that the task has no cache hit. Next, tasks where the popularity P_k is the maximum popularity are cached in the cache space. Otherwise, the calculation result is directly obtained. Additionally, the tasks are cached in the corresponding type area depending on the type of task. The second part of the process is passive cache, in which the agent of the D3QN interacts with the environment to obtain the state information from the SDVs, RSUs and tasks, so as to cache tasks that are not the most popular. Additionally, the agent uses the greedy strategy to find the optimal policy and chooses the optimal action to cache the appropriate task according to the system states that cache the space of the cache area, the popularity of the task and the size of task, and so on. The cache resource states of the RSU will be checked in each slot. If the cache resource has been exhausted, the corresponding tasks that belong to the same type will be deleted according to the maximum expulsion value calculated by using Equation (2). Otherwise, the cache action will continue to be executed. More details about the caching and deletion of tasks are shown in Algorithm 1.

Algorithm 1: Task Caching and Deletion.

Input: K
Initialize the cache space of RSU as well as computing resource of RSU and SDV;
Initialize the actor network and critic network;
for each task k **do**
  compute the popularity P_k of each task k by Equation (1);
  obtain the caching action of task k by D3QN of Algorithm 2 based on state information of Equation (37);
  if $x_t = 1$:
    task is cached in cache area of RSU;
  else:
    if the popularity P_k of task k is maximum popularity:
      task is cached in cache area of RSU;
    else:
      task is not cached;
    end
  end
  if cache area if full:
  compute $P_{\exp}$ of the same type of task in the RSU by Equation (2);
  delete tasks belong to the same type corresponding to maximum $P_{\exp}$;
end

3.2.6. Joint Edge Computing and Caching Based on the D3QN

The specific process is as follows. First, we determine whether the task has encountered a cache hit; if so, it means that the task has already been cached, the RSU only needs to transmit the calculation results to the SDV and the agent can directly obtain the maximum reward value and the next state information. In this case, two caching models and the D3QN algorithm have no effect, and the computation offloading and caching of tasks will not be performed. If there is no cache hit, which means that the task cannot be found in the RSU, then it needs to be calculated and cached. First of all, we calculate the popularity of the task according to Equation (1), then determine whether the popularity is the maximum value. If so, the task will be cached in the cache area of the RSU. Secondly, the agent of the D3QN will generate offloading decisions according to the observed size of the task, the computing resources required of the task and other environmental information to generate offloading actions to decide where the task will be offloaded for calculation. After the task is computed completely, the corresponding results are also cached in the cache area. Finally, the agent obtains corresponding rewards and the next states. If the popularity of the task calculated by using Equation (1) is not the maximum value, then the agent will generate caching and offloading decisions according to the observed popularity and size of the task as well as the states of the RSU. Based on the action, the agent will choose whether to cache the task and where to offload the task. After an action is completed, the agent will obtain the corresponding reward value and next states. The reward, states, action and next states as four-tuple transformations of each slot are stored in the replay memory pool, when the number of iterations is greater than the size of the memory pool; the sampling was carried out using batch conversion for training. Afterwards, key network paameters are updated by using Equation (32) at regular intervals. The average revenue of the system is calculated when each step has been performed. More details of the proposed algorithm are shown in Algorithm 2.

Algorithm 2: Joint Computing and Caching Based on the D3QN.

Initialize system environment and replay memory pool;
Initialize the actor network and critic network;
for each episode **do**
  initialize system state s_t according to Equation (37), $r_t = 0$;
  for time step = {1,2, . . . , T} **do**
    while true:
      if $z_{k,t} = 1$:
        obtain next state s_{t+1} and reward r_t according to Equations (37) and (39);
        update state $s_t = s_{t+1}$;
      else:
        select caching and offloading action according to Equation (38) based on state information of system;
        obtain the next state and reward according to Equation (37) as well as Equation (39);
        store the transition (a_t, s_t, r_t, s_{t+1}) into replay memory;
        sample batch transitions from experience replay buffer;
        update the actor network and critic network by Equation (33);
        update state $s_t = s_{t+1}$;
      end
    end
  end
  calculate average revenue of system;
end

4. Simulation Results

4.1. Simulation Parameters

In this section, the performance of the proposed D3QN-based algorithm is introduced and then their performance with the DQN, DDQN and D2QN with or without the caching policy are compared. For different algorithms, we set the same training parameters: the learning rate γ affects the convergence performance and convergence speed of the whole network, and it was set is set to 0.001; the memory size D was set to 3000; the number of iterations was set to 30,000; too many iterations easily causes overfitting, and too few iterations easily make the training parameters not optimal. After continuous attempts, it was found that a memory size of 3000 had the best effect. The greed index was set to 0.99; through an extensive literature review, it can be seen that the greed index is generally 0.99, 0.95, 0.995, etc. However, it cannot be equal to 1, which poses a risk of "excessive Q value". The batch size B was set to 512; the larger the discount factor, the higher the chance that the current state value will be affected in the future. When the discount factor μ is set to 0.9, generally, there are more discrete actions, so the value will be larger; otherwise, smaller values will be selected. In this paper, each task has five discrete states and two discrete actions; there are fewer discrete actions, so μ is smaller. The network is updated every 200 times; both full flow layers are set to 128. Other simulation parameters related to the system environment are summarized in Table 1.

Table 1. Simulation parameters.

Parameter	Value
The number of SDVs N	4
The number of RSUs M	2
The number of tasks W in slot t	30
Delay tolerance of task $T^{\mathrm{m}}_{k,t}$ (s)	0.1
Size of task $I_{k,t}$ (KB)	8–10
Total computing resources of SDV $\psi_{v,t}$ (GHZ)	0.4
Total computing resources of MEC server $\psi_{m,t}$ (GHZ)	5
The type of tasks L	4

4.2. Experimental Results and Analysis

In the experiment, four algorithm models were set up and two environments were designed to prove the good performance of the proposed algorithm in terms of latency, energy consumption and reward. Here, maximizing system revenue is the purpose of this article; the higher the revenue, the better the performance. The latency and energy consumption of the system, as well as the unfinished rate of tasks and the utilization of cache resources, are taken as indicators to evaluate algorithm performance.

In Figure 3, we plot the average system revenue for eight scenarios in each episode. The icon "D3QN−cache" refers to the proposed joint edge computing and caching method based on the D3QN; icon "D3QN" refers to the D3QN algorithm; icon "D2QN" refers to the D2QN algorithm; "D2QN−cache" refers to the D2QN algorithm with caching models that are based on popularity as well as on the D2QN; and so on for the other icons. It can be seen from Figure 3 that the average revenue of D3QN−cache in each episode is much higher than that of other DRL algorithms, which means that D3QN−cache can better achieve the goal of maximizing system revenue. This is because in the D3QN, the RL agent uses double frame to observe the changed state of the environment and the reward due to the execution of the action, which makes the Q values more efficient than those of the D2QN, DDQN and DQN. In the D2QN, the RL agent uses the target network to obtain the action values of all actions in the next state, and then calculates the target value based on the optimal action value. Due to its maximization operation, there is an overestimation problem in the algorithm, which affects the accuracy of decision making. In the DDQN, the agent only cares about the action of updating the Q value function. However, the DQN has limited ability within the high-dimensional state and action space, and it is difficult to explore and find the optimal joint offload and cache strategy due to its limited capability within the high-dimensional state and action space. So compared to other DRL algorithms, the agent of the D3QN can find a better joint offload and cache policy than other DRL algorithms in accomplishing tasks. From 0 to 6000 times, the average system revenue of the proposed algorithm tends to decline, which is because the agent is constantly learning, exploring a good strategy and reaches a stable state about 6000 times. It can be seen that, compared with other DRL algorithms, the D3QN can converge quickly and reach a steady state more easily. This is because the passive cache model based on the D3QN is used to cache tasks, which can use the advantage function and target Q value for choosing a suitable strategy to execute the action of caching tasks, with the agent thus being prompted to explore a better strategy and action in order to obtain higher rewards. In order to reduce system latency and energy consumption, some appropriate tasks can be cached in RSUs, which can reduce the number of computation and transmission tasks, thus reducing system costs and improving system revenue. For this reason, the average revenue of a system with a DRL algorithm caching model is much higher than that of a system without a caching model. As can be seen from Figure 4, compared with DQN−cache, DDQN−cache, D2QN−cache, DQN, DDQN, D2QN and D3QN, the total average system revenue of D3QN−cache is increased by 65%, 35%, 66%, 257%, 142%, 312% and 57%, respectively. This shows that the proposed algorithm is more effective than other DRL algorithms.

