A Task Offloading Algorithm With Cloud Edge Jointly Load Balance Optimization Based on Deep Reinforcement Learning for Unmanned Surface Vehicles

LINJIE YAN\(^1\), HAIMING CHEN\(^1,2\), (Member, IEEE), YOUPENG TU\(^1\), AND XINYAN ZHOU\(^1\)

\(^1\)Faculty of Electrical Engineering and Computer Science, Ningbo University, Zhejiang 315211, China
\(^2\)Zhejiang Provincial Key Laboratory of Mobile Network Application Technology, Ningbo, Zhejiang 315211, China

Corresponding author: Haiming Chen (chenhaiming@nbu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62002183, and in part by the Ningbo Natural Science Foundation under Grant 2021J090 and Grant 202003N4087.

**ABSTRACT** Unmanned Surface Vehicles (USVs) generate a large amount of data that needs to be processed in real time when they work, but they are usually limited by computational and battery resources, so they need to offload their tasks to the edge for processing. However, when numerous USVs offload their tasks to the edge nodes, some offloaded tasks may be thrown due to queuing timeouts. Existing task offloading methods generally consider the latency or the overall system energy consumption caused by the collaborative processing at the edge and end layers, and do not consider the wasted energy when the tasks are thrown. Therefore, to address the above situation, this paper establishes a task offloading model to minimize long-term task latency and energy consumption by jointly considering the requirements of latency and energy-sensitive tasks and the overall load dynamics in the cloud, edge, and end layers. A deep reinforcement learning (DRL)-based Task Offloading with Cloud Edge Jointly Load Balance Optimization algorithm (TOLBO) is proposed to select the best edge server or cloud server for offloading. Simulation results show that the algorithm can improve the utilization of energy consumption of the cloud edge nodes compared with other algorithms. At the same time, it can significantly reduce the task throw rate, average latency, and energy consumption of end devices.

**INDEX TERMS** USV, mobile edge computing, offloading delay, energy consumption, DRL.

**I. INTRODUCTION**
The Unmanned Surface Vehicle (USV) is a kind of intelligent surface ship that can carry out autonomous navigation and mobile surveillance tasks. Compared with traditional boats, USV has the advantages of low cost and multiple functions. They can perform tasks in complex and hazardous water environments, which reveal a wide range of application scenarios in maritime cruise [1], environmental monitoring [2], anti-submarine, and other fields.

The complex and changing ocean environment, unpredictable navigational emergencies, and multi-source sensor data are all factors that place high demands on the intelligent processing capability of USVs. In recent years, deep learning (DL) in artificial intelligence has achieved great success in various fields, which also provides an opportunity to enhance the intelligence of USVs. For example, USVs can effectively extract features from the data collected by sensors through convolutional neural networks, which accelerates USVs’ intelligence progress and puts high demands on USVs’ storage and computation capabilities. However, limited by the local computing ability and power supply of USVs, USVs cannot handle these resource-intensive intelligence tasks quickly [3], so most of the intelligence tasks are still handled by traditional cloud-based centralized solutions. This solution provides scalable resources to handle these intelligent tasks, but the cloud servers are often far from the data source. The considerable data communication overhead and latency caused by traffic congestion will be the critical bottleneck of the cloud computing model. Driven by this
problem, edge computing is emerging as a practical solution that distributes computing and storage resources near IoT devices [4]. Thus the data generated by USVs that need to be processed can be directly offloaded to nearby edge servers. With this approach, the data transmission latency can be significantly reduced, and congestion in the network can be alleviated.

Although edge computing has many advantages, existing studies have shown that there are significant challenges in guaranteeing stable operation of edge servers over long periods [5], such as satisfying multiple user requests with high-performance requirements at the same time. With the increase of computational tasks, the limited resources in some edge nodes cannot handle such computational tasks, which will lead to queuing of some tasks for execution at the edge nodes [6]. In contrast, some other computational resources may remain idle or non-busy, thus leading to an imbalance in the edge server load. Therefore, we must ensure load balancing at the edge nodes to improve the quality of user experience (QoE) and satisfy the quality of service (QoS) requirements. In addition, it is another major challenge to determine whether or where to offload tasks in multi-edge network architecture, limited by the computing ability and power of mobile terminals.

Scholars have investigated several approaches to overcome some of these challenges. Most of them jointly consider task latency and energy consumption in task offloading decisions [7]–[11]. However, only the time delay and energy consumption for given task completion is considered. The energy waste caused by some tasks being thrown after computation or offloading is still out of consideration because they cannot meet the latency requirements, which may lead to a waste of power supply and the reducing in USV working hours. For example, Thai T et al. [7] proposed a new edge computing network architecture that enables edge nodes to collaborate and share computational and wireless resources. Jie et al. [8] offloaded tasks to multiple edge servers by deep reinforcement learning algorithms to reduce energy consumption and average computational latency. However, these studies tend to consider collaboration between edges and ends, or end-to-end. Therefore, this study proposes a DRL-based system offloading algorithm in a cloud-edge collaboration scenario, TOLBO (Task Offloading with Cloud Edge Jointly Load Balance Optimization). It is featured by making offloading decision based on the task size of the USV, the queue information of the cloud, edge, and USVs, and the load level at the cloud and edge nodes. Therefore, it can improve the resource utilization of cloud and edge servers, especially the utilization of energy. In other words, when the computing tasks are intensive, the throw rate of tasks can be minimized. The main contributions are as follows:

- A task offloading problem model is developed for latency-sensitive and energy-sensitive tasks. The model considers the load level dynamics of the cloud and edge nodes to minimize the expected long-term cost of the task, especially the delay of the task, the penalty of the thrown task, and the energy consumption required to process the task.
- In order to achieve the desired long-term cost minimization, a DRL-based cloud edge collaborative joint optimization offloading algorithm named TOLBO is proposed.
- Simulation experiments show that TOLBO can better utilize the processing ability of the cloud-edge-end layers. Compared to other algorithms, it can significantly reduce the task drop rate, the average latency, and the energy consumption of mobile terminals.

The rest of this paper is organized as follows. Section II describes the current related work. Section III presents the system model. Section IV presents the design of the DRL-based offloading algorithm. Section V describes the experimental setup and the performance evaluation of the algorithm. Finally, We summarize the full paper in Section VI.

