Cold-Start-Aware Offloading and Resource Allocation by Importance Sampling-Based Double Dueling DQN in Serverless Edge Computing

Peihao Wu, Haiming Chen, *Member, IEEE*, Tianhao Wu, Kaiqi Gu, and Yinshui Xia, *Member, IEEE*

***Abstract*—Serverless edge computing (SEC) seamlessly integrates edge computing with serverless computing, not only overcoming the limitations of resource-constrained edge nodes but also alleviating the high latency associated with cloud response. Due to the elastic scalability of serverless computing platforms, the cold start of latency-sensitive serverless functions (SFs) has become a significant challenge. Traditional strategies, such as resource reservation and prewarming, often suffer from low resource utilization. Meanwhile, offloading-based approaches simplify the problem by assuming a fixed high cold start delay cost, which is unsuitable for heterogeneous SEC scenarios. This article proposes a cold-start-aware offloading by doubledueling-DQN (CSODQN) model for SFs in a cloud–edge–device serverless computing system. The model creates an instance warming pool for SFs to enable reuse and allocates edge service node resources based on the priority of user and SFs, achieving multiobjective offloading optimization that considers cold starts. Our goal is to balance the frequency of cold start and resource utilization. To address the partially observable offloading optimization problem among agents, we employ a multiagent deep reinforcement learning approach. By introducing an priority of action-based sampling strategy, we accelerate the convergence of learning for each agent. Simulation results demonstrate that our method improves task success rates, reduces average task latency and cold start occurrences, and enhances resource utilization. Our approach alleviates the frequency of cold starts without excessively consuming system resources and costs, achieving long-term optimization of service quality, device energy consumption, and expenses.**

***Index Terms*—Cold start, deep reinforcement learning (DRL), function offloading, prioritized action sampling, serverless edge computing (SEC).**

I. Introduction

With the deployment model of traditional Infrastructure as a Service (IaaS) evolving toward Function Computing or Function as a Service (FaaS) [1], serverless computing breaks down complex applications into a set of loosely coupled concise functions [serverless functions (SFs)]. This allows for agile and flexible development, management, and scaling of SFs, enabling elastic resource allocation based on user demand. It allocates fine-grained computing resources for function requests, such as the creation of container and WebAssembly runtime environments. Serverless computing has become the de facto standard for the next generation of cloud computing [2]. However, when a system receives a SF request, the required computational resources may be uninitialized or in a sleep state, resulting in additional time for resource allocation and environment initialization. The process is known as the cold start [3], as reported in AWS Lambda, Google Cloud Functions, and Azure Functions, the cold start delays typically range from hundreds of milliseconds to several seconds [4]. However, the execution delay of these simple SF services (i.e., Web services, machine learning inference, and IoT data processing) are typically less than 1 s, which contrasts sharply with their cold start latency.

The stateless execution characteristic of serverless computing refers to the recycling and destruction of the runtime environment created for a SFs request after returning the computation results. To mitigate the high overhead of cold start latency, the runtime environment is not immediately deleted. Typically, runtime environments such as containers remain active for 5 to 15 min after computation ends [5], allowing subsequent function requests to reuse warm function instances and execute with warm starts. Serverless edge computing (SEC) overcomes the practical limitations of the cloud, reducing network overhead and latency, enabling real-time services, and improving Quality of Service (QoS) [6]. For instance, AWS IoT Greengrass, Azure IoT Edge, and Google Cloud IoT Edge adopt cross-edge and cloud paradigms to develop and deploy IoT SFs. In SEC environments with low bandwidth and high communication latency, the cold start problem severely hinders system QoS [7]. It not only increases the total execution time of latency-sensitive SF services but also reduces task success rates. Existing serverless computing providers primarily adopt two approaches to address the cold start problem: 1) reducing cold start latency through lightweight techniques and 2) avoiding cold start occurrences by predictive methods. For example, WebAssembly, which is closer to machine code, is used to compress code package images to reduce cold start latency [8]. Predictive methods, such as prewarming and reuse, are employed to avoid cold starts [9], [10]. Vahidinia et al. [11] proposed a joint optimization approach for prewarming and reuse, determining the types of prewarmed containers needed

Received 11 May 2025; revised 26 June 2025; accepted 9 July 2025. Date of publication 14 July 2025; date of current version 25 September 2025. This work was supported in part by the Natural Science Foundation of Zhejiang Province under Grant LY24F020001; in part by the Ningbo 2035 Key Technology Breakthrough Program under Grant 2024Z145 and Grant 2025Z032; and in part by the Major S&T Projects of Ningbo High-Tech Zone under Grant 2022BCX05001. *(Corresponding author: Haiming Chen.)*

The authors are with the Faculty of Electrical Engineering and Computer Science, Ningbo University, Ningbo 315211, Zhejiang, China (e-mail: chenhaiming@nbu.edu.cn).

Digital Object Identifier 10.1109/JIOT.2025.3588727

and the optimal time to keep containers warm. Whether through lightweight techniques that compress the cold start process or predictive methods like prewarming and reuse, neither approach alters the original SF request arrival patterns at serverless computing nodes. These methods build upon the existing SF request arrival patterns to make subsequent predictions and optimize runtime environments. The effectiveness of prewarming or caching reuse depends on the accuracy of these predictions. Existing task offloading strategies alter the arrival patterns of SFs requests to achieve multiobjective optimization of latency, energy consumption, and cost, thereby improving system QoS while enhancing resource utilization. For example, an optimal joint offloading scheme based on resource occupancy prediction [12] models the task offloading decision process as a Markov decision process (MDP) and employs reinforcement learning methods to optimize task offloading decisions [13], [14]. These approaches aim to strike a balance between service latency, task success rates, and resource cost efficiency. However, they lack consideration of the cold start problem.

For instance, Tang et al. [15] proposed a multiagent task offloading algorithm based on Dueling Double Deep Recurrent Networks to improve system QoS and resource utilization. To alleviate the cold start problem, Zhao et al. [2] and Chen et al. [16] modeled it as a container switching cost problem, exploring task offloading strategies to optimize cold start effectiveness. These methods introduce fixed cold start delay costs to optimize task offloading strategies, which are suitable for scenarios with relatively fixed function request types and predictable request patterns. However, they are less applicable in edge computing scenarios characterized by diverse and dynamic SFs requests. Existing methods for alleviating the cold start problem improve system service quality but often neglect considerations of system resource utilization. Meanwhile, task offloading strategies typically model cold start latency as a high delay cost, optimizing average latency and system resource utilization. These strategies are suited for horizontal offloading modes but overlook the heterogeneity of edge service nodes and their differences in cold start latency. In the cloud–edge–end collaborative serverless computing architecture, we systematically study the task offloading problem and the cold start issue of SEC. We introduce a vertical offloading mechanism and a warm instance pool for SFs, creating different cold start execution models for various service nodes. A cold-start-aware distributed function offloading method based on deep reinforcement learning (DRL) is proposed, using individual SFs request tasks as the basic unit of SEC task offloading. Each user's function request executes an offloading strategy, effectively addressing the challenges of traditional reinforcement learning in handling high-dimensional state and action spaces [17]. Moreover, a priority-based action sampling strategy is introduced to accelerate the learning convergence of agents [18].

Our method aims to enhance task success rates and system service quality while reducing task cold start probabilities, execution delays, energy consumption, and cost. Our main contributions are as follows.

1) We investigated the offloading method for SFs in a SEC system. To reduce the frequency of cold starts and efficiently utilize resources in the SEC distributed computing environment, we developed a practical execution model based on a warm instance pool for SFs and established a multilayer SEC system model comprising cloud, edge, and end devices.
2) The multiuser, multitype function request offloading problem is transformed into a multiobjective joint optimization problem involving cold start frequency, task execution delay, task energy consumption, and task cost. To improve task success rates under constraints of maximum task delay and resource energy consumption, we proposed an action-priority-sampling Double Dueling DQN algorithm. This algorithm identifies the optimal task offloading strategy that minimizes cold start frequency, maximizes task success rates, and reduces energy consumption and cost for user devices while accelerating the learning convergence of agents.
3) We conducted extensive simulation experiments to evaluate the performance of the proposed algorithm. The simulation results demonstrate that our approach significantly accelerates the convergence speed of DRL agents. Compared to standard baseline algorithms, our method achieves superior performance in improving task success rates, reducing system energy consumption, lowering task costs, and minimizing task cold start frequency.

The remainder of this article is organized as follows. In Section II, we review related work. Section III presents the problem scenario and the modeling process. Section IV introduces the algorithm design. Section V describes our simulation experiments and evaluation. Finally, Section VI concludes this article.

II. Related Work

Methods for Alleviating Cold Starts in Serverless Computing: Serverless computing refers to a model where users are not required to manage infrastructure servers and can focus solely on code and business logic development. The allocation and elastic scaling of computational resources are dynamically managed by cloud service providers on-demand. Based on the characteristics of serverless computing, cloud providers must instantiate a specific runtime environment for the first invocation of a function request. This process is known as the cold start process [9].

Methods for alleviating the cold start problem in serverless computing can be categorized into two main approaches. The first approach does not avoid the occurrence of cold starts but focuses on reducing cold start latency to improve the execution of latency-sensitive functions. For example, the WebAssembly runtime approach is used to initialize the runtime environment at near machine-code speeds [19]. Wang et al. [20] designed a highly scalable middleware, FaaSNet, to reduce cold start latency. The second approach alleviates cold starts by prewarming functions and reusing function instances, ensuring that requests are executed as much as possible via warm starts. Prewarming refers to the practice of proactively loading the

runtime environment before the arrival of function requests, thereby avoiding cold start issues upon request arrival. Reusing means that after the function execution is completed, the runtime environment is not immediately destroyed, but is retained for a certain period (typically 5 to 15 min), allowing subsequent function requests to reuse the existing warm instances. For example, Zhao et al. [2] and Golec et al. [21] investigated the correlation between the occurrence of cold starts and the number of requests sent to servers. Dehury et al. [22] examines the percentage of user requests handled by fog and cloud, maintaining function instance warmth by optimizing the number of requests allocated to different service nodes. Qiao et al. [23] improved prewarming rates by solving near-optimal cache replacement strategies with machine learning. Fuerst and Sharma [10] treated warm function containers as cache objects, while Chiang et al. [24] modeled the container warming control problem as a MDP. COCOA [25] explores the correlation between cold starts and time-to-live (TTL) caching, optimizing system resource consumption and cold start frequency by adjusting container lifetimes. Vahidinia et al. [11] leveraged reinforcement learning to determine the optimal time for maintaining container warmth and use long short-term memory (LSTM) to identify required prewarmed container types. Their joint optimization method for prewarming and reuse mitigates cold starts. However, prewarming or reuse methods do not alter the arrival pattern of task requests at service nodes. Instead, these methods observe task request patterns to prewarm, extend, or shorten the lifecycle of function instances for reuse. The effectiveness of prewarming or caching reuse relies on prediction accuracy. Barcelona-Pons et al. [26] explored sharing stateful java virtual machine (JVM) instances across functions. However, designing a universal runtime image remains challenging and can result in cumbersome runtime environments.