An active cache based on the popularity and a passive cache based on the D3QN are used together; the parallel use of the two caching models enables more tasks to be cached in the RSU, and its calculation result is also retained. If the requested task can match the task in the cache area, the task does not produce any computation and transmission latency and energy consumption; it only generates the transmission latency and energy consumption of the transmission result. With an active cache, the most popular task will be cached in the cache area, and this will reduce the calculation of part of the tasks; however, when the cache area is enough, only caching the most popular task is not enough to maximize the use of the cache area, so a passive cache is proposed, using the caching policy of the D3QN to cache tasks that are not the most popular tasks in order to maximize the use of the cache space.

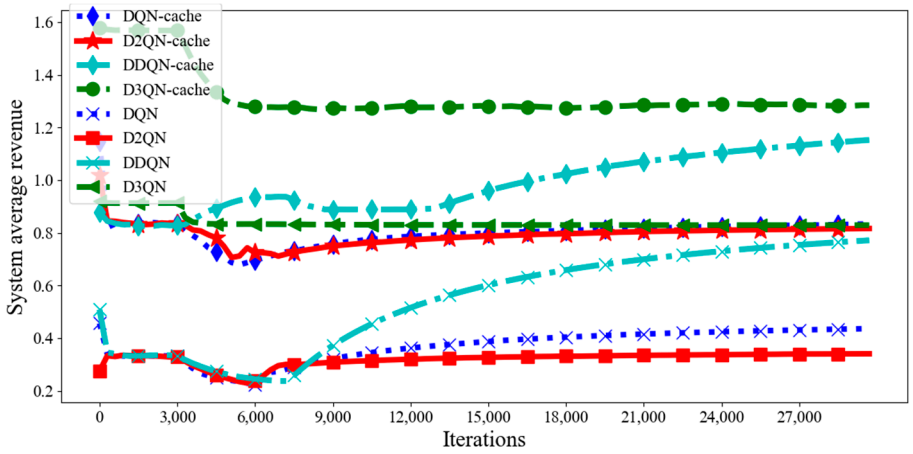

Figure 3. Average revenue in each episode with different DRL algorithms.

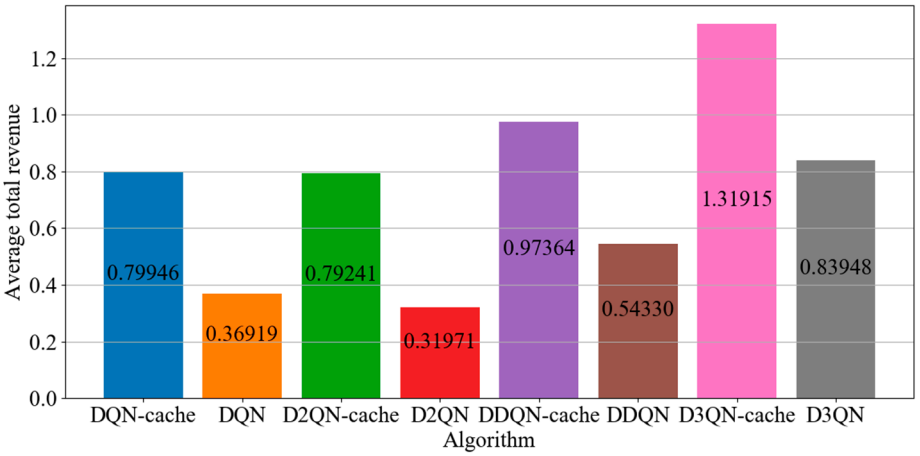

Figure 4. Average total revenue with different DRL algorithms.

The latency and energy consumption of the transmission results are very small and negligible compared to the computation and transmission tasks. Figures 5 and 6 illustrate that the average latency and energy consumption of the system with caching models are much lower than those without the caching policy. This is because we use two caching models to cache tasks together, which increases the probability that tasks will be cached. Caching as many tasks as possible without exceeding cache resources improves the utilization of cache resources and reduces system latency and energy consumption due to the numbers of computation and offloading tasks being reduced, which proves the necessity and importance of the caching model in computing tasks. In addition, the proposed algorithm always has much lower average latency and energy consumption than other DRL algorithms in each episode, and it is easier to reach a steady state. This is because the agent of the D3QN executes the cache action based on the greedy strategy to select an action that maximizes the reward and offloads tasks to an appropriate location for calculation based on the characteristics of the tasks. At the same time, distributed MEC enables nearby MEC servers to help with computing tasks, which relieves computing pressure. After calculation, compared with DQN−cache, DDQN−cache, D2QN−cache, DQN, DDQN, D2QN and D3QN, the total average system latency of the D3QN−cache is reduced by 53%, 42%, 53%, 66%, 59%, 66% and 49%, respectively, and the total average energy consumption of the system is reduced by 15%, 28%, 22%, 44%, 38%, 37% and 57%, respectively. Therefore, from what has been discussed above, the proposed algorithm can greatly reduce system latency and energy consumption.

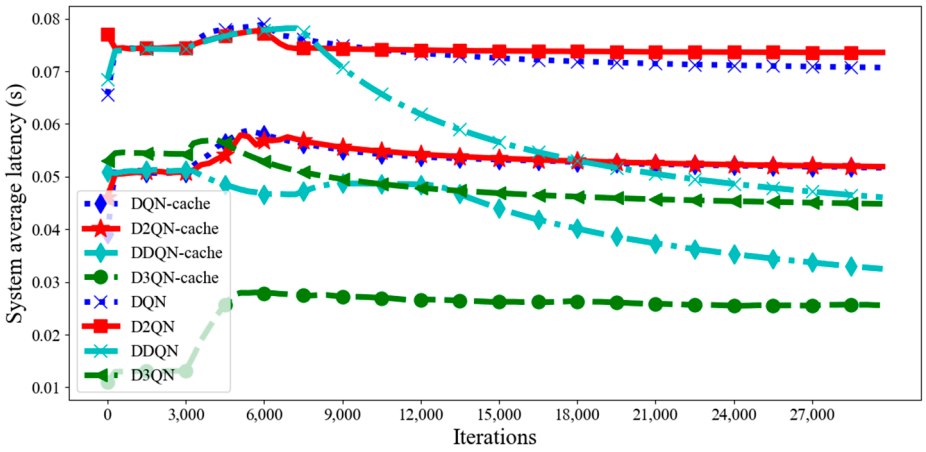

Figure 5. Average latency in each episode with different DRL algorithms.