II. RELATED WORKS

The main goal of USV’s task offloading is to minimize task completion time and reduce the energy consumption of mobile terminals.

Completion time is generally defined as the time duration from submitting a request to receiving the task result, and it is often considered as a QoS metric in the design of offloading methods. The task completion time is affected by many factors such as waiting delay, transmission delay, and computation delay.

Due to the limited battery capacity of various mobile devices, the energy consumed by the devices to complete their tasks is also a matter to be considered. Energy consumption mainly includes data processing and data transmission energy consumption. For both types of energy consumption, all existing related works use simple linear models in which the energy consumed increases linearly with the amount of data processed or transmitted.

Since task offloading requirements are diverse, task offloading decision is usually a multi-objective optimization problem. A common solution to this problem is to transform multiple objectives into one objective by some methods (weighting method) or use one of the most important objectives as the optimization one and the others as constraints. Thus, the optimal task offload problem can be reduced to selecting the right task offloaded to the right edge server at the right time to improve the application’s performance while efficiently utilizing the edge server resources. This problem is an NP-hard multi-objective optimization problem [12], for which only an approximate optimal solution can be found.

In recent years, many scholars have researched the offloading decision problem to optimize latency and energy consumption. Thai T et al. [7] proposed a new edge computing network architecture that enables edge nodes to share computational and wireless resources collaboratively. They
describe the joint task offloading and resource allocation optimization problem as mixed-integer nonlinear programming (MINLP) and solve the problem using the interior point and branch-and-bound methods. It is demonstrated through experimental results that the proposed method minimizes the total energy consumption and satisfies all the delay requirements for mobile users. Du et al. [13] modeled the multi-objective optimization problem as a mixed-integer nonlinear programming problem containing two subproblems of task unloading decision and resource allocation, and solved the resource allocation subproblem using fractional programming and Lagrangian pairwise decomposition. However, they did not consider the waiting time delay for executing the task. Sun et al. [14] proposed a DBO-based online learning algorithm that jointly utilizes projected dual-gradient iteration and greedy methods to minimize the task computation communication delay at the user terminal. The long-term effectiveness of the algorithm was demonstrated experimentally. In large-scale offloading scenarios, traditional methods such as convex approximation [15], game theory [16], and metaheuristics [17] usually require complex mathematical modeling of the system. Thus, they have difficulty coping with the computational burden caused by the exponential growth of the search space, and traditional algorithms may find locally optimal rather than globally optimal offloading decisions.

In contrast, Reinforcement Learning (RL) can learn optimal offloading policies by interacting with edge networks, which can achieve model-free control without knowing the system’s internal transitions. Therefore, reinforcement learning methods are widely used in offloading [18]. Zhou et al. [19] proposed a Q-Learning-based spectrum access scheme in mobile networks to enable mobile users to access the optimal spectrum and maximize the transmission rate. Dab et al. [9] proposed a QL-JTAR algorithm to minimize the energy consumption of mobile terminals under application delay constraints. Traditional reinforcement learning stores tuples containing states, actions, and rewards into Q tables. However, they are not applicable when the state or action space is enormous. Deep Reinforcement Learning (DRL) can solve the above problem by integrating neural networks into reinforcement learning to approximate Q values. Therefore, DRL for joint optimization problems has attracted increasing interest from researchers, and complex offloading decision problems can be solved more efficiently using this approach. Jie et al. [8] offload tasks to multiple edge servers by deep reinforcement learning algorithms to reduce energy consumption and average computational latency. The offloading strategy is further optimized by transmission rate and battery power into the decision conditions during the learning process and improved the service performance of the whole edge system. Lu et al. [10] modeled a large-scale heterogeneous MEC environment and proposed an IDRQN algorithm based on the LSTM algorithm to address the shortcomings of DRL. Finally, the experiments verify its validity. Similarly, Meng et al. [11] also proposed an online DRL algorithm in their model to reduce the queuing delay of end-users by designing a new action space and reward function for making the trade-off between the energy consumption of mobile devices and the delay of tasks.

In order to better compare the working scenarios of the above mentioned working methods and optimize the dimensions, we have compared all methods accordingly in Table 1.

From the table, we can see that the current work usually only considers the collaboration among edge servers, but ignore cloud servers with a large number of computing resources. Also, all papers do not consider the waste of computational resources and energy when tasks are thrown. Therefore, in order to deal with large-scale edge computing scenarios, we propose the TOLBO algorithm based on DRL. This algorithm takes the task thrown into account and utilizes the computing resources of cloud edge servers to make the best decision.

### III. SYSTEM MODEL

In this section, we first give an overview of the system model and then describe the task model, delay model, energy model, and problem modeling successively.

As shown in Fig. 1, a quasi-static scenario will be considered in this paper. During task offloading, all mobile end devices and wireless network states do not change. At time slot, there are a series of USVs: \( U = \{u_1, u_2, u_3, \ldots, u_N\} \) connected wirelessly to the edge servers: \( M = \{m_1, m_2, m_3, \ldots, m_K\} \) and RSUs (Road Side Unit), where

#### TABLE 1. Comparison of existing methods.

| Method | Paper | Scenes | Target | Task Throw Penalty |
|--------|-------|--------|--------|--------------------|
| (traditional) | [7][12][14][16] | ✓ | ✓ | ✓ | ✓ |
| RL | [19] | ✓ | ✓ | ✓ | ✓ | ✓ |
| DRL | [8] | ✓ | ✓ | ✓ | ✓ |
| Ours | [10][11] | ✓ | ✓ | ✓ | ✓ |

![FIGURE 1. System architecture diagram.](image-url)
\[ d_u(t) = \begin{cases} 1, & 0, 0], \text{it means local execution, when} \ d_u(t) = [0, 1, 0], \text{it means execution on the edge server, and when} \ d_u(t) = [0, 0, 1], \text{it means offloading to the cloud server.} \]

When a task is offloaded to the edge server, we define a binary variable \( O_{m,u}(t) \in \{0, 1\} \) to indicate whether the task generated by the USV user \( u \) at time slot \( t \) will be offloaded to the edge server \( m \). When \( O_{m,u}(t) = 1 \), the task is offloaded to the edge server \( m \). Conversely, it is not offloaded. Because tasks are non-separable and a task is not repeatedly offloaded to multiple edge servers, \( O_{m,u}(t) \) is restricted by

\[
\sum_{m \in \mathcal{M}} O_{m,u}(t) = 1, \quad u \in \mathcal{U}, \quad t \in \mathcal{T}. \tag{1}
\]

### B. DELAY MODEL

1) COMPUTATION DELAY

When task \( \Omega_{i,t} \) is selected for local processing, it will be placed in the computation queue. At the beginning of the time slot \( t \), if the local computation queue is not empty and the USV user \( u \) at time slot \( t \), when at time slot 0, \( C_u(t) = 0 \), \( T_{\text{comp}}^{\text{cloud}}(t) \) denotes the maximum number of time slots that can be processed within the tolerated time delay at the current queue length, and \( \lceil D_u \rceil_{f_{\text{usv}}}^{\text{comp}} \) represents the number of time slots required for the task to be fully processed (rounded up).

