







Multi-Objective Deep Reinforcement Learning for Function Offloading in Serverless Edge Computing

Yaning Yang , *Student Member, IEEE*, Xiao Du, *Student Member, IEEE*, Yutong Ye , *Student Member, IEEE*, Jiepin Ding , Ting Wang , *Senior Member, IEEE*, Mingsong Chen , *Senior Member, IEEE*, and Keqin Li , *Fellow, IEEE*

Abstract—Function offloading problems play a crucial role in optimizing the performance of applications in serverless edge computing (SEC). Existing research has extensively explored function offloading strategies based on optimizing a single objective. However, a significant challenge arises when users expect to optimize multiple objectives according to the relative importance of these objectives. This challenge becomes particularly pronounced when the relative importance of the objectives dynamically shifts. Consequently, there is an urgent need for research into multi-objective function offloading methods. In this paper, we redefine the SEC function offloading problem as a dynamic multi-objective optimization issue and propose a novel approach based on Multi-objective Reinforcement Learning (MORL) called MOSEC. MOSEC can coordinately optimize three objectives, i.e., application completion time, User Device (UD) energy consumption, and user cost. To reduce the impact of extrapolation errors, MOSEC integrates a Near-on Experience Replay (NER) strategy during the model training. Furthermore, MOSEC adopts our proposed Earliest First (EF) scheme to maintain the policies learned previously, which can efficiently mitigate the catastrophic policy forgetting problem. Extensive experiments conducted on various generated applications demonstrate the superiority of MOSEC over state-of-the-art multi-objective optimization algorithms.

Index Terms—Serverless edge computing, function offloading, multi-objective optimization, deep reinforcement learning.

I. INTRODUCTION

SERVERLESS computing is an emerging cloud computing framework in which applications are constructed from fine-grained functions, known as Function as a Service (FaaS) [1],

Received 14 May 2024; revised 16 September 2024; accepted 23 October 2024. Date of publication 31 October 2024; date of current version 6 February 2025. This work was supported in part by the Natural Science Foundation of China under Grant 62272170, and in part by Shanghai Trusted Industry Internet Software Collaborative Innovation Center. (*Corresponding authors: Mingsong Chen; Ting Wang.*)

Yaning Yang is with the MoE Engineering Research Center of Hardware/Software Co-design Technology and Application, East China Normal University, Shanghai 200062, China, and also with the School of Physics and Electronic Information Engineering, Ningxia Normal University, Guyuan 756500, China.

Xiao Du, Yutong Ye, Jiepin Ding, Ting Wang, and Mingsong Chen are with the MoE Engineering Research Center of Hardware/Software Co-design Technology and Application, East China Normal University, Shanghai 200062, China (e-mail: twang@sei.ecnu.edu.cn; mschen@sei.ecnu.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA.

This article has supplementary downloadable material available at <https://doi.org/10.1109/TSC.2024.3489443>, provided by the authors.

Digital Object Identifier 10.1109/TSC.2024.3489443

[2], [3]. Within this model, developers focus solely on coding and deploying functional components, whereas managing the underlying hardware and servers falls upon cloud service providers. Concurrently, resource consumption can dynamically scale with the demands of applications. As such, users incur costs for the resources they actually consume rather than for continuous server operation. Prominent platforms for serverless computing include AWS Lambda [4], Google Cloud Functions [5], and Azure Functions [6].

By extending the serverless computing model to the edge of the network, a Serverless Edge Computing (SEC) framework has been proposed. Due to the capability of offering low-latency and cost-effective services, this framework has gained significant attention from both industry and academia [7], [8], [9]. Extensive research has been conducted on SEC architecture [10], application deployment [11], and function offloading [12]. Given the pivotal role of function offloading in enhancing application performance, optimizing function offloading strategies has become a critical research focus in this field.

Compared to serverless computing environments hosted solely in the cloud, SEC is characterized by a heterogeneous collaboration between edge servers and cloud centers. Application functions can be offloaded to the cloud center or heterogeneous edge servers for execution. Different offloading strategies can lead to different performances, including application completion time, User Device (UD) energy consumption, and user costs. Consequently, SEC can provide optimal function offloading strategies tailored to different optimization objectives. By focusing on user optimization needs and improving user experience satisfaction, service providers can increase the number of user requests, thereby increasing service provider revenue. However, in SEC platforms, the limited resources of edge servers prevent the deployment of all application functions and the configuration of execution environments for each function, thereby restricting which edge servers can execute the offloaded functions. Additionally, when utilizing the available resources of edge servers to execute functions, the cold start latency of function containers must be considered. These factors collectively make function offloading optimization in SEC environments significantly more complex.