Offloading Strategies in SEC: Due to the resource constraints of edge nodes, lightweight platform frameworks such as TinyFaaS [27] and UnFaaSener [28] are designed to reduce system resource consumption. Pandey and Kwon [29] predicted and reduces excessive memory demands of SFs. Work in [15] proposes multiagent algorithms to address task scheduling issues among noncooperative edge nodes. Lu et al. [30] introduced an algorithm based on deep deterministic policy gradient (DDPG) to enhance user experience quality, mitigating instability and slow convergence in task offloading within edge computing. To improve resource utilization, Russo et al. [4], [31] optimized resource costs across edge and cloud paradigms using horizontal and vertical offloading strategies. Yao et al. [32] employed a distributed reinforcement learning framework with cloud training and edge execution to address the challenges of sample diversity and high exploration costs in offloading strategies for SEC. Mampage et al. [33] designed heuristic algorithms for deadline-sensitive tasks, aiming to minimize provider costs associated with maintaining cloud infrastructure. Bilal ety al. [34] separated memory and CPU allocation through analytical models. Tang et al. [35] utilized partially observable MDP (POMDP) and decentralized POMDP approaches to minimize IoT device energy consumption while meeting task processing latency requirements. Their goal is to efficiently utilize system resources on SEC platforms, reduce average task latency and energy consumption, but they overlook the impact of cold start issues.

To address the latency of executing latency-sensitive functions, in [36], a multiagent DDPG (MADDPG) method is proposed to enable efficient task offloading and resource allocation for IoT AR application requests. Xu et al. [37] presented an effective heuristic online learning-driven algorithm to address stateful serverless application deployment with function dependencies. Suresh et al. [38] classified incoming SFs requests based on resource consumption and lifecycle patterns, optimizing container placement both within and across containers. To optimize specific SFs offloading strategies, Xu et al. [39] adopted different memory usage patterns for containers at various stages to balance memory shortages and cold start issues. Lou et al. [40] determined the execution priority of SFs based on execution time, cold start time, and the number of queued requests. Chen et al. [16] and Kim et al. [41] modeled the cold start problem as a container switching cost problem, utilizing an integer linear programming model to optimize container switching, communication, and runtime costs for function request offloading. Li et al. [42] proposed a new hybrid offloading algorithm within a cloud–edge–end three-tier serverless framework, employing a greedy approach to reduce average latency. Wang et al. [43] introduced the coefficient of variation (CV) to measure irregularities in function invocations and uses reinforcement learning to resolve issues related to redundant containers or code prefetching. Agarwal et al. [44] reduced the frequency of cold starts for specific workloads by determining the optimal number of function instances in advance through reinforcement learning. These offloading and scheduling strategies to mitigate cold starts primarily rely on quantitative analyzes of cold start latency to improve service quality. However, they do not further alleviate cold starts through function instance reuse or optimize resource utilization.

From Table I, it can be observed that existing works on SEC primarily focus on the design of architectures, platforms, and models, aiming to optimize certain system performance metrics such as latency, energy consumption, and cost. However, these optimization approaches lack consideration for the cold start latency of tasks in SEC. Additionally, they fail to fully leverage the distributed nature of edge computing and often overlook long-term performance optimization. This article addresses these issues by establishing a cold-start-aware multiagent DRL offloading model. The proposed model improves the execution success rate and service quality of latency-sensitive SFs requests while reducing task latency, energy consumption, and cost. Furthermore, it balances the tradeoffs between cold start frequency and resource utilization costs in SEC.

III. System Model

In this section, the problem scenarios and problem modeling of cold-start-aware cloud–edge collaborative computing are outlined. The challenges faced by traditional cold-start mitigation methods and SFs offloading strategies are summarized.

TABLE I
SUMMARY OF EXISTING SERVERLESS COMPUTING WORKS

Ref.	Problem Challenges	Solutions	Cold Start Optimization Considered	Optimization Goals	Parameters	Limitations
[32]	1.Sample diversity 2.High exploration in cloud-edge	Experience Share in DRL	No consider	1.Average latency 2.Convergence speed	1.Fixed cold start latency 2.The delay of max tolerable	1.No SF reuse 2.No vertical offloading
[27] [28]	Lightweight edge serverless platform	Pub/Sub, CoAP	No consider	1.Delay 2.Cost	1.Code size of SF, 2.Invocation cost	Longer livetime container
[2] [21]	High cold start frequency in edge	Machine learning	Considered	Reduce cold start frequency	1.SF requests Num 2.Cold start latency	Higher request to keep SF warm
[45] [46]	Dynamic task offloading in cloud	MQTT Apache NiFi	No consider	1.Latency 2.Security	1.Task dependencies 2.Communication topology	No vertical offloading to further reduce latency
[10] [25] [24] [11] [41] [23] [44]	Mitigating cold start issues in cloud-edge	Caching,reuse, pre-warming	Considered	1.Cold start frequency 2.Resource utilization	1.Cache time 2.Cold start latency 3.Tolerable latency	No offloading strategy
[20] [19]	Reducing cold start latency in cloud	WebAssembly, container configuration	Considered	Shorten the cold start process	1.Cold start latency 2.Network trans.	No optimize cold start occur
[15]	Task offloading in edge	DQN	No consider	1.Service quality 2.Resource utilization	1.Cold start latency 2.Processing capacity 3.Network trans	Fixed cold start latency
[31] [4]	Framework across edge and cloud	Real-time migrate, request queues	No consider	1.Task offloading 2.Migration ways	1.Response time, 2.Request throughput	Supports one type SF
[16]	Request dispatch, container cache in edge	Integer linear programming	No consider	1.Latency cost, 2.Operating cost	1.Cold start cost 2.Edge resources 3.Trans. cost	Fixed cold start
[22]	SF deployment in cloud-fog	DRL	No consider	Percentage of requests in fog	1.User priority, 2.Distance,3.Latency 4.Resource available	Fixed cold start
[42]	SF offloading in multi-edge to cloud	Multi-hop communication	No consider	Min execution latency	1.Average latency 2.Edge resources 3.Device resources	Average cold start in edge and cloud
[40]	Cold start latency request blocking in cloud-edge	Polynomial time schedule and enhanced shortest	Considered	Min execution latency	1.Average latency 2.Edge resources 3.bandwidth demand	Fixed cold start with horizontal offloading
[our]	Cold start frequency, Multi-SFs offloading with Multi-users in Cloud-Edge-End	SF warming pool, SF Priority sched -uling by DQN, Importance Samp -ling	Considered	1.Reduce cold start frequency, 2.Enhance QoS 3.Resource utilization	1.Max task latency 2.All layers resource 3.Network trans. 4.SF priority	Maximize the success rate of all task types, no horizontal offloading

By defining multiple control decisions for three types (cloud, edge) and further characterizing the total latency, cold-start frequency, total energy consumption, and cost models in SFs offloading, the long-term multiobjective optimization problem is formulated as an MDP problem. Table II summarizes the main symbols used in this article.

A. Problem Scenario

This article considers a SEC network, which consists of intelligent industrial IoT clients, edge computing nodes, and a cloud center as the three main components. As shown in Fig. 1, in the SEC environment, there are N IoT devices $N = \{1, 2, 3, \ldots, N\}$ and M edge servers $M = \{1, 2, 3, \ldots, M\}$ equipped. IoT devices are connected to edge servers via wireless links, and edge servers are connected to the cloud server C through the core network. Each IoT device has specific computing capabilities. Task requests are generated by intelligent industrial IoT devices, such as SF tasks for resizing, grayscale adjustment, and detection in cloud–edge collaborative reasoning [47]. Specifically, maintenance of SF code warm pools on various service nodes is implemented to guide whether SF request are executed in a cold or warm start manner. The survival time of each SF instance in the code pool depends on the corresponding function type.

In this scenario, each IoT device is a user, and each user decides the offloading location for its function requests. Cost calculations are made from the perspective of IoT users, including energy consumption and monetary costs.

Energy Consumption: This includes the computational energy consumption incurred when the task is executed on IoT devices and the transmission energy consumption incurred when the task is offloaded to edge or cloud servers.

Monetary Costs: This includes the request and computation prices when tasks are processed on edge or cloud servers. Tasks computed locally on IoT devices incur no monetary costs. The monetary cost of a task is proportional to the computation time on the server nodes.

Specifically, when SF requests are executed on IoT devices, they exclusively utilize the devices' resources, incurring only computational resource consumption without monetary costs. When SF requests are executed on edge nodes, they share

TABLE II
LIST OF NOTATIONS

Symbol	Meaning
N	the set of IIoTEs
M	the set of edge servers
C	Cloud computing center
$R^f_{n_i}(t)$	The i-th request of device n at timeslot t
$B^f_{n_i}(t)$	The data size of task $R^f_{n_i}(t)$
$D^f_{n_i}(t)$	The CPU cycles needed of task $R^f_{n_i}(t)$
$\tau^f_{n_i}(t)$	Maximum tolerable latency of task $R^f_{n_i}(t)$
F	Set of serverless functions
$pool_{L_n}$	Device n instance pools
$pool_{E_m}$	Edge server m instance pools
$pool_C$	Cloud instance pools
$a_{n_i}(t)$	Action taken of task $R^f_{n_i}(t)$
$r_{edge}(t)$	Maximum transmission rate n to edge x
$r_{cloud}(t)$	Maximum transmission rate n to cloud C
$P_n(t)$	Transmission power of device n at timeslot t
$Price_{mx}$	Computational price per unit time at edge x
$Request_{edge}(t)$	Price for requesting edge x computation
$Price_c$	Service price per unit time at cloud C
$Request_{cloud}(t)$	Price for requesting cloud computation
$T_{local}(t)$	The total delay of task $R^f_{n_i}(t)$ in local
$T_{edge}(t)$	The total delay of task $R^f_{n_i}(t)$ in edge
$T_{cloud}(t)$	The total delay of task $R^f_{n_i}(t)$ in cloud
$E_{compute}(t)$	The calculate energy consumption
$E_{tran}(t)$	The transmite energy consumption
$T_{wait}(t)$	Waiting transmite and resource allocate
$T_{tran}(t)$	The transmission delay of task $R^f_{n_i}(t)$
$T_{start}(t)$	The cold start delay of task $R^f_{n_i}(t)$
$T_{compute}(t)$	The computation delay of task $R^f_{n_i}(t)$
$T_{n_i}(t)$	Total latency of the task $R^f_{n_i}(t)$ with $a_{n_i}(t)$
$E_{n_i}(t)$	Total energy of the task $R^f_{n_i}(t)$
$M_{n_i}(t)$	The total monetary of task $R^f_{n_i}(t)$
ϕ	The energy coefficient
s	State vector representing the SEC observation

the computational resources of the edge node, resulting in lower transmission latency and energy consumption but higher monetary costs. When SF requests are executed in the cloud computing center, they utilize larger computational resources exclusively, incurring relatively lower monetary costs, and results in the highest transmission latency and energy consumption. Within the same service node, the processing order of SF requests follows a first-come-first-served (FCFS) principle.

Typically, the SF code pool only needs to store a single image of each type of SF. If no corresponding requests are made within the SF's keep-alive time, its function code is removed from the SF code pool, adhering to the elasticity principle of serverless services. When an SF request is found in the code pool, its initialization time is negligible, and the request is processed as a warm start. If the SF request misses in the code pool, the SF image code must be fetched from the remote image repository, resulting in a cold start. The cold start cost of each type of function depends on the size of its image code and the network resource environment.

To mitigate the frequency of cold starts for heterogeneous SEC SF requests, a multiagent cold-start-aware task offloading strategy is designed. This strategy not only reduces the frequency of cold starts but also lowers the processing time of SF requests, improving service quality while avoiding excessive consumption of system resources and monetary costs. Next, we formulate a joint optimization model with multiple decision variables, considering cold-start awareness, delay cost, energy consumption, and monetary cost.