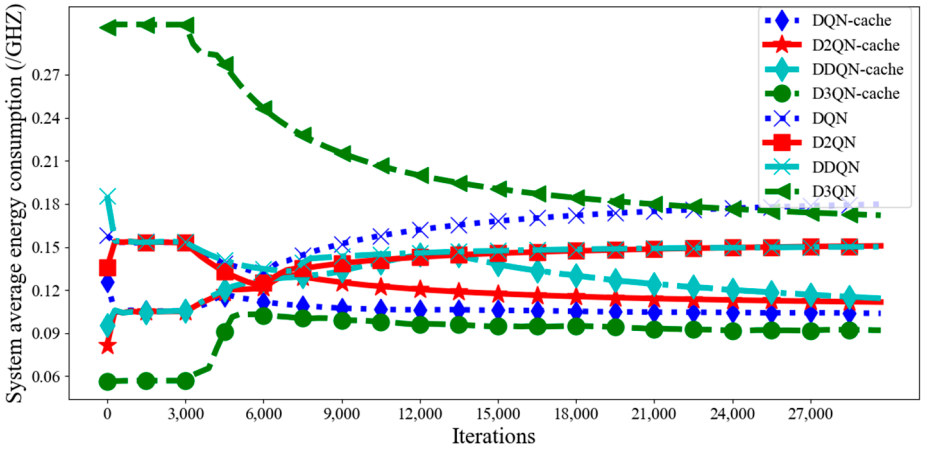

Figure 6. Average energy consumption in each episode with different DRL algorithms.

In the whole system, we chose to discard the unfinished task, with the agent then receiving a corresponding negative reward because the task was not completed. In order to receive a more positive reward, the agent chose an appropriate action to complete the task.

$$P_{\mathrm{U}} = \frac{\sum_{k \in \mathbf{K}} \zeta(u_{k,t} = 1)}{W} \tag{41}$$

where P_{U} is probability of unfinished tasks, $u_{k,t} = 1$ indicates that task k is not completed and $k \in \mathbf{K}$, W is the number of requested tasks in slot t.

Figure 7 shows the probability of unfinished tasks under the eight algorithms in each slot. As can be seen from Figure 7, the probability of unfinished tasks with the D3QN algorithm is much lower than that with other algorithms. This is because the D3QN not only uses the effect of the state on the Q value, but also the effect of the action on the Q value. When the task needs to be cached, the state has a great impact on the Q value. This is seen with the popularity of the task, the size of the task and other states having a great impact on the Q value. Additionally, when a task needs to be offloaded, the action has a great impact on the Q value. The choice of which action produces a greater revenue also greatly affects the Q value. Combined with the advantages of both, the performance of the D3QN is improved. In addition, it changes the way that it chooses the target Q value by using the action of the target Q-value, which eliminates overestimation. Compared with the D3QN, although the probability of unfinished tasks with D3QN−cache is higher at the beginning, when the number of iterations increases, the probability of unfinished tasks with D3QN−cache keeps decreasing, because the agent is constantly exploring better actions to complete tasks. However, a lower probability of unfinished tasks with D3QN

is at the cost of high latency, high energy consumption and low profit; in our study, the agent of the D3QN chose a high-pay, low-return action to complete the task. Additionally, when the number of iterations increases, the agent in pursuit of a high profit also begins to explore actions that provide low latency and low energy consumption, so the probability of unfinished tasks begins to rise. Therefore, considering the latency, energy consumption, revenue and probability of unfinished tasks, the proposed algorithm is better.

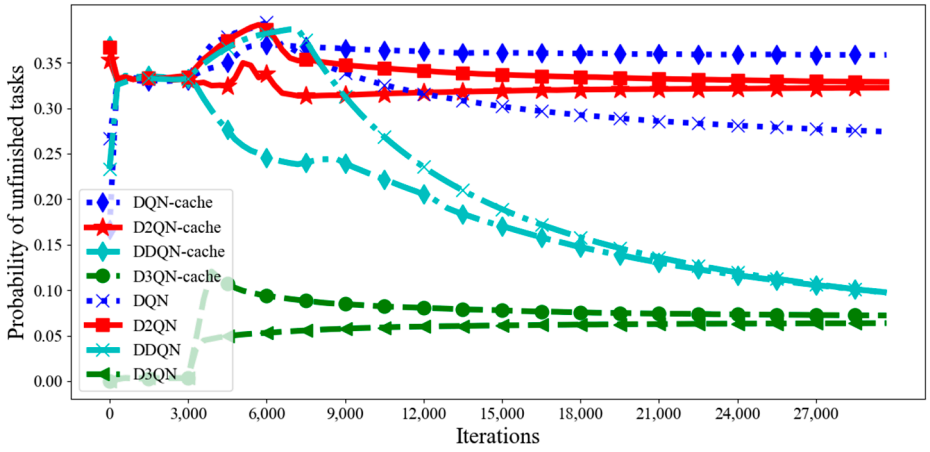

Figure 7. Probability of unfinished tasks in each episode with different DRL algorithms.

In order to complete tasks as much as possible and obtain maximum system revenue, it is necessary to make full use of the cache space while not exceeding cache capacity, completing as many cache tasks as possible in the process. The utilization of cache area is the ratio of the size of all tasks in the cache area-to-the size of the cache space. The utilization of the cache area can be calculated as

$$U = \frac{\sum_{k \in \mathbf{K}} \zeta(h_{k,t} = 1) I_{k,t}}{C} \tag{42}$$

where $h_{k,t}$ represents the caching decision of task k in slot t, $I_{k,t}$ represents the size of task k in slot t and C represents the size of the cache space. As long as the size of the cache space is not exceeded, this number should be as large as possible.

Figure 8 shows the average utilization of the cache area under the four algorithms. As can be seen from Figure 8, compared with DQN−cache, D2QN−cache and DDQN−cache, D3QN−cache, that is, the proposed algorithm, has the highest average utilization of cache space, with its average utilization being improved by 28%, 42% and 36%. It makes use of two caching models, the active cache and passive cache, and not only caches the most popular tasks, but also uses the D3QN to cache other tasks. The cache difference between these four algorithms is a result of the different DRL algorithms used in the passive cache. In the proposed algorithm, the best action is selected to cache more tasks according to the advantages of the D3QN mentioned above; by doing so, the utilization of the cache space is improved.

The average utilization of the cache space of D3QN−cache is only 63% because when cache resources are exhausted, in order to release a large amount of idle cache resources, some tasks of a certain type in the cache space need to be deleted, which means a large number of tasks will be deleted, which will lead to a sharp decrease in the number of tasks in the cache space. In each episode or several episodes, cache resources are released, which results in a low average utilization of cache resources. However, the advantage of D3QN−cache is that the number of task deletions and operational burden on the server are reduced.

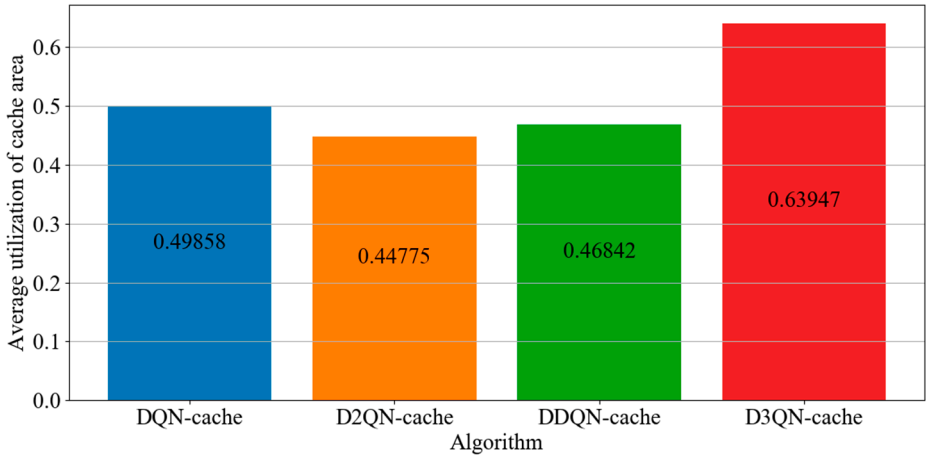

Figure 8. Average utilization of cache area in each episode with different DRL algorithms.