We set the minimum resolution of the task execution time to 1-time slot by default, where \( f_{\text{usv}} \) indicates the processing capacity of USV. The symbol \( \lceil \cdot \rceil \) means rounding up, \( Q_u^{\text{comp}}(t) \) indicates the queue length (in time slots) of USV user \( u \) at time slot \( t \), when at time slot 0, \( Q_u^{\text{comp}}(t) = 0 \), \( T_{\text{comp}}^{\text{cloud}}(t) \) denotes the maximum number of time slots that can be processed within the tolerated time delay at the current queue length, and \( \lceil D_u \rceil_{f_{\text{usv}}}^{\text{comp}} \) represents the number of time slots required for the task to be fully processed (rounded up).

When the execution time \( T_u^{\text{comp}}(t) \) and queue length \( Q_u^{\text{comp}}(t) \) are known, we can calculate the queue length of the next time slot as follows:

\[
Q_u^{\text{comp}}(t+1) = Q_u^{\text{comp}}(t) + T_u^{\text{comp}}(t) - 1. \tag{3}
\]

For example, at time slot 0, the tolerated latency \( T_{\text{max}} \) of task \( \Omega_{u,0} \) is 10 time slots, and suppose that the task needs 3 time slots to be fully processed. At this time slot, the queue length \( Q_u^{\text{comp}}(0) = 0 \), and therefore \( T_u^{\text{comp}}(t) = \min\{10 - 0, 3\} = 3 \), then the queue length for the next time slot is \( Q_u^{\text{comp}}(1) = 0 + 3 - 1 = 2 \). Also, we can calculate the total latency \( T_{\text{local}}^{\text{comp}} \) for the task to be processed locally and completed or thrown.

\[
T_{\text{local}}^{\text{comp}} = Q_u^{\text{comp}}(t) + T_u^{\text{comp}}(t). \tag{4}
\]

When the task is executed on the edge server, the edge server receives the tasks offloaded by each USV user. To ensure the fairness of each task completion, we consider that the resources allocated to each USV user on the same

\[ N, K \] denote the number of USVs and the number of edge servers, respectively. Meanwhile, RSUs are wired to remote cloud servers to transfer task data and computation results.

The heterogeneous type of the edge server can be represented by

\[
Q_{\text{comp}} = \{(Q_{\text{comp}}^{\text{edge}}, f_{\text{comp}}^{\text{edge}}), \ldots, (Q_{\text{comp}}^{\text{cloud}}, f_{\text{comp}}^{\text{cloud}})\}
\]

where \( Q_{\text{comp}}^{\text{edge}} = \{Q_{\text{comp}}^{\text{edge}, 1}, \ldots, Q_{\text{comp}}^{\text{edge}, K}\} \) denotes the task queue information and \( f_{\text{comp}}^{\text{edge}} \) denotes the maximum CPU frequency. Similarly, the cloud server can be denoted as

\[
C = (Q_{\text{comp}}^{\text{cloud}}, f_{\text{comp}}^{\text{cloud}}).
\]

We consider that the tasks generated by USVs are indistinguishable and each USV generates at most one task in a time slot. So at the beginning of the time slot, when the task arrives, it can be selected to be processed locally, offloaded to an edge node or a cloud server. As shown in Fig. 2, if the USV decides to place the task locally for processing, the task is added to the queue waiting for computation (i.e., computation queue). Conversely, the task is placed into the queue waiting for transmission (i.e., transmission queue) depending on the offloading destination. If the task is offloaded to the cloud server, it is uploaded to the transmission queue of RSU first. As for all the transmission queues and computation queues, we consider them a FIFO queue, assuming that the current queue length, and \( \lceil D_u \rceil_{f_{\text{usv}}}^{\text{comp}} \) represents the number of time slots required for the task to be fully processed (rounded up).

When the execution time \( T_u^{\text{comp}}(t) \) and queue length \( Q_u^{\text{comp}}(t) \) are known, we can calculate the queue length of the next time slot as follows:

\[
Q_u^{\text{comp}}(t+1) = Q_u^{\text{comp}}(t) + T_u^{\text{comp}}(t) - 1. \tag{3}
\]

For example, at time slot 0, the tolerated latency \( T_{\text{max}} \) of task \( \Omega_{u,0} \) is 10 times lots, and suppose that the task needs 3 time slots to be fully processed. At this time slot, the queue length \( Q_u^{\text{comp}}(0) = 0 \), and therefore \( T_u^{\text{comp}}(t) = \min\{10 - 0, 3\} = 3 \), then the queue length for the next time slot is \( Q_u^{\text{comp}}(1) = 0 + 3 - 1 = 2 \). Also, we can calculate the total latency \( T_{\text{local}}^{\text{comp}} \) for the task to be processed locally and completed or thrown.

\[
T_{\text{local}}^{\text{comp}} = Q_u^{\text{comp}}(t) + T_u^{\text{comp}}(t). \tag{4}
\]

When the task is executed on the edge server, the edge server receives the tasks offloaded by each USV user. To ensure the fairness of each task completion, we consider that the resources allocated to each USV user on the same...
edge server are equal. Assume that at time slot \( t \), there are \( G_m(t) \) USV user’s tasks computed on the edge server \( m \). Therefore, the computational resources shared by each USV user is \( r^\text{edge}_m / G_m(t) \). It is worth noting that \( G_m(t) \) changes between time slots as tasks end or new tasks arrive. Hence, the computation queue information on the edge server also changes with time slots.