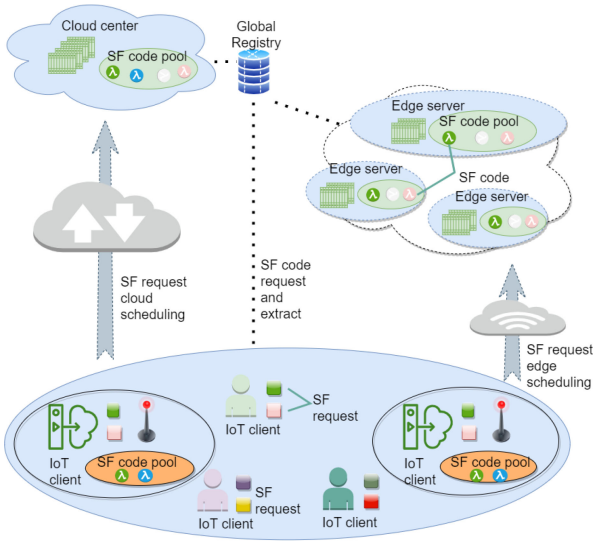


Fig. 1. Example of SF request offloading in IIoT with cloud–edge collaborative SEC. During each time period, IoT clients generate tasks, and the decision agent deployed on each device evaluates the attributes of the tasks and the current environmental conditions (e.g., the number of tasks for various SF requests in local, edge, and cloud nodes, the total number of all task requests, and the lifecycle of the SF code pool). Based on this evaluation, the agent makes decisions regarding task offloading and resource allocation. The agent then issues these offloading and resource allocation instructions, which are executed by the respective devices and servers.

B. Task Model

Time is divided into equally spaced discrete time slots $\{1, 2, 3, \ldots, T\}$, with a fixed time interval τ in each time slot. Intelligent IoT devices generate an application request at the beginning of each time slot with a certain probability. The IoT client decomposes this task into i independent SF computation tasks, which are represented as $R^f_{n_i}(t) = \{B^f_{n_i}(t), D^f_{n_i}(t), \tau^f_{n_i}(t)\}$, where $R^f_{n_i}(t)$ denotes the ith request task of device n at time slot t, belonging to function f, $B^f_{n_i}(t)$ represents the task data size, $D^f_{n_i}(t)$ indicates the number of CPU cycles required to complete task $R^f_{n_i}(t)$, and $\tau^f_{n_i}(t)$ signifies the maximum tolerable latency for task completion (deciding the success of task execution), which is one of the main constraints in the optimization problem. Here, $f \in F = \{1, 2, \ldots, F\}$, where F represents the function set of the SEC network. Due to the resource constraints of the distributed server infrastructure, the sizes of pools in IoT device n, edge m, and cloud instance C are denoted as pool_{L_n}, pool_{E_m}, pool_C, where $\text{pool}_{L_n} < \text{pool}_{E_m} < \text{pool}_C$.

When task $R^f_{n_i}(t)$ arrives, a decision on task offloading $a_{n_i}(t)$ needs to be made, represented as $a_{n_i}(t) = \{a_{nn}, a_{nm1}, a_{nm2}, \ldots, a_{\text{nmm}}, a_{nc}\}$, where $a_{nn}, a_{nm1}, a_{nm2}, \ldots, a_{\text{nmm}}, a_{nc} \in (0, 1)$, and $a_{nn} + a_{nm1} + a_{nm2} + \cdots + a_{\text{nmm}} + a_{nc} = 1$. For example, when $a_{nn} = 1$, all others are 0, indicating that this task will be executed locally. When $a_{nm1}(t) = 1$, with all others being 0, this task will be

offloaded to edge server 1, and so on. This approach ensures that tasks are not migrated to devices or servers at the same hierarchical level and avoids repeated offloading.

C. Transmission Model

Considering the transmission model of SFs requests in the actual industrial IoT environment, as shown in Fig. 1, where N IoT devices are all connected to the edge base station via wireless channels, the maximum uplink rate from industrial IoT device n to edge server x at time slot t is given by: $r_{\text{edge}}(t) = W_n(t)\log(1 + [P_n(t)h_{\text{edge}}(t)/\sigma^2 + N_{\text{edge}}])$ here, $W_n(t)$ represents the channel bandwidth, $P_n(t)$ denotes the transmission power allocated at time slot t, $h_{\text{edge}}(t)$ signifies the wireless channel gain from end device n to edge server, σ^2 is the background noise variance, and $N_{\text{edge}} = \sum_{n' \in N, n' \neq n} P_n(t)h_{\text{edge}}(t)$ represents the interference signal-to-noise ratio (SINR). Similarly, $r_{\text{cloud}}(t)$ denotes the fixed uplink rate from IoT device n to the cloud server. The transmission delay of task $R^f_{n_i}(t)$ from end device n to edge or cloud server at time slot t can be calculated as follows:

$$T_{\text{tran}}(t) = \begin{cases} \frac{B^f_{n_i}(t)}{r_{\text{edge}}(t)}, & \text{if } a_{\text{nmx}} = 1 \\ \frac{B^f_{n_i}(t)}{r_{\text{cloud}}(t)}, & \text{if } a_{nc} = 1. \end{cases} \tag{1}$$

The transmission energy consumption $E_{\text{tran}}(t)$ of task $R^f_{n_i}(t)$ is represented as

$$E_{\text{tran}}(t) = P_n(t) \cdot T_{\text{tran}}(t). \tag{2}$$

D. Computing Model

Total delay in SF computation includes waiting transmission delay, transmission delay, waiting computation delay, cold start delay, and computation delay. Specifically, when an SF request is offloaded to an IoT device for computation, both waiting transmission delay and transmission delay are 0 ($T_{\text{tran}}(t) = 0$). When an SF request is executed on a compute node using warm start, cold start delay is 0. Typically, in SEC computation, the returned data volume of SF requests is very small, hence download delay for the results is not considered.

Total energy consumption of tasks primarily refers to the battery energy consumption of IoT devices, including energy consumption for transmitting SF tasks to other service nodes and local computation. The transmission energy consumption for offloading SF requests to the cloud is greater than offloading to the edge, but less than the energy consumption generated when processing locally.

In SEC, tasks incur two monetary costs when computed on edge or cloud servers, which are the request price for edge or cloud computation, and the computation cost for edge or cloud computation. Generally, the price for requesting edge computation and the unit computation price over time are higher than those of cloud servers, but the transmission delay is lower. The closer the computation, the more expensive the computation price, the smaller the computing capacity, and the lower the transmission delay.

1) Local Computing Model: In Industrial Internet of Things (IIoT), IoT devices are equipped with more powerful chips and batteries, enabling them to handle computational tasks akin to a slightly weaker server. Therefore, the SF task $R^f_{n_i}(t)$ can be offloaded to the local IoT device n, with $T_{ni}(t)$ representing the total delay when offloaded to device n for computation, expressed as

$$T_{\text{local}}(t) = T_{\text{wait}}(t) + T_{\text{start}}(t) + T_{\text{compute}}(t) \tag{3}$$

here, $T_{\text{wait}}(t)$ denotes the delay for resource allocation, $T_{\text{start}}(t)$ represents the cold start delay of the function instance, and $T_{\text{compute}}(t)$ is the computation delay. $T_{\text{start}}(t) = \begin{cases} \frac{\text{FS}_f}{\text{HP}_n}, & \text{cold start} \\ 0, & \text{warm start} \end{cases}$, where FS_f is the resource package size of the SF f, and HP_n is the maximum load for device n to obtain the resource package. $T_{\text{compute}}(t) = [D^f_{n_i}(t)/f_n(t)]$, where $f_n(t)$ is the processing capability of device n. The task transmission energy consumption $E_{\text{tran}}(t)$ is 0. And the energy consumption calculation is given by

$$E_{\text{local}}(t) = E_{\text{compute}}(t) = \phi \times \left(f_{n_n}(t)\right)^2 \times D^f_{n_i}(t) \tag{4}$$

here, $\phi = 10^{-26}$ denotes the energy coefficient, which typically depends on the hardware platform. It characterizes the energy consumption per unit computational workload and per unit squared CPU frequency [18].

2) Edge Computing Model: In the offloading decision $a_{n_i}(t)$, where $a_{\text{nmx}} = 1$ and all others are 0, indicating that the SF task $R^f_{n_i}(t)$ offloads to the edge server x for execution, the total latency for its execution is given by

$$T_{\text{edge}}(t) = T_{\text{tran}}(t) + T_{\text{wait}}(t) + T_{\text{start}}(t) + T_{\text{compute}}(t) \tag{5}$$

here, $T_{\text{wait}}(t)$ represents the waiting transmission latency and the waiting time for computing resources, $T_{\text{tran}}(t)$ represents the transmission latency, $T_{\text{start}}(t) = \begin{cases} \frac{\text{FS}_f}{\text{HP}_{\text{edge}}}, & \text{cold start} \\ 0, & \text{warm start} \end{cases}$ represents the cold start delay, and $T_{\text{compute}}(t) = [D^f_{n_i}(t)/f_{\text{edge}}(t)]$ represents the computation latency in edge server. HP_{edge} is the maximum load for edge server to obtain the SF code from remote global registry, and $f_{\text{edge}}(t)$ is the computation resource allocated by edge server x to tasks from IoT device n, satisfying $\sum_{n=1}^{N} f_{\text{edge}}(t) \leq F_{mx}$, where F_{mx} denotes the total computing capacity of edge server x.

The total energy consumption for task $R^f_{n_i}(t)$ executed on edge server x is $E_{\text{edge}}(t)$

$$E_{\text{edge}}(t) = E_{\text{tran}}(t) + E_{\text{compute}}(t). \tag{6}$$

And the total monetary expenditure is $M_{\text{edge}}(t)$

$$M_{\text{edge}}(t) = \text{Request}_{\text{edge}}(t) + \text{Price}_{mx} \times T_{\text{compute}}(t) \tag{7}$$

here, Price_{mx} is the computational price per unit time for edge server x.

3) Cloud Computing Model: Similar to edge computing, when the vector $a_{n_i}(t)$ has $a_{\text{nmc}} = 1$ and all other elements are 0, task $R^f_{n_i}(t)$ is offloaded and executed at a cloud computing center. Considering that the cloud server is capable of handling various tasks and is equipped with sufficient network and computing resources, tasks arriving at the cloud computing center do not need to wait for other tasks to execute. They can start immediately in a cold or warm start mode after arriving

at the cloud. The total processing time delay $T_{\text{cloud}}(t)$ for offloading tasks to the cloud is given by

$$T_{\text{cloud}}(t) = T_{\text{tran}}(t) + T_{\text{wait}}(t) + T_{\text{start}}(t) + \frac{B^f_{n_i}(t)}{r_{\text{cloud}}(t)} \tag{8}$$

where $T_{\text{start}}(t) = \begin{cases} [\text{FS}_f/\text{HP(cloud)}], & \text{cold start} \\ 0, & \text{warm start} \end{cases}$ represents the cold start time at the cloud computing center, HP(cloud) represents the maximum load at which the cloud computing center acquires the SF code. The energy consumption $E_{\text{cloud}}(t)$ and monetary expenditure $M_{\text{cloud}}(t)$ for executing task $R^f_{n_i}(t)$ at the cloud computing center are given by

$$E_{\text{cloud}}(t) = P_n(t) \times T_{\text{tran}}(t) \tag{9}$$

$$M_{\text{cloud}}(t) = \text{Request}_{\text{cloud}}(t) + \text{Price}_c \times T_{\text{compute}}(t) \tag{10}$$

where Price_c is the price for cloud server services per unit time.

E. Problem Formulation

In the SEC, the SFs may need to run on the local IoT device due to long waiting times to begin task processing within the maximum delay possible. The SFs may also need to be offloaded to cloud or edge servers due to long waits for local resource allocation. However, cloud servers are distant and cost-effective, while edge servers are closer but expensive. Specifically, the long cold start initialization operation of the SFs significantly reduces the QoS the SEC system. Specifically, the prolonged cold start initialization of the SFs significantly degrades the QoS in the SEC system. Therefore, achieving cold-start awareness can mitigate the cold start issue from the source of SFs requests, saving resource consumption and cost expenses. The goal of this article is to consider the frequency of cold starts of SFs in the resource-constrained SEC environment and achieve joint optimization of success rate, total delay, total energy consumption, and total monetary expenditure.