To develop the optimal offloading strategy, extensive research has investigated function offloading methods in SEC [13], [14], [15], [16]. Most focus on optimizing a single performance objective, such as minimizing energy consumption or

application completion time. However, users often need to balance multiple objectives, such as completion time, UD energy consumption, and user costs, based on objectives preferences, which may change dynamically. For instance, when processing latency-sensitive applications, users may prioritize minimizing completion time. Conversely, in budget-constrained scenarios, they may focus on reducing energy consumption while also considering delays and user costs. In such cases, single-objective optimization methods are insufficient, underscoring the need for multi-objective function offloading approaches.

Multi-objective optimization algorithms are well-suited for decision-making scenarios involving multiple conflicting objectives [17], [18], [19]. Recently, Multi-objective Reinforcement Learning (MORL) algorithms have gained significant research interest based on the effectiveness of Deep Reinforcement Learning (DRL) algorithms in dynamic environments [20], [21], [22]. MORL has been widely used in various fields, including communication [23], [24], electric power [25], and medical treatment [26]. Furthermore, MORL-based algorithms have been extended to task offloading in edge computing [28]. However, traditional edge computing approaches cannot be directly applied to SEC due to additional complexities, such as function deployment constraints and cold start latencies. These factors make SEC function offloading optimization more intricate, highlighting the urgent need for research in multi-objective function offloading tailored to SEC environments.

In this paper, we employ a multi-objective optimization approach to address the SEC function offloading problem. We model the application as a Directed Acyclic Graph (DAG) and focus on optimizing application completion time, UD energy consumption, and user costs. The relative importance of these objectives is captured through dynamic preference vectors. We aim to devise optimal function offloading strategies that can dynamically adapt to varying preferences among the three objectives. The primary contributions of this paper are summarized as follows:

- We formalize the SEC function offloading problem as a multi-objective optimization problem, which optimizes application completion time, UD energy consumption, and user cost based on the dynamic preference vectors of three objectives.
- To address the above problem, we propose a novel MORL-based method named MOSEC, which incorporates a Near-on Experience Replay (NER) to suppress the impact of extrapolation errors. Furthermore, MOSEC adopts our proposed Earliest First (EF) scheme to maintain previously learned policies, efficiently mitigating the policy forgetting problem.
- We conduct extensive experiments on a collection of artificially generated test applications with diverse topologies and task profiles. The experimental results show that our MOSEC approach outperforms state-of-the-art multi-objective optimization algorithms.

The rest of this paper is organized as follows. Section II presents a comprehensive overview of the related work. A formal definition of the SEC scenario and multi-objective optimization problem is presented in Section III. Section IV describes our

proposed approach in detail. In Section V, we present the results of our experimental study. Section VI discusses the advantages and limitations of this work, as well as future work. We conclude the paper in Section VII.

II. RELATE WORK

This section provides a comprehensive overview of function offloading in SEC, in which both the independent and dependent function offloading approaches are investigated. Furthermore, we delve into multi-objective optimization for task offloading in edge computing.

A. Function Offloading in SEC

1) *Function Offloading for Independent Functions*: Various methods have been proposed to investigate the independent functions offloading in SEC. For instance, Anirban et al. [15] proposed a dynamic function placement framework to optimize the completion time of an application under constraints of cost and deadline. Aslanpour et al. [13] presented an energy-aware function offloading algorithm with an SEC prototype to reduce energy consumption on the edge nodes. Additionally, Tang et al. [29] designed a multi-agent function offloading approach based on a stochastic game to obtain the optimal offloading scheme for a given objective. However, these methods mainly perform to obtain optimal offloading strategies for independent functions. At the same time, they focus on optimizing a single objective, disregarding the complex dependencies among functions, and optimizing multiple objectives. Hence, they are unsuitable for more complex applications and scenarios for optimizing multiple objectives.