Here we assume that the tasks generated by USV users at time slot \( t \) are offloaded to the edge server at time slot \( t \). Thus, we can define \( \Omega_{u,m,t} = \Omega_{u,m,t} \).

Since the computational resources allocated by the edge server to different USV users change dynamically with the number of tasks on the edge server, it is difficult for us to derive the queue information on the edge server by a simple calculation. Therefore, we define \( q^\text{edge}_{u,m}(t) \) to indicate the queue length (in bits) of USV user \( u \) on edge server \( m \) at time slot \( t \). The update formula for \( q^\text{edge}_{u,m}(t) \) is shown below, where \( \epsilon^\text{edge}_{u,m}(t) \) represents the amount of data thrown by the task at the end of time slot \( t \).

\[
q^\text{edge}_{u,m}(t) = q^\text{edge}_{u,m}(t-1) + D_{u,m,t} - \frac{D_{u,t}}{f_m} / G_m(t) - \epsilon^\text{edge}_{u,m}(t).
\]

(5)

To better calculate the latency of tasks on the edge server, we define \( \bar{q}^\text{edge}_{u,m}(t) \in T \) to indicate that task \( \Omega_{u,m,t} \) is processed to completion or thrown at this time slot. Also, we define \( \epsilon^\text{edge}_{u,m}(t) \in T \) to denote that task \( \Omega_{u,m,t} \) starts to be processed at this time slot. Thus, we can conclude that

\[
\bar{q}^\text{edge}_{u,m}(t) = \max \{ t \in [0, \ldots, t-1], \bar{q}^\text{edge}_{u,m}(t') + 1 \}.
\]

(6)

The above equation indicates that task \( \Omega_{u,m,t} \) will start only after all tasks have been computed or thrown before time slot \( t \). Therefore, we can calculate \( \bar{q}^\text{edge}_{u,m}(t) \) from the following equation:

\[
\bar{q}^\text{edge}_{u,m}(t) = \sum_{t' = \bar{q}^\text{edge}_{u,m}(t)} f_m / G_m(t') \geq D_{u,m,t} - \sum_{t' = \bar{q}^\text{edge}_{u,m}(t)} f_m / G_m(t') < D_{u,m,t}.
\]

(7)

3) TASK DELAY

The above equation indicates that at time slot \( \bar{q}^\text{edge}_{u,m}(t) - 1 \), the computation size of the edge server is less than the task data size, while at time slot \( \bar{q}^\text{edge}_{u,m}(t) \), the computation size of the edge server is greater than or equal to the task data size, and the computation delay \( T^\text{edge}_{u,t} \) can be expressed as

\[
T^\text{edge}_{u,t} = \bar{q}^\text{edge}_{u,m}(t) - \bar{q}^\text{edge}_{u,m}(t).
\]

(9)

Meanwhile, the cloud server can think of as an edge server with more computing power, so its queue information and computational latency are similar to those of an edge server, which can be defined as \( q^\text{cloud}_{u,m}(t) \) and \( T^\text{cloud}_{u,t} \).

2) TRANSMISSION DELAY

The transmission model is also a FIFO queue. Assuming that the transmission power of the USV user is \( P^\text{tran}_u \), the maximum transmission rate in an additive Gaussian white noise (AWGN) channel [20] is

\[
r^\text{tran}_u = B \log_2 \left( 1 + \frac{P^\text{tran}_u |h|^2}{d^\beta N_0} \right), u \in U,
\]

(10)

where \( B \) denotes the bandwidth of the channel, \( d \) denotes the distance between the USV user and the RSU, \( N_0 \) is the noise power. Also, the above transmission model uses Rayleigh fading channel model. \( h \) is the channel attenuation coefficient and \( \beta \) is the path attenuation coefficient.

Similar to the computation queue, at the beginning of the time slot, if the transmission queue is not empty and the USV user \( u \) does not execute the transmission task, the task at the top of the queue is added to the transmission process. Also, we set the minimum resolution of the task transmission time to 1-time slot by default, then the actual execution time of the task from the USV user \( u \) to the edge server \( m \) is

\[
T^\text{tran}_u(t) = \min \left\{ T_{u,t}^{\text{max}} - Q^\text{tran}_u(t), \left[ \frac{D_{u,t}}{r^\text{tran}_u} \right] \right\}.
\]

(11)

Therefore, we can derive the transmission queue \( Q^\text{tran}_u(t+1) \) for the next time slot \( t \) as follows:

\[
Q^\text{tran}_u(t+1) = Q^\text{tran}_u(t) + T^\text{tran}_u(t) - 1.
\]

(12)

Another part of the tasks is offloaded to the cloud server for execution, which is first offloaded to the roadside unit (RSU) and then transmitted to the cloud server via the RSU. Since RSUs and cloud servers are typically connected by wired connections such as fiber optics, the data size of tasks can be neglected for the transmission rate, and only the overhead between transmissions needs to be considered. We define this overhead delay as a constant \( t_{\text{RSU}}^\text{tran} \), and the actual execution time of the RSU offloading to the cloud server can be defined as:

\[
T^\text{tran}_u(t) = \min \left\{ T_{u,t}^{\text{max}} - Q_{\text{RSU}}^\text{tran} - T^\text{tran}_u(t), t_{\text{RSU}}^\text{tran} \right\}
\]

(13)

The transmission queue for the next time slot can be defined as

\[
Q^\text{tran}_u(t+1) = Q^\text{tran}_u(t) + T^\text{tran}_u(t) - 1.
\]

(14)

3) TASK DELAY

Depending on the offloading decision of the task, the total delay of task completion or being thrown by the USV user at time slot \( T \) is defined by \( T_{u,t} \), which is expressed as:

\[
T_{u,t} = d_{u,t} \left[ T^\text{local}_{u,t} + T^\text{edge}_{u,t} + T^\text{cloud}_{u,t} \right].
\]

(15)
C. ENERGY MODEL

Since USVs need to execute tasks in the outdoor water environment, they carry limited battery capacity. While edge servers and cloud servers are powered by fixed power supplies, the energy consumption of servers is negligible. Therefore, in this paper, our primary goal is to reduce the energy consumption of USVs for processing tasks.

The energy consumption of USV processing tasks consists of two main parts. When the tasks are executed locally, we mainly consider transmission energy consumption.