The total delay $T_{n_i}(t)$, total energy consumption $E_{n_i}(t)$, and total monetary expenditure $M_{n_i}(t)$ of task $R^f_{n_i}(t)$ under SEC computation are represented as follows:

$$T_{n_i}(t) = \left[T_{\text{local}}(t), \sum_{x=1}^{M} T_{\text{edge}}(t), T^i_{\text{cloud}}(t) \right] \cdot \left[a_n n(t), \sum_{x=1}^{M} a_n mx(t), a_n c(t) \right]^T \tag{11}$$

$$E_{n_i}(t) = \left[E_{\text{local}}(t), \sum_{x=1}^{M} E_{\text{edge}}(t), E_{\text{cloud}}(t) \right] \cdot \left[a_n n(t), \sum_{x=1}^{M} a_n mx(t), a_n c(t) \right]^T \tag{12}$$

$$M_{n_i}(t) = \left[0, \sum_{x=1}^{M} M_{\text{edge}}(t), M_{\text{cloud}}(t) \right] \cdot \left[a_n n(t), \sum_{x=1}^{M} a_n mx(t), a_n c(t) \right]^T. \tag{13}$$

The cold start delay of task $R^f_{n_i}(t)$ in SEC system is $T_{\text{start}}(t)$, if $T_{\text{start}}(t) > 0$, it means that task $R^f_{n_i}(t)$ is executed in a cold start manner under offloading policy $a_{n_i}(t)$. Otherwise, SFs request $R^f_{n_i}(t)$ is executed in a warm start manner.

In SEC system, due to the waiting for network bandwidth resources and differences in allocated computing resources for tasks, the actual execution time of a task may exceed the maximum tolerated time. Therefore, using $\text{Success}_{n_i}(t)$ to denote whether a task meets the completion condition under the maximum delay

$$\text{Success}_{n_i}(t) = \begin{cases} 1, & T_{n_i}(t) \le \tau^f_{n_i}(t) \\ 0, & T_{n_i}(t) > \tau^f_{n_i}(t). \end{cases} \tag{14}$$

If $\text{Success}_{n_i}(t) = 1$, it signifies that task $R^f_{n_i}(t)$ is successfully completed within the maximum tolerated delay $\tau^f_{n_i}(t)$. Otherwise, SFs request $R^f_{n_i}(t)$ did not successfully execute.

The more SFs requests task with cold start in SEC, the greater the total delay of tasks. Therefore, implementing cold-start aware offloading strategies can reduce the frequency of cold start in SEC and also reduce the overall system cost. Transforming the offloading problem of multiple SFs into a joint optimization problem, the problem can be defined as

$$\begin{aligned} \mathbb{P}: \text{Minimize } & \Big(\textstyle\sum_{t=1}^{T} \sum_{n=1}^{N} \sum_{i=1}^{R} T_{n_i}(t) \text{ and } \\ & \textstyle\sum_{t=1}^{T} \sum_{n=1}^{N} \sum_{i=1}^{R} E_{n_i}(t) \text{ and } \\ & \textstyle\sum_{t=1}^{T} \sum_{n=1}^{N} \sum_{i=1}^{R} M_{n_i}(t) \Big) \\ \text{s.t. } & C1: T_{n_i}(t) \le \tau^f_{n_i}(t) \\ & C2: a_{nn} + a_{nm1} + a_{nm2} + \cdots + a_{\text{nmm}} + a_{nc} = 1 \\ & C3: \sum_{t=1}^{T} \sum_{n=1}^{N} \sum_{i=1}^{R} E_{n_i}(t) \le E_n. \end{aligned} \tag{15}$$

Here, $C1$ denotes the total completion delay constraints of the delay-sensitive tasks, and $C2$ denotes that each task can only be offloaded to one computing node. $C3$ means the energy consumption of tasks does not exceed the device's battery capacity.

IV. Design of CSODQN

The objective of this article is to address the cold start problem in SEC. Unlike methods such as preheating and caching used to mitigate cold starts, this approach jointly optimizes the cold start tasks with resource consumption and computation costs as an offloading problem, aiming to achieve a tradeoff between cold starts and resource consumption. The joint optimization problem $\mathbb{P}$ of function offloading is modeled as a MDP. In the SEC, as the number of IoT devices and edge devices increases, the state and action spaces of the MDP grow exponentially, making it difficult to solve the optimization problem in polynomial time. The capability of deep learning to handle high-dimensional state spaces and the strong learning ability of reinforcement learning, a DRL-based algorithm is designed for computation offloading and resource allocation to maximize the expected reward.

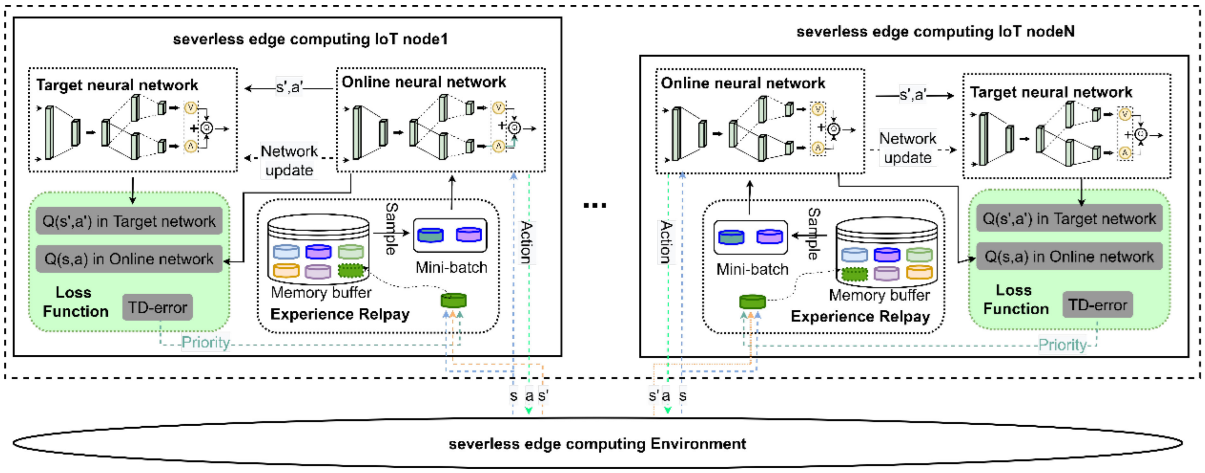


Fig. 2. Learning and sampling process of CSODQN. During CSODQN training, the agent collects system resource as the computing ability of edge server,then input of D3QN, and the network output is Q-value. The agent performs the corresponding action, receives the reward, and the system state is updated to the next state. The agent stores the collected MDP sets into the priority experience replay buffer, and selects training samples according to the sampling probability to train the D3QN network repeatedly.

A. MDP

Due to the complexity of the heterogeneous cloud–edge collaborative SEC environment, it is nearly impossible to fully observe all state transitions. Therefore, the SEC computing environment is modeled as an MDP with internal state transition probabilities ρ. An MDP consists of a 5-tuple $M = \{S, A, \rho, R, \gamma\}$, where S represents the state space, A represents the action space, ρ is the state transition probability, R is the reward function, and γ is the discount factor for future rewards. At the beginning of each time slot t, when task $R^f_{n_i}(t)$ arrives, the decision maker perceives the current environment state $s_{n_i}(t) \in S$ and executes an action $a_{n_i}(t) \in A$. After the action $a_{n_i}(t)$ is completed, the system state is updated to the next state $s_{n_i}(t+1)$, and the environment provides a reward $r_{n_i}(t) \in R$ to the agent. The details of the MDP we designed are as follows.

1) *State Space:* At the beginning of each time slot t, the agent collects information about the current SEC environment. The system state is represented as a 5-tuple $S(t) = \{R(t), F_n(t), F_m(t), F_c(t), B(t)\}$, where $R(t)$ represents the task profile to be executed, $F_n(t)$ represents the computation resources of the terminal device, $F_m(t)$ represents the computation resources of the edge server, $F_c(t)$ represents the computation resources of the cloud center, and $B(t)$ represents the remaining bandwidth resources of the wireless channel. $S(t)$ is the state of the environment at time t before the task $R(t)$ decision is made.

2) *Action Space:* The action space represents the number of offloaded work nodes. The agent, by observing the current state, executes an action $a_n(t) = \{a_{nn}, a_{nm1}, a_{nm2}, \ldots, a_{\text{nmm}}, a_{nc}\}$, as described in Section III-B.

3) *Reward:* The objective is to complete the task execution within the maximum delay, with the reward defined as the positive reward for task execution success and the penalty for execution failure, as well as the penalty for the system cost incurred during task execution. By adjusting the weights of the execution outcome state rewards ($R_{\text{Success}_{n_i}}$, $R_{\text{Failed}_{n_i}}$) and Sys_cost_{n_i} is the system cost, the final reward function is designed. This aims to achieve a higher success rate with minimal system cost consumption, thus improving QoS while maximizing the utilization of system costs

$$\text{Sys\_cost}_{n_i} = \alpha \times T_{n_i}(t) + \beta \times E_{n_i}(t) + \gamma \times M_{n_i}(t) \quad (16)$$

where α, β, and γ are coefficients representing the contribution of total task delay, total energy consumption, and total monetary expenditure to the system cost

$$R_{\text{Success}_{n_i}} = a \times e^{-\frac{\omega T_{n_i}(t)}{\tau_{n_i}(t)}} + w \times f_{\text{id}} \quad (17)$$

$$\begin{aligned} R_{\text{Failed}_{n_i}} = &-a \times \log(f_{\text{id}}) \times R_{\text{Success}_{n_i}} \\ &-b \times e^{-\frac{\omega T_n(t)}{\tau_n(t)}} \\ &-c \times \log(f_{\text{id}} + \text{failed\_state}) \end{aligned} \quad (18)$$

where a, b, c, w, and ω are coefficients that control the rewards for lower-priority SFs request task types, ensuring that their absolute values do not exceed those for higher-priority SFs request task types. f_{id} represents the priority of the current task type, with a higher f_{id} indicating a higher priority task type. The final reward obtained by task $R^f_{n_i}(t)$ is calculated as

$$\begin{aligned} \text{reward}_{n_i}(t) = e \times \Big(& R_{\text{Success}_{n_i}} \times \text{Success}_{n_i}(t) \\ & + (1 - \text{Success}_{n_i}(t)) \times R_{\text{Failed}_{n_i}} \Big) \\ & - f \times \text{Sys\_cost}_{n_i}. \end{aligned} \quad (19)$$

Thus, the cumulative reward for reinforcement learning is $\text{Reward} = \sum_{t=1}^{T} \sum_{i=1}^{R} \text{reward}_{n_i}(t)$. The joint optimization problem $\mathbb{P}$ of SFs request offloading is transformed into the calculation of Max(Reward).

B. CSODQN Model

This section describes the design of a multiagent DRL-based algorithm to address the proposed cold-start perceivable task scheduling optimization problem. In our work, it is suitable to use the DQN architecture to learn the optimal task scheduling decisions for SEC. As shown in Fig. 2, based on the DQN architecture, we design an improved

multiagent task scheduling algorithm, cold-start-aware offloading by double-dueling-DQN (CSODQN), based on the Dueling DDQN network. Below, we introduce the method of using the multiagent CSODQN approach to solve the cold-start perceivable task offloading problem.