4.2.1. The Influence of Different Training Parameters in D3QN−Cache

We compared the effects of batch size, memory size, gamma and learning rate (lr) on simulation results. The batch size, memory size, gamma and learning rate of D3QN−cache are 512, 3000, 0.99 and 0.0001, respectively. Using a larger batch size allows the agent to perform parallel calculations to a greater extent, and less time being required for training in each period can significantly accelerate model training. However, the larger the batch size, the slower the reduction of training losses, and the overall training time becomes longer. The memory records the previous training experience. When similar states occur, the value function network can be updated based on the previous successful experience, and the model will learn faster, whereas if there is too little experience stored, some good experiences will be discarded. If there is too much experience stored, some bad experiences will be learned. Therefore, to improve profits, it is necessary to choose a suitable memory size. If the gamma value is not large, only the current reward will be considered, which means adopting a short-sighted strategy. If we want to balance current and future rewards, a larger gamma value will be considered, which means that the long-term future reward will be taken into account in the value generated by the current behavior. However, if the gamma value is too large, it is considered too long-term, and even beyond the range that the current behavior can affect, which is obviously unreasonable. The learning rate controls the learning progress of the model and determines the time when the network obtains the global optimal solution. The learning rate being set too high can cause the network to not converge, but the learning rate being set too low can reduce the speed of network convergence and increase the time taken to find the optimal value. Therefore, in summary, from Figures 9 and 10, it can be seen that the training parameters of the D3QN are optimal for the average system revenue.

From Figure 11, it can be seen that the system average latency of D3QN−cache is much lower than that of other parameters, because a larger batch size can accelerate the convergence speed of the model through a large number of parallel calculations, and the memory size is 3000, which means that the empirical data is not too much or too little, and the agent learns some good behaviors and ignores some bad ones. Additionally, the gamma is set as 0.99, and this can effectively balance the relationship between current rewards and future rewards; furthermore, the learning rate of 0.0001 can quickly find the optimal parameters of the model. Although the average latency of D3QN−cache is the lowest, it does not mean that the energy consumption is the lowest. From Figure 12, it can be seen that the average energy consumption of D3QN−cache did not reach the minimum. This is because D3QN−cache aims to maximize the optimization problem, which involves cache costs. To reduce cache costs, energy consumption must be sacrificed.

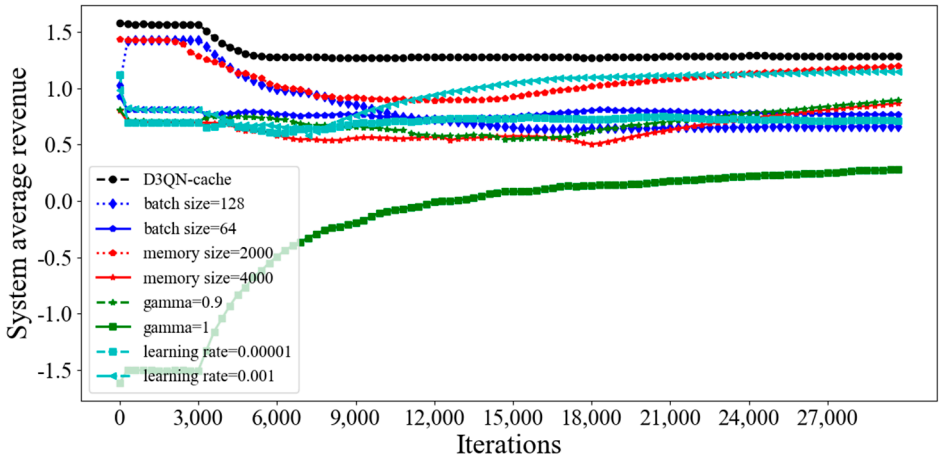

Figure 9. Average system revenue for different training parameters in D3QN−cache.

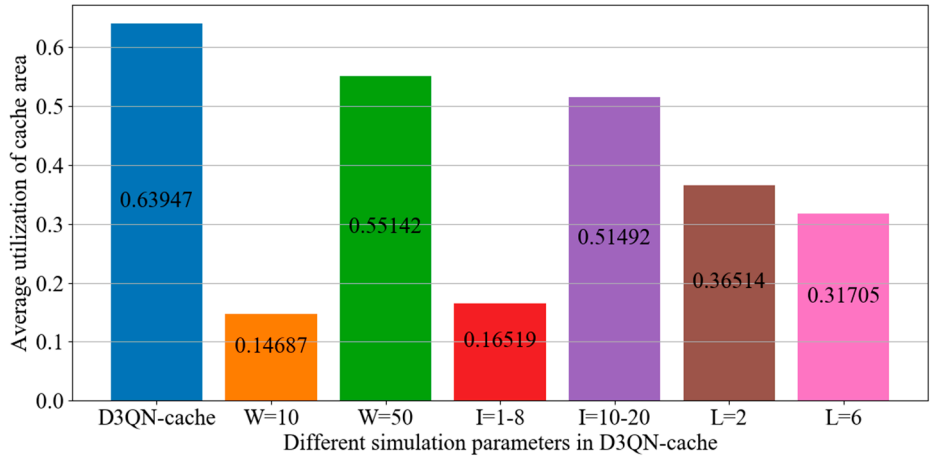

Figure 10. Average total revenue for different training parameters in D3QN−cache.

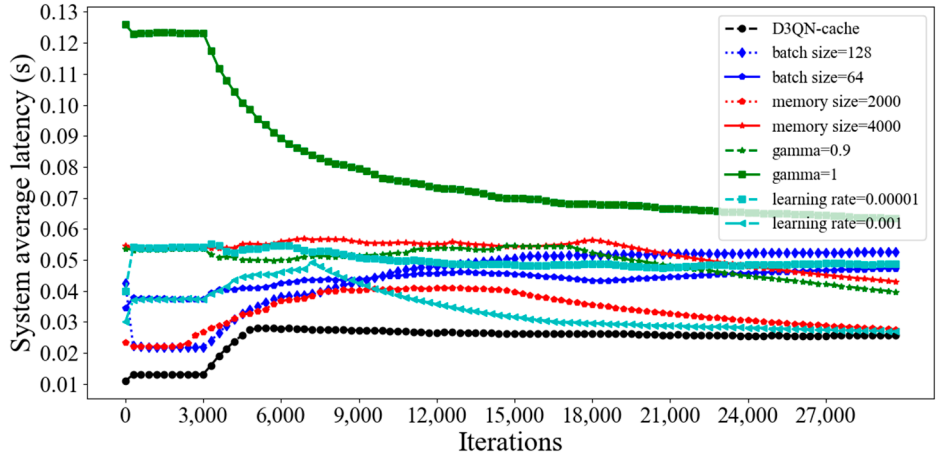

Figure 11. Average system latency for different training parameters in D3QN−cache.