Computation energy consumption: In subsection III.B.1 we mentioned that the computational capability of the USV user is $f_u^{av}$, and for the task $\Omega_{u,t}$ the actual execution time is $T_u^{comp}(t)$. It is worth noting that it does not necessarily mean that the task will be completed, and the task may be thrown. Thus, the computational energy consumption incurred by processing the task is similar to what was defined in [21].

$$E_{u,t}^{comp} = \begin{cases} c \left( f_u^{av} \right)^2 \cdot D_{u,t} \cdot \frac{\alpha_{u,t}}{c \left( f_u^{av} \right)^2} \cdot T_u^{comp}(t) & \text{complete} \\ c \left( f_u^{av} \right)^2 \cdot D_{u,t} \cdot T_u^{comp}(t) & \text{drop} \end{cases}$$

(16)

Transmission energy: If a task is decided to be offloaded to an edge server or a cloud server, then the energy consumption mainly comes from the transmission energy. Recall that in the previous subsection, we defined the transmission capability of the USV user as $P_{usv}^{trans}$, then the energy consumption required for transmitting the task $\Omega_{u,t}$ is [22].

$$E_{u,t}^{trans} = \begin{cases} p_{usv}^{trans} \cdot D_{u,t} \cdot T_{u,t}^{trans} & \text{complete} \\ p_{usv}^{trans} \cdot T_{u,t}^{trans} & \text{drop} \end{cases}$$

(17)

Therefore, depending on the offloading decision of the task, the total energy consumption of the USV user generated by the task completed or thrown can be defined as:

$$E_u = d_{u,t} \cdot \left[ E_{u,t}^{comp} \quad E_{u,t}^{trans} \quad E_{u,t}^{drop} \right]^T$$

(18)

D. PROBLEM MODELING

The focus of this work is to optimize the long-term performance of the USV, i.e., to maximize the number of completed tasks and reduce the completion delay of each task within the tolerable delay range while reducing the energy consumption of the USV. We can therefore model this optimization problem as follows:

$$P = \min \left( \sum_{u \in U} \sum_{t=0}^{\infty} \alpha_{u,t} + \beta E_{u,t} \right)$$

(19)

IV. DRL-BASED OFFLOADING ALGORITHM DESIGN

A. MDP MODEL

At the beginning of each time slot, each USV observes its state (e.g., task size, queue information) and transmits it to the decision agent. If a new task is to be processed, the agent selects an action for that task. The current state of the USV and the task decision action incur corresponding costs (i.e., the delay of the task, the penalty when the task is not completed, and the energy consumption required to process the task). The agent’s goal is to minimize their expected long-term costs by optimizing the policy from state to action.

1) State Space

At the beginning of the time slot $t \in T$, each USV $u \in U$ observes its state information, including task size, queue information, and load level history of the edge nodes. Specifically, the USV user $u$ observes the following states:

$$S_u(t) = (D_{u,t}, Q^m_{comp}(t), Q^m_{trans}(t), Q_{RUE}(t), q^m_{edge}(t-1), q^m_{cloud}(t-1))$$

(20)

2) Action Space

At the beginning of the time slot $t \in T$, if a USV user $u \in U$ has a new task $\Omega_{u,t}$ arriving, then it will choose the offloading action for the task: (a) whether to process the task locally or offload it to an edge node or a cloud server, i.e., $d_{u,t} = 1$; (b) to which edge node the task is offloaded, i.e., $a_{u,t}$. Thus, the action of USV user $u$ in time slot $t$ is represented by the following action vector

$$A_u(t) = (d_{u,t}, a_{u,t})$$

(21)

3) Reward

If the task is processed successfully, then the task latency is the duration between the time the task arrives and the time the task is processed. If the task is thrown, we set the task latency as a penalty delay ($\gamma T_{max}$). Thus, the final task latency can be expressed as

$$Delay_{u,t} = \begin{cases} T_{u,t} \cdot \frac{\alpha_{u,t}}{\gamma T_{max}} & \text{complete} \\ \gamma T_{max} & \text{drop} \end{cases}$$

(22)

We design a flexible reward function that solves all optimization objectives at once using a weighted sum of energy consumption and delay. The reward function $R_t (S_t, A_t)$ can be expressed as

$$R_t (S_t, A_t) = -(1 - \eta)\beta_1 Delay_t - \eta \beta_2 log_2 (E_t)$$

(23)

where $\beta_1$ and $\beta_2$ are the parameters that normalize the task delay and task energy consumption. $\eta \in [0, 1]$ is used to smooth the weighting factor of task delay and energy consumption. The task energy consumption is approximated as a squared multiple of the device operating frequency, so we scale it with a logarithmic function.

The agent can only obtain the immediate reward in the current state, not the future reward, so in order to maximize the long-term reward, we go through the learning strategy to predict the expected future reward. Here we introduce the Q function, which is defined as the expected value of the long-term reward obtained by taking specific action in a certain state. The Bellman equation of Q function can be expressed as

$$Q(s, a) = \mathbb{E} [R_t + \gamma G_{t+1} | s_t = s, a_t = a]$$

(24)
where $R_t$ is the immediate reward, $\gamma$ is the discount factor, since our assessment of the future is not always accurate, a discount needs to be added to the expected future reward to express future uncertainty: $\gamma G_{t+1}$ is the long-term reward after the discount, which can be calculated by

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}. \quad (25)$$

The expected future reward is usually predicted by the strategy $\pi$, so we need to find an optimal strategy $\pi^*$ to maximize the Q function by

$$Q^*(s, a) = \max_{\pi} E \left[ R_t + \gamma G_{t+1} \mid s_t = s, a_t = a, \pi \right]. \quad (26)$$

### B. DRL MODEL

In this subsection, we propose a DRL model to solve the joint optimization problem presented in the previous section.