The network takes the SEC state vector s as input. It first passes through two fully connected layers with 64 neurons each, both activated by ReLU functions. Then, the network splits into two separate branches: 1) a value stream and 2) an advantage stream. The value stream outputs a scalar $V(s)$ through a fully connected layer with linear activation, while the advantage stream outputs an action-wise vector $A(s, a)$ through another fully connected layer with linear activation. The final Q-value is computed by combining these two streams as shown in (20)

$$Q\left(s_{n_i}(t), a_{n_i}(t)\right) = V\left(s_{n_i}(t)\right) + A\left(s_{n_i}(t), a_{n_i}(t); \theta\right) - \sum_{a' \in a_n(t)} A\left(s_{n_i}(t), a'; \theta_{\text{main}}\right)/|A|. \tag{20}$$

$$\delta\left(s_{n_i}(t), a_{n_i}(t)\right) = \text{reward}_{n_i}(t) - Q\left(s_{n_i}(t), a_{n_i}(t); \theta\right) + \gamma \cdot Q\left(s_{n_i}(t+1), \arg\max_{a'} Q(s_{n_i}(t+1), a'; \theta_{\text{main}})\right). \tag{21}$$

To ensure a constraint with an expected value of 0 and thus improve the stability of the Q-value, the equation is modified by subtracting the average value for each A:$Q(s, a) = V(s) + \left(A(s, a) - [\sum_{a'} A(s, a')/|A|]\right)$. Further to enhance the accuracy of the Q-value estimation and the stability of the learning process, the concept of Double DQN is introduced. We use a Double Dueling DQN network for training by adding a target network to determine the action value for Q-value computation. This reduces the overestimation bias, improving the accuracy and stability of the Q-value, as shown in (20), θ and θ_{main} represent the parameters of the online network and the target network.

In this article, multiple agents run simultaneously in a single SEC environment, and their experiences are not shared publicly. Therefore, we adopt centralized prioritized replay, where higher-priority experiences are more likely to be sampled. Samples with significant differences between predicted values and TD targets are prioritized because a large TD error $\delta\left(s_{n_i}(t), a_{n_i}(t)\right)$ indicates that more learning is needed, then represent the priority of each experience $p\left(s_{n_i}(t), a_{n_i}(t)\right) = |\delta\left(s_{n_i}(t), a_{n_i}(t)\right)| + e$, a small constant e ensures that no experience has a priority of 0.

As shown in Algorithm 1, the agent executes the current task's offloading decision on SEC service nodes. The function instance reuse in the SF code pool determines whether the function is executed via a warm start or a cold start. The algorithm returns the state before and after the task execution, along with the reward advantage obtained. The experience storage consists of $\{S_t, A_t, R_{t+1}, S_{t+1}, p_t\}$.

Specifically, we implement a container reuse mechanism in the SEC service nodes, where function instances are cached in a container pool. Upon receiving a task, the system first checks the availability and state of containers in the pool. If a suitable container instance exists, the task is scheduled directly to the existing container, enabling a warm start. Otherwise, a new container is initialized, resulting in a cold start. This reuse mechanism not only reduces task execution latency but also provides state feedback for the agent to make cold-start-aware offloading decisions.

Algorithm 1 Cold Start Execution Model in SEC

Input: State S at time slot t, action A for the current task, maximum tolerable delay T.
Output: Final state S' after task execution and reward obtained.
if A in local IIoT **:**
  Calculate waiting time t_1 in the local execution queue.
  if $t + t_1 \geq T$ **:**
    Return state at time $t + t_1$ and failure penalty.
  Calculate cold-start cost *cold* at time $t + t_1$; replace the earliest container instance if the pool is full.
  if $t + t_1 + cold \geq T$ **:**
    Return state at time $t + t_1 + cold$ and failure penalty.
  Calculate runtime *run* for local task execution.
  if $T - (t + t_1 + cold) < run$ **:**
    Return state at time $t + T$ and failure penalty.
  Return state at time $t + t_1 + cold + run$ and success reward.
if A in remote server **:**
  Calculate waiting time t_2 in the local transmission queue.
  if $t + t_2 \geq T$ **:**
    Return state at time $t + t_2$ and failure penalty.
  Calculate transmission time *tran* to the server.
  if $t + t_2 + tran \geq T$ **:**
    Return state at time $t + t_2 + tran$ and failure penalty.
  Calculate waiting time t_4 in the server's execution queue.
  if $t + t_2 + tran + t_4 \geq T$ **:**
    Return state at time $t + t_2 + tran + t_4$ and failure penalty.
  Calculate cold-start cost *cold* at time $t + t_2 + tran + t_4$; replace the earliest instance if the pool is full.
  if $t + t_2 + tran + t_4 + cold \geq T$ **:**
    Return state at time $t + t_2 + tran + t_4 + cold$, failure penalty.
  Calculate runtime *run* for task execution on the server.
  if $T - (t + t_2 + tran + t_4 + cold) < run$ **:**
    Return state at time $t + T$ and penalty for failure.
  Return state at time $t + t_2 + tran + t_4 + cold + run$ and reward for success.

After assigning each experience a priority, apply a randomized priority, which calculates the probability of sampling a given sample based on its priority. The probability of sampling the tth sample is given by

$$P\left(s_{n_i}(t), a_{n_i}(t)\right) = p\left(s_{n_i}(t), a_{n_i}(t)\right)^{\partial} / \sum_k p_k^{\partial} \tag{22}$$

$$\omega\left(s_{n_i}(t), a_{n_i}(t)\right) = \left(\frac{1}{N \cdot P(s_{n_i}(t), a_{n_i}(t))}\right)^{\beta}. \tag{23}$$

Algorithm 2 Cold-Start-Aware Offloading Algorithm With CSODQN

Initialize: state S, action A, and PER memory N, at the beginning of $t = 0$
for each episode **do**
  Load task $R_{n_i}^f(t)$ from datasets.
  Reset the IIoT system environment.
  while True **do**
    Get $a_{n_i}(t)$ from A and $s_{n_i}(t)$.
    Execute $a_{n_i}(t)$, observe $s_{n_i}(t+1)$, and obtain $r_{n_i}(t)$ with algorithm 1.
    Store $\left(s_{n_i}(t), a_{n_i}(t), r_{n_i}(t), s_{n_i}(t+1)\right)$ into the prioritized experience replay buffer.
    Obtain a batch of samples from the buffer N based on Equation (22) and (23).
    Calculate $Q\left(s_{n_i}(t), a_{n_i}(t)\right)$ with Equation (20).
    if time slot t is the last slot
      break
    end if
  end while
end for

In (22), if $\partial = 0$ corresponds to uniform random sampling, and $\partial = 1$ selects the experience with the highest priority. Randomized priority experience sampling, the sampling probability distribution is altered, which results in the decision engine developing a preference for high-priority actions, but also allowing experiences with lower priorities to be sampled. However, high-priority samples introduce bias, leading to the risk of overfitting. To correct this bias, in (23), we use importance sampling (IS) weights ω_t when updating the Q-network. The weight ω_t adjusts the contribution of each sample to the loss function, N is the size of the prioritized experience replay buffer. At the beginning of training, β is set to a small value (e.g., 0.0005) and gradually increases toward 1, allowing for better bias correction during the later stages of training. The details of the CSODQN-based cold-start-aware offloading and resource allocation algorithm are presented in Algorithm 2.

V. Performance Evaluation

The advantages of the CSODQN algorithm were verified by comparing it with other advanced baselines, including DQN, DDQN, Double Dueling DQN-based D3DQN, earliest start first offloading (ESFO), random offloading decision (Random), offloading only at the edge (Edge0), and offloading only to the cloud (Cloud). To validate the superiority of the CSODQN algorithm, five simulation experiments were conducted. In each experiment, each IoT device generates a task request in each time slot with different task arrival rate. There are 10 types of tasks and a total of 10 000 time slots. To avoid extreme cases that may arise from heuristic algorithms and to ensure the fairness of the experiments, both the ESFO and random offloading heuristic algorithms adopt the same task arrival rate and environment initialization procedures as the reinforcement learning algorithms. For the random offloading algorithm, we perform repeated experiments and use the average result as the final performance metric.

A. Experimental Environment

We consider a cloud computing center in an industrial IoT environment. In this experiment, we consider a scenario with 50 industrial IoT devices, 6 edge servers, and 1 cloud server. All neural networks in the DRL algorithms are four-layer networks, consisting of an input layer, an output layer, and two hidden layers.

B. Parameter Setting

The DRL method in this article is trained online. We need to analyze the effect of different DRL hyperparameter settings (such as learning rate, discount factor and batch size) on the performance of CSODQN. Based on this, the DRL hyperparameters are determined. The parameter settings of this experiment are shown in Table III.

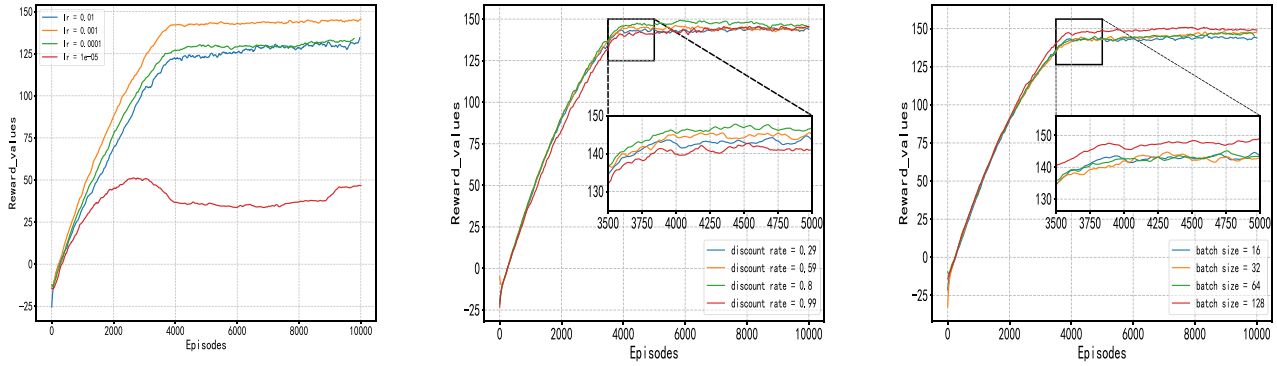
1) Effect of Learning Rate on Convergence of the Reward: Fig. 3(a) illustrates the impact of the learning rate on the reward function's convergence performance in the deep learning process of the CSODQN algorithm. The learning rates are set to 0.01, 0.001, 0.0001, and 0.00001, respectively. When the learning rate is 0.01, the reward curve begins to converge around 4500 episodes. Compared to other convergence curves, the convergence is slower and gets trapped in a local optimum until it fully converges at approximately 6000 episodes. When the learning rate is 0.001, the reward curve converges at 4000 episodes and achieves the optimal reward value compared to other convergence curves. When the learning rate is 0.0001, convergence occurs at 5000 episodes, with its reward curve falling between those of learning rates 0.01 and 0.001. Finally, when the learning rate is 0.00001, the convergence result is the farthest from the optimal solution. Therefore, this experiment selects a learning rate of 0.001 to improve the algorithm's convergence stability while accelerating the convergence speed.

2) Effect of Discount Factor on Convergence the Reward: Fig. 3(b) describes the effect of different discount factors on reward convergence. The discount factors are set to 0.29, 0.59, 0.80, and 0.99. When the discount factor is set to 0.80, both the convergence speed and the final convergence result outperform the other discount factor settings. Our algorithm selects a discount factor of 0.80, as it strikes a balance between immediate rewards and long-term future rewards, allowing more subsequent requests without SF to benefit after the current SF request is processed.