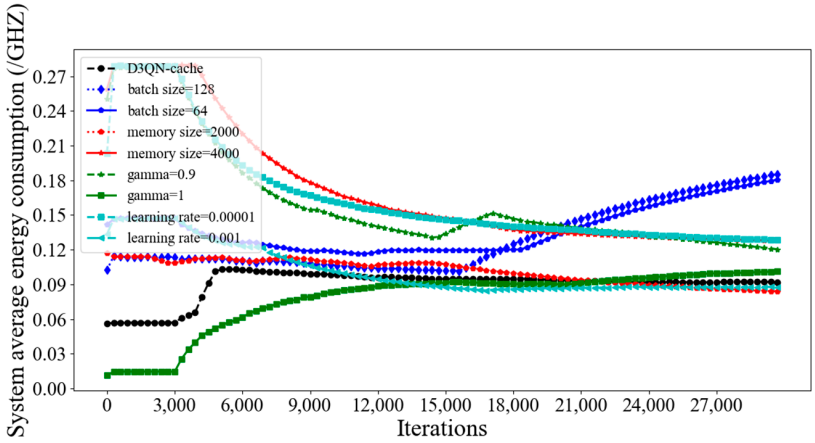

Figure 12. Average system energy consumption for different training parameters in D3QN−cache.

Figures 13 and 14 compare the probability of unfinished tasks and utilization of cache space for different training parameters in D3QN−cache, respectively. As the batch size increases, the speed of processing the same amount of data increases, but the number of epochs required to achieve the same accuracy increases. Due to the contradiction between the two factors, the batch size must be increased to a certain value in order to achieve the optimal time. From Figures 13 and 14, it can be seen that when the batch size is 512, the agent will find the optimal joint decision of offloading and caching at the fastest speed, resulting in the lowest probability of unfinished tasks and the highest utilization of cache space. If the memory size is very small, there will be very little experience, which is not conducive to training. However, if there is too much experience, which leads to useless experiences being added to the training, it is clearly not conducive to training. From Figure 13, it can be seen that a memory size of 3000 has the best effect, but from Figure 14, it can be seen that utilization of cache space with a memory size of 2000 exceeds D3QN−cache, which indicates that a memory size of 2000 is the most advantageous for cache decision making. However, we are considering the overall system revenue, so in order to achieve this goal, some cache costs must be sacrificed. The gamma being equal to 1 means that the algorithm lacks convergence ability and the agent will consider possibilities, which will increase complexity. However, when the gamma value is too small, this means that the agent only focuses on current information and has a poor estimation of action. Therefore, it can be seen from Figure 13 that the probability of unfinished tasks with gamma values of 1 and 0.9 is lower, and the utilization of cache space as shown in Figure 14 is also lower. If the learning rate (lr) is set too high, it will lead to overfitting and a poor convergence effect, but if it is set too small, the convergence speed will be slow. From Figures 13 and 14, it can be seen that the probability of unfinished tasks and utilization of cache space with the learning rates sets as 0.00001 and 0.001 are lower.

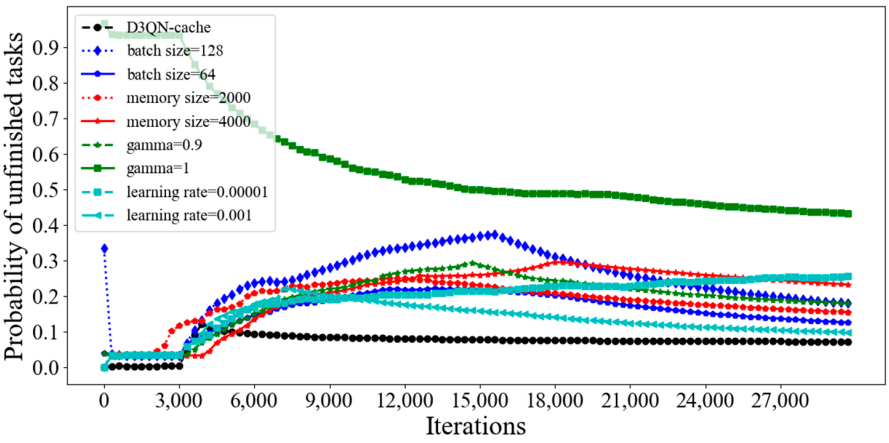

Figure 13. The probability of unfinished tasks for different training parameters in D3QN−cache.

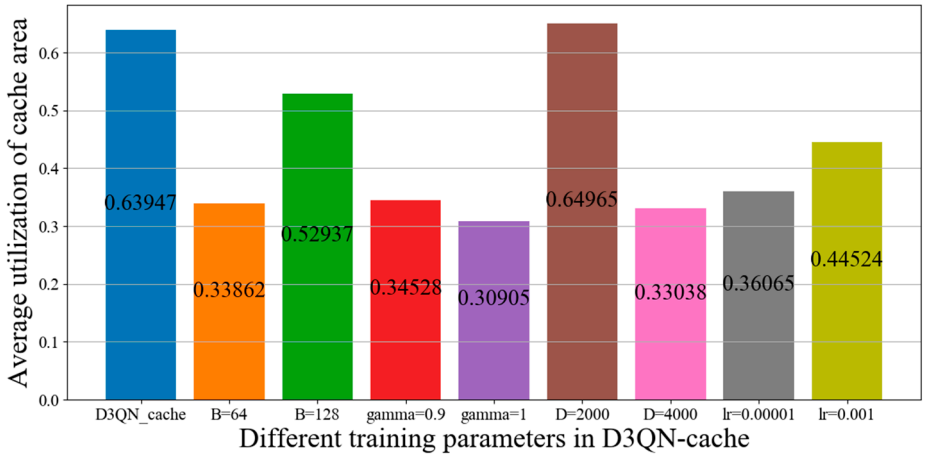

Figure 14. The utilization of cache space for different training parameters in D3QN−cache.

4.2.2. The Influence of Different Simulation Parameters in D3QN−Cache

We compared the impact number *W*, size *I* and type *L* of tasks on the simulation results. The number *W*, size *I* and type *L* of tasks for D3QN−cache are 30, 8–10 and 4. From Figures 15 and 16, it can be seen that the average revenue for D3QN−cache is the highest for the following reasons. When the number of tasks increases from 10 to 30, the average revenue will greatly increase. This is because there are more tasks to process, which leads to more revenue. However, when the number of tasks increases from 30 to 50, the revenue will decrease. This is because with more tasks, the greater the pressure of the backhaul link and the greater the number of computing resources required. However, the computing resources of SDVs and RSUs are limited and cannot handle a large number of tasks. This will result in a decrease in revenue. When the size of the task decreases from 10–20 KB to 8–10 KB, the average revenue will greatly increase. This is because the larger the size of the task, the greater the number of resources that are required, making it is impossible to calculate and store tasks that are too large. When the size of the task decreased from 8–10 KB to 1–8 KB, the average revenue decreased, but it did not decrease much. This is because the currently designed resources of SDVs and RSUs are suitable for sizes of between 1–10 KB. Since the deletion rule of the task is type deletion, the type of task will affect the cache hit rate of each task. When there are very few types of tasks, each type will contain many tasks, so when performing task deletion, it will cause a large number of tasks to be deleted, resulting in a decrease in the cache hit rate and average revenue. When there are many types of tasks, it indicates that the number of same tasks is very small, and the experience data that the agent can obtain are very small, which can lead to a lower average revenue. Therefore, the average revenue for *L* is 4 times higher, and for *L* it is two and six times higher.