2) *Function Offloading for Dependent Functions*: Due to the interdependencies among functions in complex serverless applications, the function offloading problem becomes more challenging. Numerous methods have been proposed to address the optimal offloading problem in complex applications. For example, Liu et al. [30] proposed GenDoc, an approximate function placement algorithm that minimizes the completion time of applications in edge clouds. Deng et al. [31] presented a dependent function embedding approach to determine the optimal starting time and execution location for each function. Li et al. [12] developed a PASS algorithm to minimize completion time while considering function assignment and communication mode. Zheng et al. [32] introduced a multi-level progressive optimization approach that minimizes the completion time during the function offloading process. Xie et al. [14] introduced a PSO-PA algorithm, which maximizes user utility by scalarizing application completion time and energy consumption. However, these methods focus on optimizing single objectives in solving function offloading problems, resulting in inadequate performance in optimizing multiple objectives, especially when the preference vectors of multiple objectives change dynamically.

B. Multi-Objective Optimization for Task Offloading

Multi-objective optimization algorithms are broadly used in task offloading for edge computing networks. For example, Liu

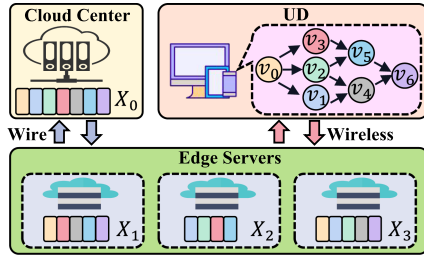


Fig. 1. System architecture of SEC.

et al. [25] proposed a multi-objective offloading algorithm to minimize the average latency and energy consumption of task offloading in edge cloud environments. In [26], Almasri et al. considered task dependencies and data distribution in mobile edge computing, proposing a multi-objective evolutionary algorithm constrained by deadlines and power consumption. These multi-objective methods independently optimize completion time and energy consumption without considering the balance between the two conflicting objectives. Pan et al. [34] proposed the CHCE algorithm to minimize execution costs and energy consumption for multiple workflows under deadline constraints. Although this method optimizes execution costs and energy consumption simultaneously, the dynamics of optimization objectives preference is not considered.

Some existing works in edge computing have optimized multiple conflicting objectives simultaneously in task offloading and balanced the objectives through dynamic preferences. Song et al. [28] proposed a modified MORL approach to optimize the application completion time, energy consumption, and usage cost in a mobile edge computing environment. Based on [28], Liu et al. [39] extended the MORL approach to UAV-assisted mobile edge computing to optimize the application completion time and energy consumption during tasks offloading process. Furthermore, Yang et al. [40] utilized the MORL approach to solve the multi-task offloading problem in edge computing. While these methods effectively balance multiple conflicting objectives in edge computing through dynamic preferences, the challenges of function deployment and cold start latency in constrained resources SEC add significant complexity to the multi-objective function offloading problem. Therefore, it is necessary to study the problem of multi-objective function offloading for SEC scenarios.

This paper proposes a novel MOSEC approach to address the function offloading problem for SEC optimizing application completion time, UD energy consumption, and user costs, in which the dynamic preference of objectives is dynamic change. To our knowledge, this work is the first attempt to solve function offloading problems in SEC using a MORL-based method.

III. SYSTEM MODEL AND PROBLEM FORMULATION

As illustrated in Fig. 1, our SEC architecture is composed of a cloud center, multiple edge servers, and a UD. The cloud center is far from UD and has substantial memory and computing resources. In contrast, the edge servers are positioned close to UD but have limited memory and computing resources. The

cloud center and edge servers collaborate to offer services for application requests from UD. A serverless application comprises multiple functions that exhibit inter-dependencies. Each function is an indivisible task. To execute a function instance, the server must pre-deploy the corresponding function to create a function execution environment. In other words, only servers with pre-deployed functions are capable of executing the respective function instances. Typically, the cloud center has abundant resources to pre-deploy all functions of a serverless application. However, the edge servers can only pre-deploy a subset of the functions due to limited resources.

The requests generated by UD are transmitted through wireless uplinks to the edge servers. If an edge server lacks resources or undeveloped a specific function, it will be unable to execute the function, and we set the execution time to infinity. Once the application is completed, the results are transmitted back to UD through the downlinks. We assume that the volume of execution results is small and that there is sufficient downlink bandwidth for transmitting the results. Hence, the return time can be neglected.