3) Effect of Batch Size on Convergence the Reward: Batch Size refers to the number of samples used to update the neural network parameters in each training iteration. Both too large and too small Batch Sizes can lead to unstable training or poor results. Fig. 3(c) describes the effect of different batch sizes on the convergence speed of the reward function. When the batch sizes are set to 16, 32, and 64, the convergence speed of the reward function is slower compared to a batch size of 128.

TABLE III
EXPERIMENTAL PARAMETER SETTINGS AND SFS LIMIT

System and DRL Parameters	Values	System and DRL Parameters	Values
Number of IIoTEs	[1, 50]	Task Generation Rate	[0, 1]
Number of Edge Servers	[2, 6]	Learning Rate	0.001
Number of Cloud Servers	1	Batch Size	128
SF Request Priority Value	[1,10]	Discount Factor	0.99
SF Code Size	[50KB,500MB]	SF Input Size	[10KB,2.5MB]
SF Request CPU Demand	[1,4]	SF Request Storage Demand	[100KB,2048MB]
SF Request NW I/O demand	[10KB,4MB]	SF Request Offloading Action Space	[4,8]



(a) (b) (c)

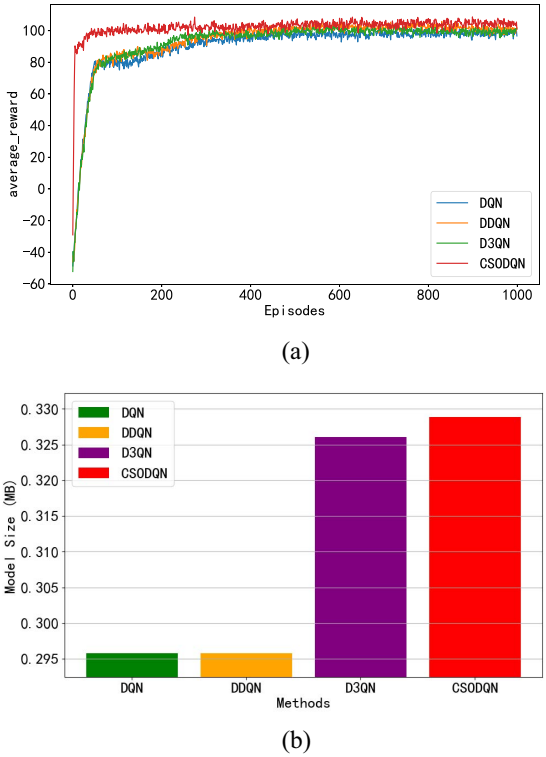
Fig. 3. Experimental evaluation of the impact of DRL hyperparameters (i.e., (a) learning rate, (b) discount factor, and (c) batch size) on the convergence of the reward function.

Furthermore, the convergence result with a batch size of 128 is the optimal one. To ensure good algorithm performance while reducing training time, the subsequent experiments use a Batch Size of 128.

C. Simulation Results

The task success rate refers to the proportion of successfully executed tasks within a unit time slot relative to the total number of tasks. The average task delay represents the average execution time of tasks in the system. A delay closer to the maximum tolerable value indicates a longer average waiting time for tasks, which reflects a poorer system QoS. The average task energy consumption refers to the battery energy consumed by each IoT device for each task. The average cost per task refers to the monetary expenditure incurred for offloading tasks to cloud or edge servers, indicating the cost that each IoT device must pay. The average cold start ratio refers to the proportion of cold start tasks among all completed tasks within a unit time slot, calculated as the number of cold start tasks divided by the sum of cold start tasks, warm start tasks, and tasks that timed out. The average reward of task computing represents the benefits each task earns under different algorithms. A lower average execution time of tasks correlates with higher benefits, while higher task energy consumption and monetary costs lead to smaller benefits. The specific reward calculation is detailed in (19).

1) Training Reward Curves and Model Convergence Size: The training speed of an algorithm affects the rapid deployment and execution of offloading instructions. The results in Fig. 4(a) show a comparison between CSODQN, DQN, DDQN, and D3QN algorithms, evaluating the efficiency of their training speed. CSODQN begins to converge around 50 episodes and reaches the optimal value at 100 episodes. DQN, DDQN, and D3QN all begin to converge around 100 episodes,



(a) (b)

Fig. 4. Training reward curves and model convergence sizes of CSODQN, D3QN, DDQN, and DQN for task offloading across 4 IoT devices under a task arrival rate of 0.4. (a) Original reward curves. (b) Model size (MB).

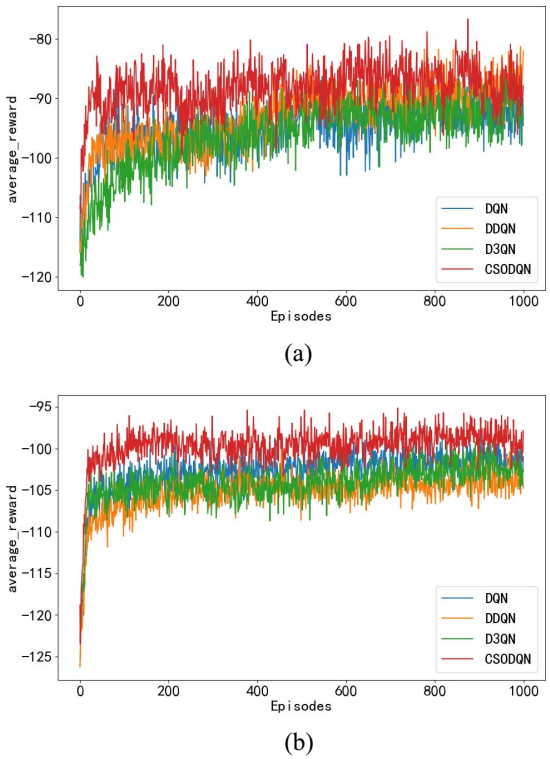


Fig. 5. Training reward curves during the process of training 4 IoT devices, and training reward curves when using the converged model of these 4 IoT devices to train 50 IoT. (a) 4 IoT at 0.6 arrival rate. (b) 50 IoT at 0.6 arrival rate.

with D3QN converging by 300 episodes, and DQN and DDQN converging by 400 episodes. CSODQN outperforms the other algorithms in terms of both convergence speed and the final convergence result. The network structures of DQN and DDQN are similar, with relatively simple architectures and fewer parameters. When the reward converges, the model sizes are similar and relatively small, both around 0.295 MB, as shown in Fig. 4(b). The model size of CSODQN is similar to that of D3QN, ranging from 0.325 to 0.327 MB. Although the CSODQN model is larger, it exhibits better performance and learning ability. In complex environments, it can make more accurate decisions. The edge scenario does not necessarily prohibit a certain level of resource consumption. Some edge devices (such as industrial gateways, smart cameras, etc.) have relatively high configurations [18], with some resource redundancy that can support the operation of larger models. The accuracy advantage of CSODQN can compensate for the resource usage disadvantage, leading to higher overall gains.

As shown in Fig. 5(a), the reward curves during the training process of 4 IoT devices under a task arrival rate of 0.6 are presented. Fig. 5(b) shows the training and validation process where the trained reinforcement learning model is transferred to a new environment with 50 IoT devices. CSODQN achieves a higher average reward during training, with a converged reward of approximately −98, which is noticeably better than DQN (−102), D3QN (−104), and DDQN (−106). Furthermore, CSODQN exhibits lower post-convergence variance, indicating greater stability. These numerical results demonstrate that even though CSODQN was trained with only 4 IoT devices, it maintains superior performance and stable policy behavior when transferred to an environment with 50 IoT devices. This suggests that CSODQN possesses strong generalization capabilities and adaptability to varying deployment scales. Therefore, it is reasonable to infer that CSODQN remains effective in ultradense edge computing scenarios involving large-scale IoT deployments (e.g., over 1000 devices), maintaining both policy effectiveness and system stability.

2) Performance Comparison With Different Task Arrival Density: Fig. 6 compares CSODQN and other baseline algorithms in terms of service quality and system resource consumption under varying task arrival densities. As the task arrival density increases, the number of requests for SF increases within the same time frame. This leads to an increase in average execution time due to resource waiting, and the task success rate begins to decline. The task success rate and reward benefits of the Random and the ESFO algorithm decline rapidly, demonstrating poor robustness. The cold start probability decreases as well. High-concurrency task requests cause more SF requests to occur within the survival time of their respective type of function, allowing for faster execution through warm starts. As the task arrival rate increases from 0.1 to 0.4, network resources are relatively sufficient, and task execution success rates remain above 80%. However, when the task arrival rate increases from 0.4 to 0.6 or higher, task density increases significantly, causing a sharp decline in task success rates, dropping to below 50%. More tasks reach the maximum delay, and fewer tasks successfully complete and receive positive rewards. More tasks fail to execute successfully, consuming system resources and leading to higher penalties. Nevertheless, CSODQN maintains the highest task success rate and the lowest average task energy consumption compared to D3QN, DDQN, and DQN algorithms, not only detects cold start occurrences effectively but also reduces the system's energy consumption.

3) Performance Comparison With Different Numbers of IIoTEs: As shown in Fig. 7, with the increase in IoT devices, SF requests rise, leading to greater competition for limited edge and network resources. This results in longer average task execution times and a declining success rate. When the number of devices increases from 1 to 6, edge server performance deteriorates due to contention. Reinforcement-learning-based methods maintain a high success rate at the cost of increased delay, while Random and ESFO algorithms experience a sharp drop in success rate, falling below 60%. CSODQN achieves the best success rate with less than a 10% increase in delay. As the number of devices rises from 6 to 20, cold start frequency gradually decreases to below 50%, enabling more warm starts. CSODQN effectively balances multiple objectives, including delay, energy consumption, and success rate.

4) Performance Comparison With Different Keep Alive Time: As shown in Fig. 8, the keep alive (survival time) refers to the period after the execution of a serviceless function, during which the function's runtime environment remains active before being destroyed. For a fixed task arrival rate,

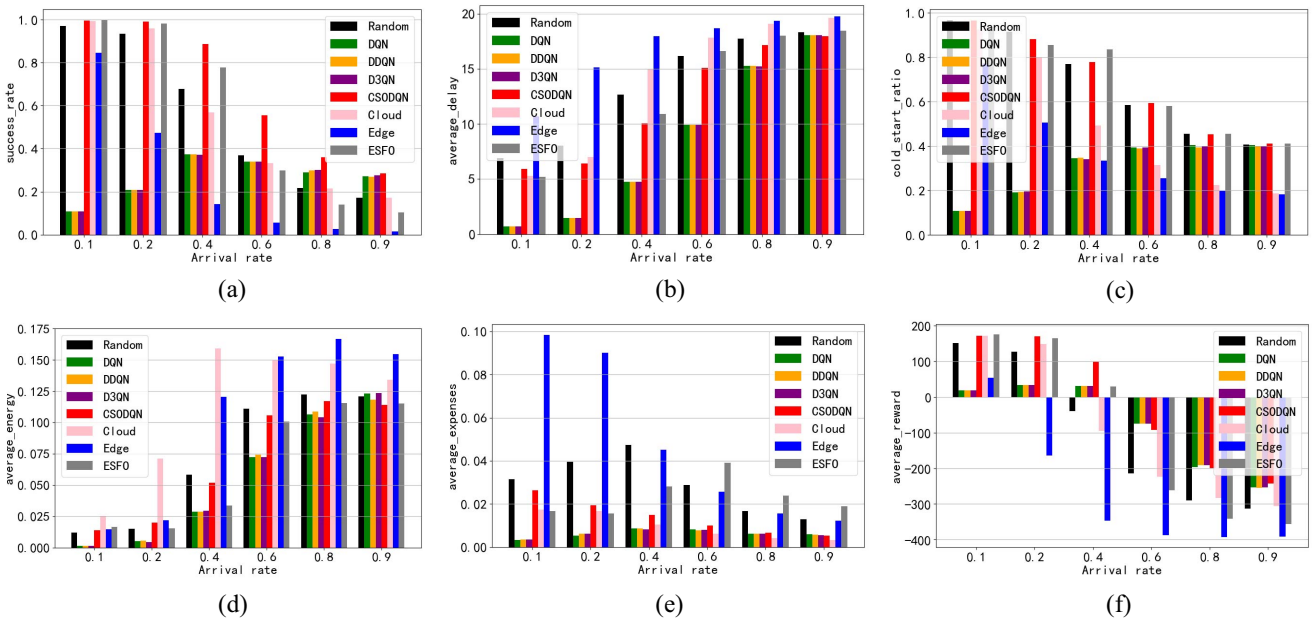


Fig. 6. Performance comparison of different offloading methods with different task arrival rates. (a) Success rate. (b) Average task delay. (c) Cold start ratio. (d) Average task energy consumption. (e) Average dollar of task spent. (f) Average reward of task computing.