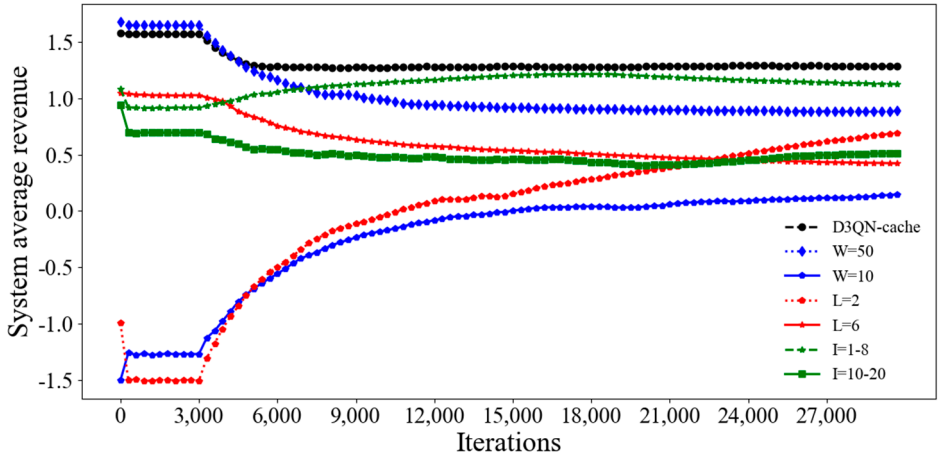

Figure 15. Average system revenue for different simulation parameters in D3QN−cache.

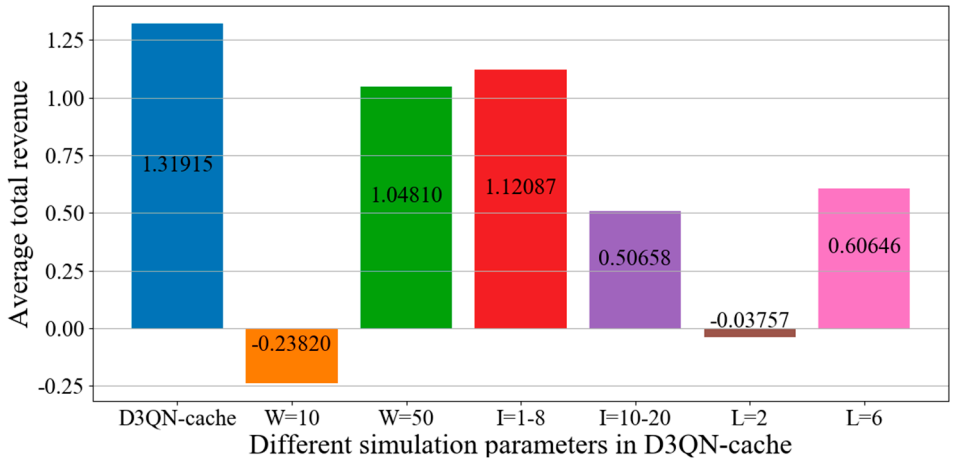

Figure 16. Total average revenue for different simulation parameters in D3QN−cache.

Figures 17 and 18 compare the average latency and energy consumption of D3QN−cache under different simulation parameters. Too many tasks can lead to an increase in action space and state space, making it more difficult to find the optimal solution, thus resulting in higher latency and energy consumption. However, if the number of tasks is too small, the agent cannot obtain a large amount of experience data and cannot make optimal strategies, resulting in high latency and energy consumption. So, from Figures 17 and 18, it can be seen that the latency and energy consumption of D3QN−cache are lower than those of *W*, which are 10 and 50. The size of the task affects latency and energy consumption. The larger the size of the task, the more offloading latency and computing resources it can occupy, leading to increase in system latency and energy consumption. So, the latency and energy consumption of tasks with a size of 1–8 KB are lower than those of the D3QN. The type of task can affect the cache hit rate, leading to changes in latency and energy consumption. Although the energy consumption and latency of D3QN−cache are not the lowest, the proposed algorithm focuses on long-term system revenues rather than the energy consumption and latency per slot, demonstrating the importance of considering long-term optimization in dynamic networks.

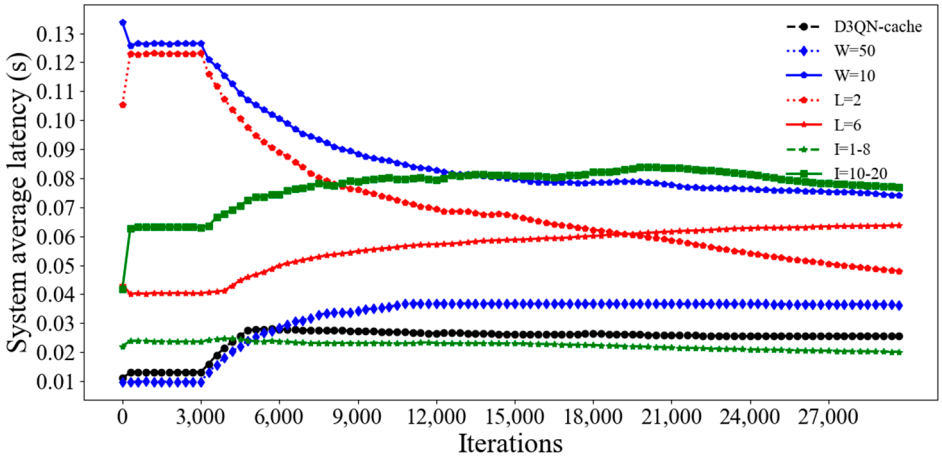

Figure 17. Average system latency for different simulation parameters in D3QN−cache.

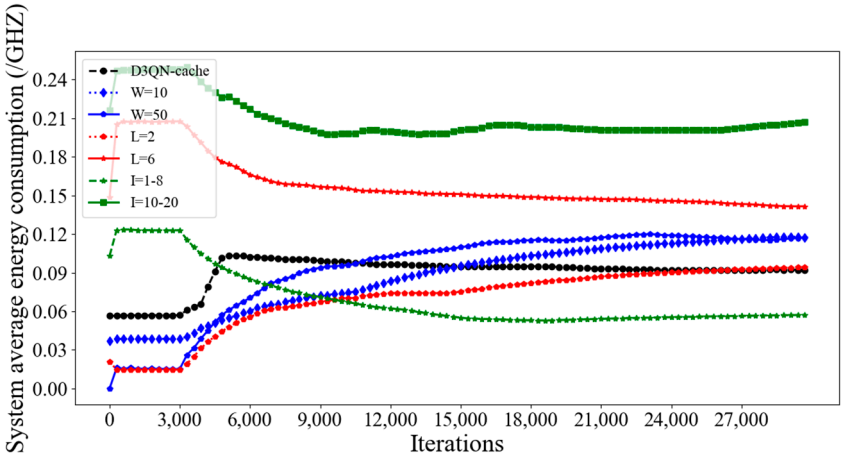

Figure 18. Average system energy consumption for different simulation parameters in D3QN−cache.

Figures 19 and 20 compare the probability of unfinished tasks and utilization of cache space of D3QN−cache under different simulation parameters, respectively. If the number of tasks is too large, it will result in a high demand for resources. However, with limited computing and storage resources, the probability of not being able to complete tasks will increase, and the number of tasks cached in the RSUs will also decrease. If the number of tasks is small, the experience data obtained by the agent will be small, making it difficult to find the optimal decisions, so the number of cached tasks will also decrease. Therefore, from Figures 19 and 20, it can be seen that the probability of unfinished tasks with a task number of 30 is lower than that when W is 10 and 50, and the utilization of the cache space of D3QN−cache is higher than that when W is 10 and 50. The fewer types of tasks, the larger the number of tasks of a certain type. Due to the type of deletion rule, there are fewer tasks in the cache space, resulting in a lower utilization of cache space and cache hit rates; therefore, the probability of unfinished tasks is high. The fewer task types, the fewer tasks deleted by the type of deletion rule, and the more tasks in the cache space and less remaining cache resources, resulting in the delayed caching of subsequent tasks. So, from Figures 19 and 20, it can be seen that the probability of unfinished tasks with L is 4 times lower, and for L it is two and six times higher. The larger the size of the task, the greater the likelihood of content delivery failure within a fixed deadline, which can result in penalty costs. So, the probability of unfinished tasks with a task size of 1–8 KB is the lowest. However, the larger the size of the task, the larger the cache resources it occupies, resulting in a higher utilization of cache space.