A. Serverless Application Model

A serverless application is represented by a DAG and denoted by $G = (V, E)$, in which V represents a set of stateless functions, denoted by $V = \{v_0, v_1, \dots, v_{N-1}\}$, and E is a set of dependency constraints between the functions. Each element $e_{ij} = (v_i, v_j)$ in E represents a immediate dependency relationship between v_i and v_j . Specifically, v_i is an immediate predecessor of v_j , and v_j is an immediate successor of v_i . $pred(v_i)$ and $succ(v_i)$ denote the sets of immediate predecessors and immediate successors of v_i , respectively. Before offloading v_i , all functions in $pred(v_i)$ must complete their execution. As indicated in Fig. 1, $pred(v_4) = \{v_1, v_2\}$, and $succ(v_4) = \{v_6\}$. Therefore, v_4 can only be offloaded after both v_1 and v_2 have been completed. Functions with no immediate predecessor are entry functions denoted by v_{entry} . Similarly, functions with no immediate successor are exit functions, denoted by v_{exit} . As depicted in Fig. 1, v_{entry} and v_{exit} are $\{v_0\}$ and $\{v_6\}$, respectively.

Each function is defined by a tuple $v_i = \langle d_i, q_i, mem_i, cw_i \rangle$, where d_i and q_i represent the input data size and output data size of v_i , respectively. Additionally, mem_i and cw_i represent the memory size and CPU cycles required to execute v_i , respectively. The input data of the function is generated from UD and transmitted to an edge server via a wireless link. Once the v_i is completed, the output data is immediately forwarded to the servers where the subsequent functions will be executed.

B. Server Model

As shown in Fig. 1, the servers in our SEC system include one cloud server denoted by X_0 and a set of edge servers characterized by $\{X_1, X_2, \dots, X_M\}$. The servers set is denoted by $X = \{X_0\} \cup \{X_1, X_2, \dots, X_M\}$. Assuming that the servers are heterogeneous and equipped with multiple CPU cores, we define the clock frequency of each server as $\omega = \{\omega_0, \omega_1, \dots, \omega_M\}$.

Each server operates as a serverless platform with a group of containers serving as units for function execution. Each container is equipped with a certain number of memory resources, and its computing resources are proportional to the number of memory resources that it possesses [35]. After a container completes a function instance, it enters a no-load state for a period, ready to execute the following offloaded function. If no function is offloaded to the container during the no-load period, its resources will be recycled. We denote the functions deployed across all servers as $F = \{F_0, F_1, \dots, F_M\}$, where $F_m \subseteq V$, $m \in [0, M]$. Here, $F_0 = V$ indicates that all functions in G are deployed on the cloud server, and $F_m \subset V$, $m \neq 0$ implies that an edge server deploys partial functions in G .

The containers on edge servers have three states, i.e., occupied, idle, and available. Occupied containers represent resources that are currently executing a function. Idle containers are resources that are in a no-load state. Available containers are recycled resources that can be restarted to execute a new function. We use $D_i^m = 1$ to indicate the presence of idle containers for v_i on the server X_m , and $D_i^m = 0$ represents the absence of idle containers. Furthermore, $R_i^m = 1$ indicates adequate available resources to execute v_i on the server X_m , while $R_i^m = 0$ indicates inadequate available resources. When v_i is offloaded to X_m and $D_i^m = 1$, v_i can be executed immediately; otherwise, $D_i^m = 0$ and $R_i^m = 1$, an available container on X_m will be restarted to execute the function, resulting in a cold start time T_c on X_m .

C. Problem Formulation

SEC can offer diverse offloading strategies for applications, resulting in different completion times, UD energy consumption, and user costs. For instance, offloading functions to the cloud center increases transmission latency, leading to longer completion time and higher UD energy consumption, although it may reduce user costs. In contrast, offloading functions to edge servers reduces transmission latency and UD energy consumption but often results in higher user costs. Additionally, if the offloaded edge server takes available resources to execute the function, starting a new container will lead to cold start latency, thereby increasing the completion time. Conversely, if the edge server possesses an idle container to execute the function, it significantly reduces function completion time.

Our MOSEC framework seeks to optimize application completion time, UD energy consumption, and user costs during the functions offloading process. The service provider considers these three objectives from the user's perspective to improve the user experience. Improving satisfaction can lead to an increase in user requests, thereby boosting the revenue of service providers. However, these optimization objectives are interrelated and often conflicting, necessitating a careful balance. The dynamic nature of their relative importance is captured through preference vectors.