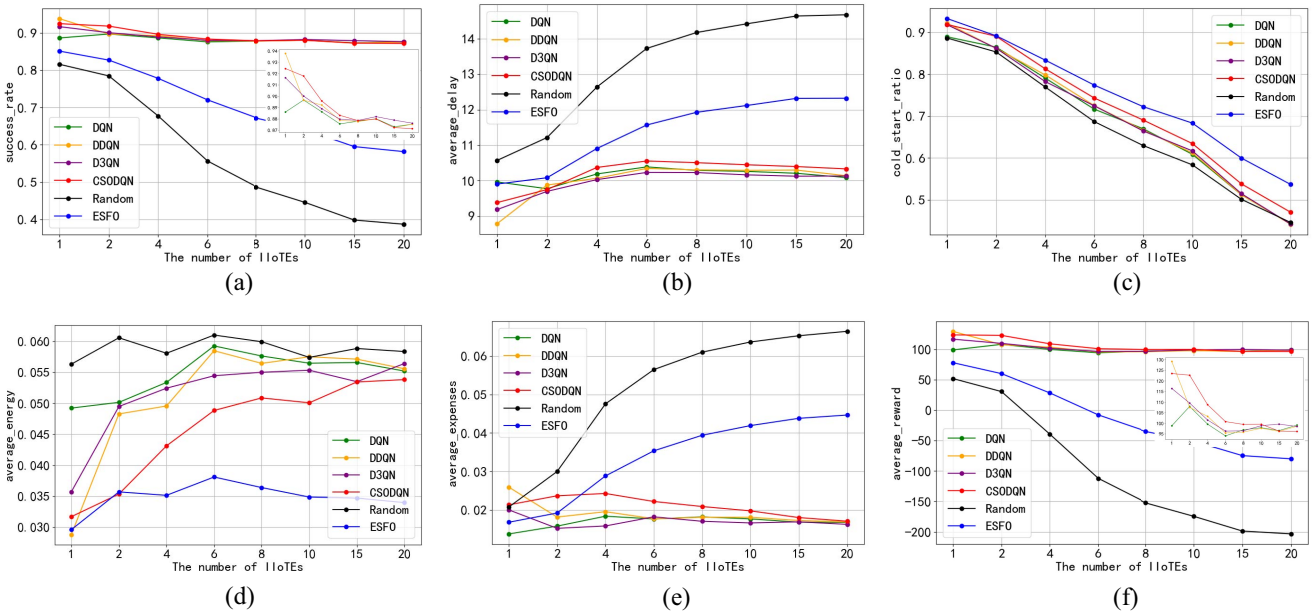


Fig. 7. Performance comparison of different offloading methods with different number of IIoTEs. (a) Success rate. (b) Average task delay. (c) Cold start ratio. (d) Average task energy consumption. (e) Average dollar of task spent. (f) Average reward of task computing.

as the survival time increases, more function requests occur within the warm state of the corresponding function, allowing the function to be instantiated and executed via a warm start. As a result, the probability of cold starts in the system significantly decreases. However, excessively long survival times consume system storage resources, which contradicts the service philosophy of serverless computing. The task success rate of ESFO fluctuates between 75% and 80%, while the success rate of Random remains below 70%. In contrast, reinforcement-learning-based algorithms can maintain a success rate above 87%. CSODQN achieves the highest overall success rate and reward, and its device energy consumption is second only to that of ESFO, effectively reducing system resource consumption while ensuring QoS.

5) Performance Comparison With Different Maximum Tolerable Delay: As shown in Fig. 9, with the increase in the maximum tolerable delay for tasks, the task success rate in the system improves significantly. According to the design of the reward function, the proportion of cold start time within the maximum tolerable delay decreases, meaning that the reward gain from transitioning tasks from cold start to warm start is reduced. As a result, the system places greater emphasis on minimizing reward loss through energy consumption optimization. Although edge servers have higher

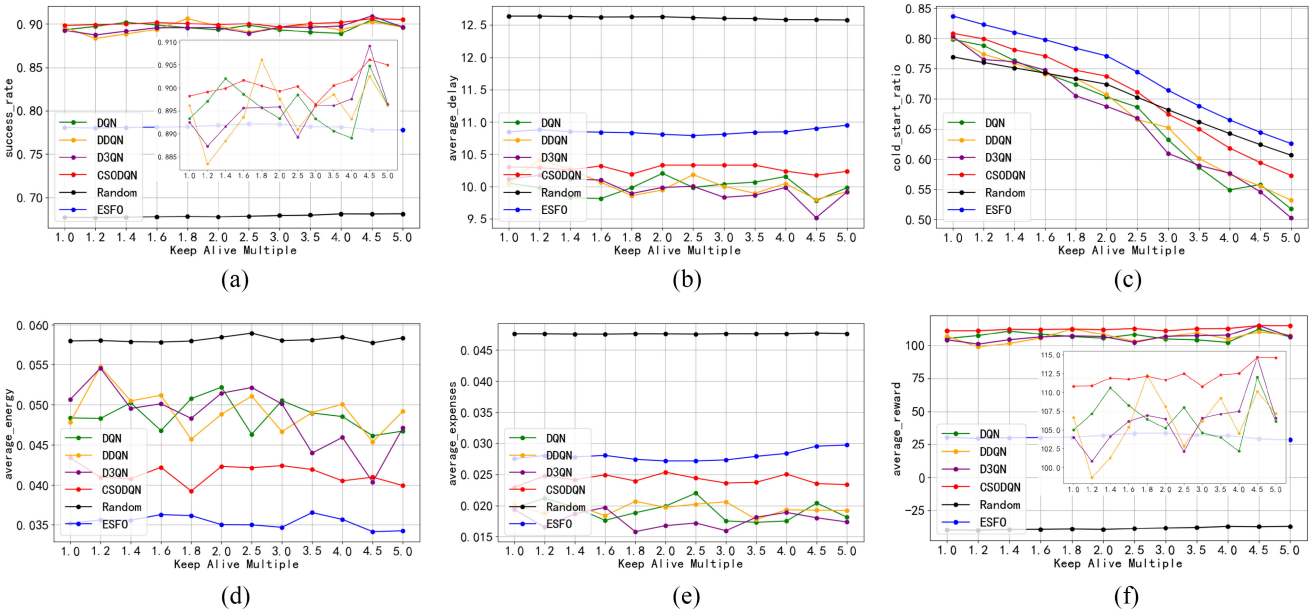


Fig. 8. Performance comparison of different offloading methods with different times of SF instance keep alive after computing. (a) Success rate. (b) Average task delay. (c) Cold start ratio. (d) Average task energy consumption. (e) Average dollar of task spent. (f) Average reward of task computing.

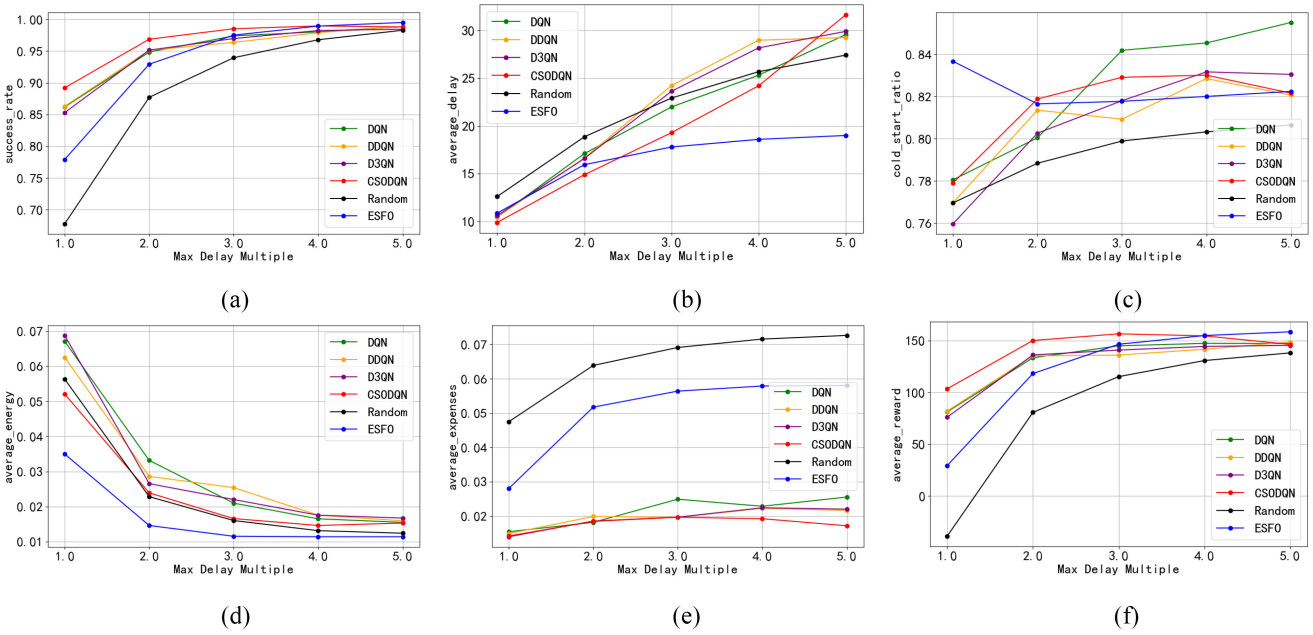


Fig. 9. Performance comparison of different offloading methods with different maximum tolerable delay. (a) Success rate. (b) Average task delay. (c) Cold start ratio. (d) Average task energy consumption. (e) Average dollar of task spent. (f) Average reward of task computing.

computational costs and lower computing capabilities compared to cloud data centers, the increased tolerable delay allows more SF requests to maintain service quality at the edge. The average dollar of task spent for Random and ESFO methods is four to five times higher than that of reinforcement-learning-based approaches, with CSODQN maintaining the lowest cost and achieving the highest reward. Although the average task processing time increases, it does not impact the overall service quality of the system. Overall, as the maximum tolerable delay increases, the task success rate of all methods remains above 90%, and CSODQN demonstrates a better balance between service quality and system cost, outperforming other algorithms.