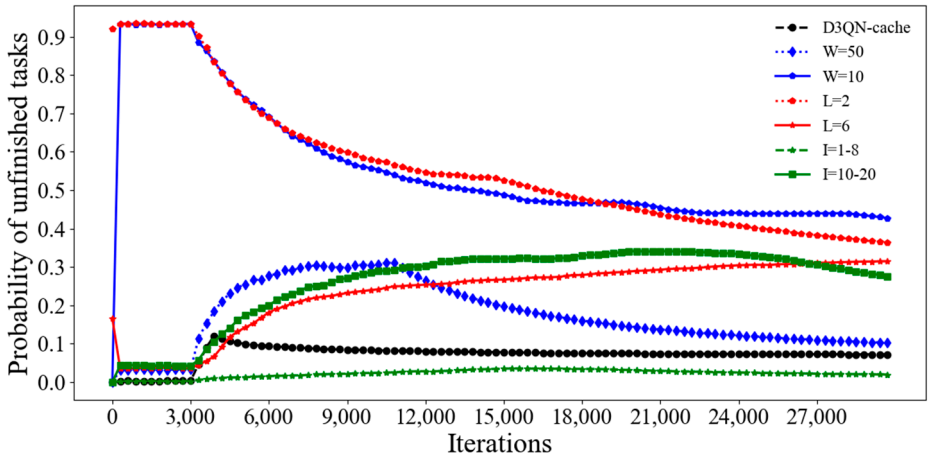

Figure 19. The probability of unfinished tasks for different simulation parameters in D3QN−cache.

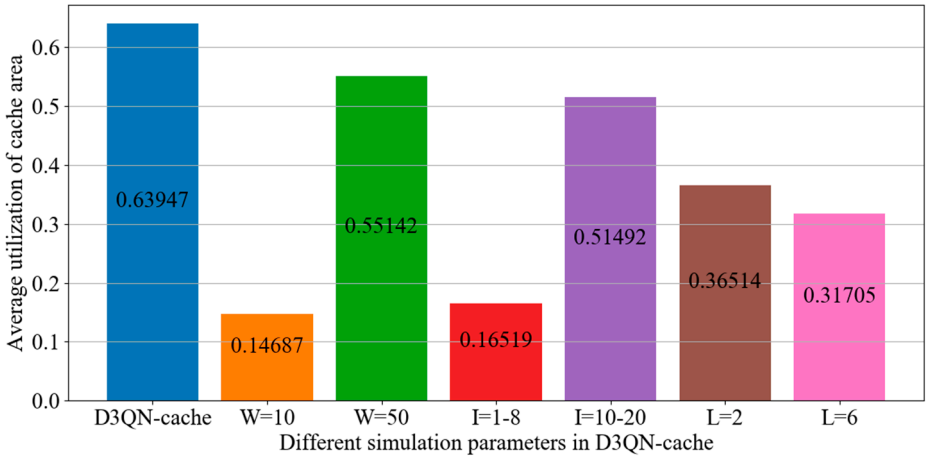

Figure 20. The utilization of cache space for different simulation parameters in D3QN−cache.

As can be seen from the simulation results in Section 4.2.1, the training parameters we selected perform well in several evaluation indexes, such as average reward. Although the effects of some evaluation indexes are not very good, they do not affect the overall effect of the system. It can be seen from Section 4.2.2 that the simulation parameters we designed are appropriate. If the malleability of the proposed algorithm is to be expanded, the number of RSUs can be increased to enhance the auxiliary computing capacity between them.

5. Conclusions

In this paper, we introduced a MEC-enabled caching and offloading system, where RSUs with caching and computing capabilities serve SDVs collaboratively to compute tasks. In order to address the raised problem of joint edge computing offloading and caching in multi-task function of the IOV, a joint edge computing and caching method based on the D3QN is proposed. First, through the computational offloading and caching of tasks, we model a joint optimization problem for maximization of system revenue, which is constrained by system latency and energy consumption as well as the cache space of the RSU, in addition to also being affected by the probability of task uncompletion. Secondly, an active cache and passive cache are proposed, with the former based on the popularity of the cache tasks, and the latter based on the D3QN cache tasks. The parallel use of the two cache models solves the proposed NP-hard problem. Additionally, the expulsion value based on the type of popularity is used to update the tasks of the cache space. In addition, according to the state of the system, the optimal action of the offloading task is determined using the e-greedy strategy. Finally, the simulation results show that the proposed algorithm has verified good performance in the aspects of system delay, energy consumption and average revenue. Additionally, the utilization of cache space of the RSU is improved greatly due to the simultaneous application of the active cache and passive cache, and the probability of incomplete tasks is reduced due to the better decisions of the D3QN. Compared with DQN−cache, DDQN−cache, D2QN−cache, DQN, DDQN, D2QN and D3QN, the proposed algorithm improves the average revenue of the system by 65%, 35%, 66%, 257%, 142%, 312% and 57%, respectively.

This article only discusses the case of a small number of vehicles and RSUs, but the results can be extended to larger IOV scenarios by replacing RSUs with edge nodes and increasing the number of vehicles. The tasks generated by vehicles within the coverage range of each edge node are treated as a set of tasks, and edge nodes that can communicate with each other are treated as set of cMECs. This allows multiple servers to collaborate with each other to achieve large-scale caching and offloading functions. The results of this article are not only applicable to computing tasks of the IOV, but also to content data such as text or video within the Internet of Things. Content requests sent by user terminals can

also be cached and processed. Through multi-agent collaboration, content can be cached to different base stations or edge nodes, thereby reducing user request costs and improving user service quality. The caching and offloading of computing tasks in the IOV are only discussed in this article, other forms of user requests are not considered. In addition, two reactive caching models are only designed to perform caching operations when tasks arrive, whereas a proactive caching model is not designed for cache tasks in advance. Therefore, in our future work, we will analyze how to make decisions that combine proactive caching and reactive caching in more complex network scenarios with multiple heterogeneous services to support different Vehicle-to-Everything services.

Author Contributions: Conceptualization, G.C.; data curation, J.S.; formal analysis, G.C. and J.S.; methodology, Q.Z.; validation, G.J. and Y.Z.; writing—original draft, G.C.; writing—review and editing, J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China under Grant No. 61701284, the Natural Science Foundation of Shandong Province of China under Grant No. ZR2022MF226, the Talented Young Teachers Training Program of Shandong University of Science and Technology under Grant No. BJ20221101, the Innovative Research Foundation of Qingdao under Grant No. 19-6-2-1-cg, the Elite Plan Project of Shandong University of Science and Technology under Grant No. skr21-3-B-048, the Hope Foundation for Cancer Research, UK under Grant No. RM60G0680, the Royal Society International Exchanges Cost Share Award, UK under Grant No. RP202G0230, the Medical Research Council Confidence in Concept Award, UK under Grant No. MC_PC_17171, the British Heart Foundation Accelerator Award, UK under Grant No. AA/18/3/34220, the Sino-UK Industrial Fund, UK under Grant No. RP202G0289, the Global Challenges Research Fund (GCRF), UK under Grant No. P202PF11, the Guangxi Key Laboratory of Trusted Software under Grant No. kx201901, the Sci. & Tech. Development Fund of Shandong Province of China under Grant No. ZR202102230289, ZR202102250695 and ZR2019LZH001, the Humanities and Social Science Research Project of the Ministry of Education under Grant No. 18YJAZH017, the Taishan Scholar Program of Shandong Province under Grant No.ts20190936, the Shandong Chongqing Science and technology cooperation project under Grant No. cstc2020jscx-lyjsAX0008, the Sci. & Tech. Development Fund of Qingdao under Grant No. 21-1-5-zlyj-1-zc, the SDUST Research Fund under Grant No. 2015TDJH102, the Science and Technology Support Plan of Youth Innovation Team of Shandong higher School under Grant No. 2019KJN024.