Due to the complexity and continuity of states, it is impractical to observe all states and actions. Deep Reinforcement Learning (DRL) introduces deep neural networks to replace Q-functions (Deep Q-Network DQN) as approximators so that the action of the optimal strategy $\pi^*$ can be expressed as

$$a = \arg \max_a Q \left( s; a', w \right). \quad (27)$$

Unlike traditional supervised learning, DRL models can learn without a priori knowledge by interacting with the environment and continuously exploring feedback. However, the environmental feedback changes with each interaction. To prevent oscillations in model training, DQN uses the target Action-Value network to back up the weight parameters of the Action-Value network, making the fitted labels relatively stable over time and thus allowing smoother convergence of the Action-Value network. Thus, the target Action-Value network and the Action-Value network are structurally identical, and their estimates of both reward and action values are related to the weight parameters. During the training process, the Action-Value network learns from the reward estimates $\tilde{F}$ given by the target Action-Value network and updates the weights of the Action-Value network through the loss function $L_t (w_t)$. At this time, the weights of the target Action-Value network are not updated. However, the weight parameters of the Action-Value network are assigned to the target network after the Action-Value network has completed a certain number of updates. The reward valuation of the target Action-Value network is calculated as

$$\tilde{F} = r + \gamma \max_a Q^* \left( s'; a'; w^-_t \right), \quad (28)$$

where $w^-_t$ denotes the weight parameter of the Action-Value network in the previous round and the loss function can be expressed as

$$L_t (w_t) = E_{s,a,r} \left[ (E_{\tilde{a}}[\tilde{F} \mid s, a] - Q (s, a; w_t))^2 \right]. \quad (29)$$

### C. TRAINING PROCESS

In this subsection, we describe the training process of the DRL model. To simplify the description, we divide the training process into three parts: initialization, training data generation, and learning from the experience pool, as shown in Fig. 3.

1) Initialization: Initialize an Action-Value network $Q(s, a \mid \theta)$ and assign the parameter $\theta$ to the Target Action-Value network $Q(s, a \mid \bar{\theta})$. A replay memory $D$ to capacity $N$ is also initialized to store the experience tuples $(s, a, r, s')$ generated by USV users interacting with the environment. Then, the whole system iteratively generates data and trains the DRL model over a while. This Episode ends when the step is greater than or equal to the designed time slot.

2) Exploration and data acquisition: USV users interact with the network environment to generate training datasets. The agent balances exploitation and exploration by using the $\epsilon$-greedy method. Initially, the agent knows nothing about the edge network environment, and it takes more random actions to explore the environment. As the agent gradually gains enough knowledge of the environment, the agent uses the learned knowledge to generate optimal policies. As a result, $\epsilon$ decreases after several Episode cycles. The reward $r_t$ and the next state $s_{t+1}$ can be obtained by interacting with the environment, and the tuple composed of the current state $s_t$, action $a_t$, reward $r_t$ and next state $s_{t+1}$ is stored in the experience pool to improve the learning speed of DRL.

3) Replay memory: When the agent interacts with the network environment, the sequences of experience tuples may be highly correlated and lead to training failure if the interaction data are sequentially input into the learning model. Therefore, we use the experience replay method to sample training data from the experience pool randomly. By this approach, data generation and learning are not dependent on each other, decoupling the sequential correlation of experience tuples. Second, replay memory enables the agent to learn more from the same experience. Most importantly, replay memory can reduce oscillations or divergences from discrete training samples by using batch samples.

4) Learning: The DRL learning process is slightly different from training a regular deep learning model. In forward propagation, the DRL model takes a batch of training samples from the experience pool and feeds them to the learning network and the target network; the error between the returns from the learning network and the target network is then used to calculate the loss. The loss function of DRL is calculated by the difference between the outputs of the learning network and the target network because the DRL model learns experience from the reward feedback rather than from the labeled data learning experience.
The pseudo code is shown in Algorithm 1.

**Algorithm 1 DRL-Based Task Offloading With Cloud Edge Jointly Load Balance Optimization Algorithm (TOLBO)**

```plaintext
// 1. Initialize:
1. Initial replay memory D to capacity N
2. Initial action-value network with random weight θ
3. Initial target action-value network with weights θ− = θ
4. for Episode 1 to M do
   // 2. Generate training data
   5. Initial input raw data x
   6. for task in raw data do
      7. Select action a from state s using: π ← ε - Greedy(ˆQ(s, a, θ))
   end
   8. Take action A, observe reward R and get next input S′
   9. Store experience tuple (S, A, R, S′) in replay memory D
10. // 3. Learning:
    11. Obtain mini-batch of (sj, aj, rj, sj+1) from D
    12. if Episode stop at step j + 1 then
       13. Set target ˆFj ← rj
    else
       14. Set target ˆFj ← rj + γ maxa′ Q(sj, a, w−)
    end
   15. Update θ ← θ + α∇wL (wj)
16. Every N steps, update: θ− = θ
   17. ε ← max (εmax, ε * dec av)
18. Store score for current episode
```

**V. EXPERIMENTAL ANALYSIS**

In this section, we evaluate the performance of TOLBO in simulations. We use Python as the programming language in this simulation. In addition, we choose PaddlePaddle [23] to build the DRL model. We consider a static scenario with 50 USV users, 5 edge servers, and 1 cloud server for the simulation experiment. That is, the number of users does not change over time. If the environment changes, we can also reset the random exploration probability to 1 and retrain the model. The parameter settings for this experiment are shown in Table 1.

| Parameter                        | Quantity      |
|----------------------------------|---------------|
| SNR(DB)                          | 100           |
| Task size (bits)                 | (2.0, 2.1, ..., 5.0) Mbits |
| USV calculation frequency        | 2.5 GHz       |
| Edge server calculation frequency| 41.8 Ghz      |
| Cloud Server Computation Frequency| 200 Ghz |
| Number of USV users              | [10, 100]     |
| Number of edge servers           | [2.5]         |
| Task Generation Rate             | [0.1]         |
| Batch size                       | 8             |
| Learning Rate                    | 0.001         |
| Discount Factor                  | 0.99          |
decreases and does not drop to the lowest convergence cost after 500 Episode cycles. Moreover, when the learning rate is large ($lr = 10^{-2}$ or $lr = 10^{-1}$), the parameter variation of the neural network becomes larger, so the anomalous data disturbance in the experiment will put the algorithm in an unstable state. It leads to a higher convergence cost than the random unloading strategy. Therefore, this experiment chooses $lr = 10^{-3}$ to reduce the convergence cost and accelerate the convergence speed.

Fig. 4(b) shows the convergence of the proposed algorithm for different batch sizes, which is the number of samples selected in each training round. When the batch size increases from 2 to 8, the convergence speed is accelerated. Furthermore, when it is increased from 8 to 32, the algorithm’s convergence speed and convergence cost are not significantly improved. Therefore, this experiment chooses batch size 8 to reduce the training time while ensuring the algorithm’s performance.