VI. Conclusion

To reduce the frequency of cold starts and efficiently utilize resources in a distributed SEC environment, this work establishes a multilayer SEC system model that involves cloud servers, edge nodes, and end IoT devices. It systematically investigates the task offloading problem and cold start issue of SF execution in SEC. A vertical offloading mechanism and SF

warm instance pools are introduced, along with distinct cold start models for different computing nodes. Each SF request is treated as the basic unit for offloading, with offloading strategies applied individually. By optimizing the reward function, the multiuser, multitype SF offloading problem is transformed into a multiobjective optimization task considering cold start frequency, delay, energy consumption, and cost. cold-start-aware distributed offloading method (CSODQN) based on DRL is proposed. To enhance training diversity and prevent overfitting from biased high-priority samples, the original sampling method is replaced with an independent action-based strategy, accelerating DRL convergence. Under task delay and resource constraints, the method seeks optimal offloading decisions that minimize cold starts, maximize success rate, and reduce energy and cost. Experimental results show that under a high-concurrency task arrival rate of 0.9, compared to the random offloading strategy, the task success rate increases by 70%, the warm start probability increases by 30%, the average monetary costs decreases by 58%, the average energy consumption increases by 14.07%, and the average delay decreases by 1.51%.

REFERENCES

[1] E. Jonas et al., "Cloud programming simplified: A berkeley view on serverless computing," 2019, *arXiv:1902.03383*.

[2] K. Zhao, Z. Zhou, L. Jiao, S. Cai, F. Xu, and X. Chen, "Taming serverless cold start of cloud model inference with edge computing," *IEEE Trans. Mobile Comput.*, vol. 23, no. 8, pp. 8111–8128, Aug. 2024.

[3] J. Manner, M. Endreß, T. Heckel, and G. Wirtz., "Cold start influencing factors in function as a service," in *Proc. UCC Compan.*, 2018, pp. 181–188.

[4] G. R. Russo, D. Ferrarelli, D. Pasquali, V. Cardellini, and F. L. Presti., "QoS-aware offloading policies for serverless functions in the cloud-to-edge continuum," *Future Gener. Comput. Syst.*, vol. 156, pp. 1–15, Jul. 2024.

[5] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX ATC*, 2018, pp. 133–146.

[6] Z. M. Nayeri, T. Ghafarian, and B. Javadi, "Application placement in fog computing with AI approach: Taxonomy and a state of the art survey," *JNCA*, vol. 185, Jul. 2021, Art. no. 103078.

[7] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proc. ASPLOS*, 2022, pp. 753–767.

[8] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. NSDI*, 2020, pp. 419–434.

[9] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX ATC*, 2020, pp. 205–218.

[10] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. ASPLOS*, 2021, pp. 386–400.

[11] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: A reinforcement learning approach," *IEEE Internet Things J.*, vol. 10, no. 5, pp. 3917–3927, Apr. 2022.

[12] Z. Sun et al., "Cloud-edge collaboration in Industrial Internet of Things: A joint offloading scheme based on resource prediction," *IEEE Internet Things J.*, vol. 9, no. 18, pp. 17014–17025, Sep. 2022.

[13] H. Lin, L. Yang, H. Guo, and J. Cao, "Decentralized task offloading in edge computing: An Offline-to-online reinforcement learning approach," *IEEE Trans. Comput.*, vol. 73, no. 6, pp. 1603–1615, Jun. 2024.

[14] L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu, and L. Li, "Delay-aware and energy-efficient computation offloading in mobile-edge computing using deep reinforcement learning," *IEEE Trans. Cogn. Commun. Netw.*, vol. 7, no. 3, pp. 881–892, Sep. 2021.

[15] Q. Tang et al., "Distributed task scheduling in serverless edge computing networks for the Internet of Things: A learning approach," *IEEE Internet Things J.*, vol. 9, no. 20, pp. 19634–19648, Oct. 2022.

[16] C. Chen, M. Herrera, G. Zheng, L. Xia, Z. Ling, and J. Wang, "Cross-edge orchestration of Serverless functions with probabilistic caching," *IEEE Trans. Services Comput.*, vol. 17, no. 5, pp. 2139–2150, Sep./Oct. 2024.

[17] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey," *J. Netw. Comput. Appl.*, vol. 181, May 2021, Art. no. 103005.

[18] W. Qin, H. Chen, and L. Wang, "PASD: A Prioritized action sampling-based Dueling DQN for cloud-edge collaborative computation offloading in industrial IoT," in *Proc. China Conf. Wireless Sensor Netw.*, 2022, pp. 19–30.

[19] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with WebAssembly runtimes," in *Proc. CCGrid*, 2022, pp. 140–149.

[20] A. Wang et al., "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute," in *Proc. USENIX ATC*, 2021, pp. 443–457.

[21] M. Golec et al., "Master: Machine learning-based cold start latency prediction framework in serverless edge computing environments for industry 4.0," *IEEE J. Sel. Areas Sensors*, vol. 1, pp. 36–48, 2024.

[22] C. K. Dehury, S. Poojara, and S. N. Srirama, "Def-DReL: Towards a sustainable serverless functions deployment strategy for fog-cloud environments using deep reinforcement learning," *Appl. Soft Comput.*, vol. 152, Feb. 2024, Art. no. 111179.

[23] C. Qiao, C. Wang, Z. Zhang, Y. Ji, and X. Gao, "Caching aided multi-tenant Serverless computing," 2024, *arXiv:2408.00957*.

[24] Y. H. Chiang, C. Zhu, H. Lin, andY. Ji, "Hysteretic optimality of container warming control in serverless computing systems," *IEEE Netw. Lett.*, vol. 3, no. 3, pp. 138–141, Sep. 2021.

[25] A. U. Gias and G. Casale, "Cocoa: Cold start aware capacity planning for function-as-a-service platforms," in *Proc. MASCOTS*, 2020, pp. 1–8, doi: 10.1109/MASCOTS50786.2020.9285966.

[26] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López., "On the faas track: Building stateful distributed applications with serverless architectures," in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 41–54.

[27] T. Pfandzelter and D. Bermbach., "tinyfaas: A lightweight faas platform for edge environments," in *Proc. IEEE ICFC*, 2020, pp. 17–24.

[28] G. Sadeghian, M. Elsakhawy, M. Shahrad, J. Hattori, and M. Shahrad, "UnFaaSener: Latency and cost aware offloading of functions from Serverless platforms," in *Proc. USENIX ATCProc.*, 2023, pp. 879–896.

[29] M. Pandey and Y. W. Kwon, "FuncMem: Reducing cold start latency in Serverless computing through memory prediction and adaptive task execution," in *Proc. 39th ACM/SIGAPP Symp. Appl. Comput.*, 2024, pp. 131–138.

[30] H. Lu, X. He, M. Du, X. Ruan, Y. Sun, and K. Wang, "Edge QoE: Computation offloading with deep reinforcement learning for Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9255–9265, Mar. 2020.

[31] G. R. Russo, V. Cardellini, and F. L. Presti, "A framework for offloading and migration of serverless functions in the edge–cloud continuum," *Pervasive Mobile Comput.*, vol. 100, May 2024, Art. no. 101915.

[32] X. Yao, N. Chen, X. Yuan, and P. Ou, "Performance optimization of serverless edge computing function offloading based on deep reinforcement learning," *Future Gener. Comput. Syst.*, vol. 139, pp. 74–86, Feb. 2023.

[33] A. Mampage, S. Karunasekera, and R. Buyya, "Deadline-aware dynamic resource management in serverless computing environments," in *Proc. CCGrid*, 2021, pp. 483–492.

[34] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 381–397.

[35] Q. Tang, R. Xie, F. R. Yu, T. Huang, and Y. Liu, "Decentralized computation offloading in IoT fog computing system with energy harvesting: A dec-POMDP approach," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 4898–4911, Feb. 2020.

[36] X. Chen and G. Liu, "Energy-efficient task offloading and resource allocation via deep reinforcement learning for augmented reality in mobile edge networks," *IEEE Internet Things J.*, vol. 8, no. 13, pp. 10843–10856, Jan. 2021.

[37] Z. Xu et al., "Stateful serverless application placement in MEC with function and state dependencies," *IEEE Trans. Comput.*, vol. 72, no. 9, pp. 2701–2716, Mar. 2023.

[38] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *Proc. ACSOS*, 2020, pp. 1–10.

[39] C. Xu et al., "FaaSMem: Improving memory efficiency of serverless computing with memory pool architecture," in *Proc. ASPLOS*, 2024, pp. 331–348.
[40] J. Lou et al., "Efficient serverless function scheduling in edge computing," in *Proc. ICC*, 2024, pp. 1029–1034.
[41] G. W. Kim, S. J. Moon, and B. J. Park, "A study on cold start and resource improvement using time warming allocation engine in Serverless computing," *Int. J. Adv. Smart Converg.*, vol. 13, no. 3, pp. 109–116, 2024.
[42] X. Li, L. Chen, Z. Yuan, and G. Liu. "Aiho: Enhancing task offloading and reducing latency in Serverless multi-edge-to-cloud systems." SSRN.com. 2024. [Online] Available: http://dx.doi.org/10.2139/ssrn.4740475
[43] Y. Wang, X. Yan, B. Li, and S. Xu., "Measuring the variation of function invocation for cold start optimization in Serverless environments," in *Proc. 11th Int. Conf. CBD*, 2023, pp. 217–222.
[44] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proc. CCGrid*, 2021, pp. 797–803.
[45] S. Poojara, C. K. Dehury, P. Jakovits, and S. N. Srirama, "Serverless data pipelines for IoT data analytics: A cloud vendors perspective and solutions," in *Predictive Analytics in Cloud, Fog, and Edge Computing: Perspectives and Practices of Blockchain, IoT, and 5G*. Cham, Switzerland: Springer, 2022, pp. 107–132.
[46] S. P. Mirampalli, R. Wankar, and S. N. Srirama., "Evaluating NiFi and MQTT based serverless data pipelines in fog computing environments," *Future Gener. Comput. Syst.*, vol. 150, pp. 341–353, Jan. 2024.
[47] I. Pelle, J. Czentye, J. Dóka, A. Kern, B. P. Gerő, and B. Sonkoly, "Operating latency sensitive applications on public serverless edge cloud platforms," *IEEE Internet Things J.*, vol. 8, no. 10, pp. 7954–7972, May 2021.

Peihao Wu received the B.S. degree in computer science and technology from Changchun University of Technology, Changchun, China, in 2020. He is currently pursuing the master's degree in computer science and Technology with the Department of Computer Science, Ningbo University, Ningbo, China.

His research interests include edge computing and Artificial Intelligence of Things systems.

Haiming Chen (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Tianjin University, Tianjin, China, in 2003 and 2006, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2010.

He is currently an Associate Professor with the Department of Computer Science, Ningbo University, Ningbo, China. From 2010 to 2017, he was a Research Assistant Professor with the Institute of Computing Technology, Chinese Academy of Sciences. He also was a Visiting Scholar from October 2015 for one year with the Department of Computer Science, College of William and Mary, Williamsburg, VA, USA. His research interests include Internet of Things and edge–cloud collaborative computing systems.

He is a Senior Member of the CCF.

Tianhao Wu received the B.S. degree in software engineering from Jiangxi University of Science and Technology, Ganzhou, China, in 2022. He is currently pursuing the master's degree with the Department of Computer Science, Ningbo University, Ningbo, China.

His research interests include edge computing and intelligent system.

Kaiqi Gu is currently pursuing the bachelor's degree with the Department of Computer Science, Ningbo University, Ningbo, China.

His research interests include edge AI and AIoT systems.

Yinshui Xia (Member, IEEE) received the B.S. degree in physics and the M.S. degree in electronic engineering from Zhejiang University, Hangzhou, China, in 1984 and 1991, respectively, and the Ph.D. degree in electronic engineering from Edinburgh Napier University, Edinburgh, U.K., in 2003.

He was a Visiting Scholar with King's College London, London, U.K., in 1999, and then joined Edinburgh Napier University as a Research Assistant and an Enterprise Fellow from 2000 to 2005. He is currently a Professor with the Faculty of Electrical Engineering and Computer Science, Ningbo University, Ningbo, China. His research interests include low-power digital circuit design, logic synthesis and optimization, and system-on-chip design.