Data Availability Statement: Data sharing not applicable.

Acknowledgments: The authors are grateful to the anonymous reviewers for providing us with so many valuable comments and suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Muniswamaiah, M.; Agerwala, T.; Tappert, C.C. A Survey on Cloudlets, Mobile Edge, and Fog Computing. In Proceedings of the 2021 8th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2021 7th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), Washington, DC, USA, 26–28 June 2021; pp. 139–142.
2. Wang, K.; Wang, X.; Liu, X.; Jolfaei, A. Task Offloading Strategy Based on Reinforcement Learning Computing in Edge Computing Architecture of Internet of Vehicles. *IEEE Access* **2020**, *8*, 173779–173789. [CrossRef]
3. Yang, Z.; Liu, Y.; Chen, Y.; Al-Dhahir, N. Cache-aided NOMA mobile edge computing: A reinforcement learning approach. *IEEE Trans. Wirel. Commun.* **2020**, *19*, 6899–6915. [CrossRef]
4. Amadeo, M.; Campolo, C.; Ruggeri, G.; Molinaro, A. Beyond Edge Caching: Freshness and Popularity Aware IoT Data Caching via NDN at Internet-Scale. *IEEE Trans. Green Commun. Netw.* **2022**, *6*, 352–364. [CrossRef]
5. Ma, X.; Zhou, A.; Zhang, S.; Wang, S. Cooperative Service Caching and Workload Scheduling in Mobile Edge Computing. In Proceedings of the IEEE INFOCOM 2020—IEEE Conference on Computer Communications, Toronto, ON, Canada, 6–9 July 2020; pp. 2076–2085.
6. Yao, I.; Ansari, N. Joint Content Placement and Storage Allocation in C-RANs for IoT Sensing Service. *IEEE Internet Things J.* **2019**, *6*, 1060–1067. [CrossRef]
7. Zhang, T.; Fang, X.; Liu, Y.; Li, G.Y.; Xu, W. D2D-enabled mobile user edge caching: A multi-winner auction approach. *IEEE Trans. Veh. Technol.* **2019**, *68*, 12314–12328. [CrossRef]
8. Li, Q.; Zhang, Y.; Li, Y.; Xiao, Y.; Ge, X. Capacity-aware edge caching in fog computing networks. *IEEE Trans. Veh. Technol.* **2020**, *8*, 9244–9248. [CrossRef]

9. Kong, X.; Duan, G.; Hou, M.; Shen, G.; Wang, H.; Yan, X.; Collotta, M. Deep Reinforcement Learning-Based Energy-Efficient Edge Computing for Internet of Vehicles. *IEEE Trans. Industr. Inform.* **2022**, *9*, 6308–6316. [CrossRef]
10. Zhang, F.; Han, G.; Liu, L.; Martínez-García, M.; Peng, Y. Joint Optimization of Cooperative Edge Caching and Radio Resource Allocation in 5G-Enabled Massive IoT Networks. *IEEE Internet Things J.* **2021**, *8*, 14156–14170. [CrossRef]
11. Li, W.; Wang, J.; Zhang, G.; Li, L.; Dang, Z.; Li, S. A Reinforcement Learning Based Smart Cache Strategy for Cache-Aided Ultra-Dense Network. *IEEE Access* **2019**, *7*, 39390–39401. [CrossRef]
12. Ren, J.; Wang, H.; Hou, T.; Zheng, S.; Tang, C. Collaborative Edge Computing and Caching with Deep Reinforcement Learning Decision Agents. *IEEE Access* **2020**, *8*, 120604–120612. [CrossRef]
13. Tang, M.; Wong, V.W.S. Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems. *IEEE Trans. Mob. Comput.* **2022**, *21*, 1985–1997. [CrossRef]
14. Tian, H.; Xu, X.; Qi, L.; Zhang, X.; Dou, W.; Yu, S.; Ni, Q. CoPace: Edge Computation Offloading and Caching for Self-Driving with Deep Reinforcement Learning. *IEEE Trans. Veh. Technol.* **2021**, *70*, 13281–13293. [CrossRef]
15. Shi, J.; Zhao, L.; Wang, X.; Zhao, W.; Hawbani, A.; Huang, M. A Novel Deep Q-Learning-Based Air-Assisted Vehicular Caching Scheme for Safe Autonomous Driving. *IEEE Trans. Intell. Transp. Syst.* **2021**, *22*, 4348–4358. [CrossRef]
16. Qian, Y.; Wang, R.; Wu, J.; Tan, B.; Ren, H. Reinforcement Learning-Based Optimal Computing and Caching in Mobile Edge Network. *IEEE J. Sel. Areas Commun.* **2020**, *38*, 2343–2355. [CrossRef]
17. Li, Y.; Qi, F.; Wang, Z.; Yu, X.; Shao, S. Distributed Edge Computing Offloading Algorithm Based on Deep Reinforcement Learning. *IEEE Access* **2020**, *8*, 85204–85215. [CrossRef]
18. Ning, Z.; Zhang, K.; Wang, X.; Guo, L.; Hu, X.; Huang, J.; Hu, B.; Kwok, R.Y.K. Intelligent Edge Computing in Internet of Vehicles: A Joint Computation Offloading and Caching Solution. *IEEE Trans. Intell. Transp. Syst.* **2021**, *22*, 2212–2225. [CrossRef]
19. Wu, Z.; Yan, D. Deep reinforcement learning-based computation offloading for 5G vehicle-aware multi-access edge computing network. *China Commun.* **2021**, *18*, 26–41. [CrossRef]
20. Zhang, S.; Wu, Y.; Ogai, H.; Inujima, H.; Tateno, S. Tactical Decision-Making for Autonomous Driving Using Dueling Double Deep Q Network with Double Attention. *IEEE Access* **2021**, *9*, 151983–151992. [CrossRef]
21. Naeem, F.; Seifollahi, S.; Zhou, Z.; Tariq, M. A Generative Adversarial Network Enabled Deep Distributional Reinforcement Learning for Transmission Scheduling in Internet of Vehicles. *IEEE Trans. Intell. Transp. Syst.* **2021**, *22*, 4550–4559. [CrossRef]
22. Nguyen, D.C.; Pathirana, P.N.; Ding, M.; Seneviratne, A. Secure Computation Offloading in Blockchain Based IoT Networks with Deep Reinforcement Learning. *IEEE Trans. Netw. Sci. Eng.* **2021**, *8*, 3192–3208. [CrossRef]
23. Huang, Y.; Wei, G.; Wang, Y. V-D D3QN: The Variant of Double Deep Q-Learning Network with Dueling Architecture. In Proceedings of the 2018 37th Chinese Control Conference (CCC), Wuhan, China, 25–27 July 2018; pp. 9130–9135.
24. Jiang, S.; Huang, Y.; Jafari, M.; Jalayer, M. A Distributed Multi-Agent Reinforcement Learning with Graph Decomposition Approach for Large-Scale Adaptive Traffic Signal Control. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 14689–14701. [CrossRef]