B. PERFORMANCE EVALUATION

We compare the proposed algorithm (TOLBO) with several existing methods, including no offload (N. Offload), random offload (R. Offload), DRL [11], and we also compare our offloading algorithm with edge servers only (TOLBO*).

1) PERFORMANCE INDICATORS

In this experiment we consider the following performance metrics:

1) Task throw rate: The task throw rate refers to the percentage of the total number of tasks thrown because they are not completed within the tolerated delay range and is measured by $\theta_{\text{throw}}$. A smaller $\theta_{\text{throw}}$ means a smaller number of thrown tasks and a more stable system. $\theta_{\text{throw}}$ can be expressed by

$$\theta_{\text{throw}} = \frac{C_{\text{throw}}}{C_{\text{task}}},$$

where $C_{\text{throw}}$ denotes the number of tasks thrown by all USV users in an Episode, and $C_{\text{task}}$ denotes the total number of tasks generated by all USV users in an Episode.

2) Average task delay: the average delay of USV users to complete each task, denoted by

$$T_{\text{avg}} = \frac{\sum_{t \in T} \sum_{m \in M} T_{t,m}}{C_{\text{task}}},$$

3) Average energy: Because the energy consumed by a task that is thrown is small, which can be easily averaged over the overall number of tasks, we calculate the average energy consumption using the average of the completed tasks relative to the energy consumption generated by the USV over the entire Episode, denoted by

$$E_{\text{avg}} = \frac{\sum_{t \in T} \sum_{m \in M} E_{t,m}}{C_{\text{task}} - C_{\text{throw}}},$$

4) Energy waste rate: This parameter represents the percentage of energy wasted by tasks that are thrown during execution to the overall USV energy consumption, measured by $\theta_{E}$. A smaller $\theta_{E}$ means a higher energy utilization by USV users $u$. $\theta_{E}$ can be expressed by

$$\theta_{E} = \frac{E_{\text{throw}}}{E_{\text{all}}},$$

2) IMPACT OF TASK ARRIVAL RATE ON PERFORMANCE

As task arrival rates increase, more tasks are generated for the same number of USV users in the same time slot. Incorrect offload decisions can lead to task backlogs in the queue, resulting in task throwing, unnecessary transmission, and computational energy consumption.

In Fig. 5(a), the number of tasks generated by each USV user in the same Episode increases as the task arrival rate increases. The proposed algorithm in this paper can maintain a lower task throw rate than the benchmark method. When the task arrival rate increases from 0.1 to 0.4, the average delay of our proposed DRL-based algorithm increases by 17.2%, while the average delay of the benchmark method increases by at least 92.5%. This implies that the average delay of the proposed algorithm increases less than that of the benchmark method as the system load increases. As the task arrival probability increases to 0.7%, the average latency of
some methods decreases as more and more tasks are thrown, bringing the average latency slowly closer to the maximum tolerated latency. Also, compared to TOLBO*, the average latency of TOLBO is reduced by 15.3% due to the support of cloud servers with higher computational power.

In Fig. 5(b), TOLBO is able to consistently maintain a lower throw rate compared to the benchmark algorithm. For example, when the task arrival probability is at 0.5, the throw rate of TOLBO is 7.28%, while the throw rate of the benchmark algorithm is between 62.13% and 83.16%. This is because TOLBO makes better use of all computing resources at the cloud edge and end side. At higher task arrival rates (0.7-1.0), the throw rate of TOLBO is reduced by at least 39.8% compared to TOLBO*. Because the computing resources at the edge server and user side are already severely insufficient when the computational pressure is high, the addition of cloud servers can effectively alleviate the situation.

In Fig. 5(c) and Fig. 5(d), the average energy consumption of our proposed algorithm is reduced by 47.55% compared to the average energy consumption of the benchmark method when the task arrival rate is 0.1. For USVs, power resources are pretty valuable. When the workload is low, the algorithm balances power consumption and delay, and offloads more tasks to the cloud edge servers within the range of time delay tolerance, so as to reduce the energy consumption of processing tasks for USVs. At the same time, when the task arrival rate is high, the tasks are thrown away excessively due to the wrong offloading decisions of other algorithms, which increases the average energy consumption of tasks and the waste rate of energy consumption.

3) IMPACT OF THE NUMBER OF USERS ON PERFORMANCE

As the number of users increases, more tasks are generated in the same time slot. Different from the task arrival rate, the more users there are, the more local computing resources there are.

In Fig. 6, we discuss the comparison of the performance of each method for the different numbers of USV users. As the number of users increases, the decision dimension increases. This reduces the edge cloud node resources allocated to each USV user, which puts forward higher requirements for the adaptability of the algorithm.

As shown in Fig. 6(a), the proposed algorithm in this paper makes good use of the computational resources of the edge cloud nodes at the different number of users so that the latency is always kept at a steady low level. TOLBO* can still maintain a low latency at 60 users, but when the number of users rises to 70-100, the task needs to be further offloaded to the cloud server due to the insufficient computing power of the edge nodes, which leads to a rapid increase in latency.

In Fig. 6(b), as the number of USV users increases, the dimensionality of the offload decision increases, thus, the wrong offload decision leads to an increasing throw rate of the benchmark method. In contrast, TOLBO always maintains a zero loss rate. Compared to TOLBO, TOLBO* has a progressively higher task drop rate when the number of users is greater than 50 due to the lack of computing resources from the cloud server.

In Fig. 6(c) and Fig. 6(d), when the number of users is small (10-50), the proposed algorithm in this paper reduces the average energy consumption by at least 31.57% and the energy waste rate by 15.44% compared to the benchmark method. When the number of USV users is high (60-100), the algorithm of this paper greatly reduces the average energy consumption and energy waste rate. For example, when the number of users is 100, the average energy consumption is reduced by 49 compared with R. Offload and DRL, and the energy waste rate is reduced by 21.35%. At the same time, TOLBO* maintains a low energy consumption and energy waste rate when the number of users is low. However, as the number of users increases, some tasks are thrown away due to the lack of computing resources in the cloud server, thus increasing the average energy consumption and energy waste rate.

VI. CONCLUSION

In this work, we consider the energy consumption overhead of the thrown tasks and study the computational offloading problem of the USV with the objectives of minimizing the task throw rate and task delay and minimizing the task energy consumption. We propose TOLBO, a DRL-based cloud edge collaborative load optimization task offloading algorithm. The proposed algorithm maximizes the long-term task pay-off. Moreover, the algorithm can make all decisions without relying on other optimization functions for joint optimization purposes. Simulation results demonstrate that the algorithm can significantly reduce the task throw rate, average latency, and the average energy consumption of the processing tasks, compared to the methods of No Offload (N.Offload), Random Offload (R.Offload), and DRL [11]. However, the experimental results of the method described in this paper are still
derived based on quasi-static simulation experiments. In the future, we will consider migrating the method to dynamic real experimental scenarios and combining the latest algorithms and techniques, such as using the LSTM algorithm to predict upcoming tasks, so as to improve the performance of the TOLBO algorithm in dynamic IoT application scenarios.

REFERENCES

[1] O. Levander, “Autonomous ships on the high seas,” IEEE Spectr., vol. 54, no. 2, pp. 26–31, Feb. 2017.
[2] W. Naeem, R. Sutton, and J. Chadley, “Modelling and control of an unmanned surface vehicle for environmental monitoring,” in Proc. UKACC Int. Control Conf., 2006, pp. 1–6.
[3] J. Lee, J.-W. Kim, and J. Lee, “Mobile personal multi-access edge computing architecture composed of individual user devices,” Appl. Sci., vol. 10, no. 13, p. 4643, Jul. 2020.
[4] A. Asheralieva and D. Niyato, “Bayesian reinforcement learning and Bayesian deep learning for blockchains with mobile edge computing,” IEEE Trans. Cogn. Commun. Netw., vol. 7, no. 1, pp. 319–335, Mar. 2021.
[5] S. Ranadheera, S. Maghsudi, and E. Hossain, “Computation offloading and activation of mobile edge computing servers: A minority game,” IEEE Wireless Commun. Lett., vol. 7, no. 5, pp. 688–691, Oct. 2018.
[6] C. Long, Y. Cao, T. Jiang, and Q. Zhang, “Edge computing framework for cooperative video processing in multimedia IoT systems,” IEEE Trans. Multimedia, vol. 20, no. 5, pp. 1126–1139, May 2018.
[7] T. T. Vu, N. V. Huynh, D. T. Hoang, D. N. Nguyen, and E. Dutkiewicz, “Offloading energy efficiency with delay constraint for cooperative mobile edge computing networks,” in Proc. IEEE Global Commun. Conf. (GLOBECOM), Dec. 2018, pp. 1–6.
[8] J. Xu, L. Chen, and S. Ren, “Online learning for offloading and autoscaling in energy harvesting mobile edge computing,” IEEE Trans. Cogn. Netw., vol. 3, no. 3, pp. 361–373, Sep. 2017.
[9] B. Dab, N. Aitsaadi, and R. Langar, “Q-learning algorithm for joint computation offloading and resource allocation in edge cloud,” in Proc. IFIP/IEEE Symp. Integ. Netw. Service Manage. (IM), May 2019, pp. 45–52.
[10] H. Lu, C. Gu, F. Luo, W. Ding, S. Zheng, and Y. Shen, “Optimization of task offloading strategy for mobile edge computing based on multi-agent deep reinforcement learning,” IEEE Access, vol. 8, pp. 202573–202584, 2020.
[11] H. Meng, D. Chao, and Q. Guo, “Deep reinforcement learning based task offloading algorithm for mobile-edge computing systems,” in Proc. 4th Int. Conf. Math. Artif. Intell. (ICMAI), 2019, pp. 90–94.
[12] Q.-V. Pham, L. B. Le, S.-H. Chung, and W.-J. Hwang, “Mobile edge computing with wireless backhaul: Joint task offloading and resource allocation,” IEEE Access, vol. 7, pp. 16444–16459, 2019.
[13] J. Du, L. Zhao, J. Feng, and X. Chu, “Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee,” IEEE Trans. Commun., vol. 66, no. 4, pp. 1594–1608, Apr. 2017.
[14] Z. Sun and M. R. Nakhai, “An online learning algorithm for distributed task offloading in multi-access edge computing,” IEEE Trans. Signal Process., vol. 68, pp. 3090–3102, 2020.
[15] B. Si and Y. Zhang, “Computation rate maximization for wireless powered mobile-edge computing with binary computation offloading,” IEEE Trans. Wireless Commun., vol. 17, no. 6, pp. 4177–4190, Apr. 2018.
[16] J. Zheng, Y. Cai, Y. Wu, and X. Shen, “Dynamic computation offloading for mobile cloud computing: A stochastic game-theoretic approach,” IEEE Trans. Mobile Comput., vol. 18, no. 4, pp. 771–786, Apr. 2018.
[17] F. Guo, H. Zhang, H. Ji, X. Li, and V. C. M. Leung, “An efficient computation offloading management scheme in the densely deployed small cell networks with mobile edge computing,” IEEE/ACM Trans. Netw., vol. 26, no. 6, pp. 2651–2666, Dec. 2018.
[18] S. K. Dash, S. Dash, J. Mishra, and S. Mishra, “Opportunistic mobile data offloading using machine learning approach,” Wireless Pers. Commun., vol. 110, no. 1, pp. 125–139, Jan. 2020.
[19] Z. Su, M. Dai, and Q. Xu, “Q-learning-based spectrum access for content delivery in mobile networks,” IEEE Trans. Cogn. Commun. Netw., vol. 6, no. 1, pp. 35–47, Mar. 2020.
[20] Z. Wang and Q. Zhu, “Partial task offloading strategy based on deep reinforcement learning,” in Proc. IEEE 6th Int. Conf. Comput. Commun. (ICCC), Dec. 2020, pp. 1516–1521.
[21] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, “A survey on mobile edge computing: The communication perspective,” IEEE Commun. Surveys Tuts., vol. 19, no. 4, pp. 2323–2358, 4th Quart., 2017.
[22] R. J. C. Bultitude, “Measurement, characterization and modeling of indoor 800/900 MHz radio channels for digital communications,” IEEE Commun. Mag., vol. COM-25, no. 6, pp. 5–12, Jun. 1987.
[23] Y. Ma, D. Yu, T. Wu, and H. Wang, “PaddlePaddle: An open-source deep learning platform from industrial practice,” Frontiers Data Comput., vol. 1, no. 1, pp. 105–115, 2019.