We denote application completion time, UD energy consumption, and user costs as T , E , and C , respectively. Let \mathbf{w} represent a preference vector, where $\mathbf{w} = \langle w^T, w^E, w^C \rangle$ reflects the relative importance of these three objectives. Each

element in preference vector ranges in $[0, 1]$ and satisfies the constraint $\sum_{i \in (T, E, C)} w^i = 1$. The optimization problem can be formulated as a multi-objective optimization problem, which is defined as follows

$$\begin{aligned} & \text{Optimize}_{l_i, \mathbf{w}} : (T, E, C) \\ & \text{Subject to :} \\ & C1 : l_i \in \{0, 1, \dots, M\}, \forall i \in \{0, 1, \dots, N-1\}, \\ & C2 : w^i \in [0, 1], \sum w^i = 1, w^i \in \mathbf{w}, i \in (T, E, C), \\ & C3 : FT_j \leq ST_i, \forall v_i \in V, v_j \in \text{pred}(v_i). \end{aligned} \quad (1)$$

Constraint $C1$ presents the function offloading strategies. $C2$ indicates the constraints of preferences for multiple objectives. Constraint $C3$ denotes the dependency relationship among functions, specifying that function v_i can only start after all of its immediate predecessor functions have been completed.

D. Completion Time Model

The completion time of a serverless application represents the total latency incurred in function execution and data transmission. It is calculated by the time difference between the start of the transmission of input data for v_{entry} and the completion of v_{exit} . To calculate T , we need to iteratively consider the processing time of each function.

1) *Transmission Time*: Assume that l_i represents the offloading strategy for the function v_i . If v_i is offloaded to the server X_m , l_i equals m . We define T_{i_t} as the transmission time of v_i , which consists of the transmission time for input data from UD and the maximum transmission time for output data from all functions in $\text{pred}(v_i)$. The transmission time for input and output data are denoted by T_{i_in} and T_{i_pred} , respectively. We calculate T_{i_in} as

$$T_{i_in} = \begin{cases} \frac{d_i}{r_w} & l_i > 0, \\ t_{ec} & l_i = 0, \end{cases} \quad (2)$$

where d_i represents the input data size of v_i , and r_w is the transmission rate between UD and the edge server. According to [16], we denote r_w as

$$r_w = B_w \log_2(1 + SNR), \quad (3)$$

where B_w and SNR are the bandwidth and Signal Noise Ratio of the wireless channel, respectively. Since UD and the cloud center communicate through the public network, it is influenced by multiple factors, such as link quality, path variations, server load, network congestion, etc., making it challenging to accurately estimate transmission time. Furthermore, the offloading strategy does not impact the transmission latency between the UD and the cloud. To simplify the model, we assume a fixed transmission time of t_{ec} from the UD to the cloud center.

Assume that the edge servers are connected by stable and reliable links. The transmission rate of the links is fixed and

defined by r_e . Thus, we define T_{i_pred} as

$$T_{i_pred} = \begin{cases} t_{ec} & l_j \times l_i = 0, l_j \neq l_i, \\ \max(\frac{q_j}{r_e}) & l_j \times l_i \neq 0, l_j \neq l_i, \\ 0 & \text{others,} \end{cases} \quad (4)$$

where l_j and q_j are the offloading strategy and the output data size of v_j , respectively, $v_j \in pred(v_i)$. Hence, T_{i_t} is calculated as

$$T_{i_t} = \max(T_{i_in}, T_{i_pred}). \quad (5)$$

2) *Execution Time*: We define T_{i_e} as the execution time of v_i , and it can be calculated as

$$T_{i_e} = \begin{cases} T_c + cw_i \times \omega_m & l_i \neq 0, v_i \in F_m, D_i^m = 0, R_i^m = 1, \\ cw_i \times \omega_m & (l_i = 0) \text{ or } (l_i \neq 0, v_i \in F_m, D_i^m = 1), \\ +\infty & \text{others,} \end{cases} \quad (6)$$

3) *Completion Time*: Based on the above description, we denote T_i as the time spent on processing v_i . T_i is defined as

$$T_i = T_{i_t} + T_{i_e}. \quad (7)$$

Assume that ST_i represents the earliest start time, and FT_i denotes the latest finish time of function v_i . We define ST_i as

$$ST_i = \begin{cases} 0 & v_i = v_{entry}, \\ \max_{v_j \in pred(v_i)} \{FT_j\} & v_i \neq v_{entry}. \end{cases} \quad (8)$$

The FT_i is denoted as

$$FT_i = ST_i + T_i. \quad (9)$$

Therefore, the completion time of the whole serverless application can be defined as

$$T = \max\{FT_i\}, v_i \in v_{exit}. \quad (10)$$

E. UD Energy Consumption Model

The total UD energy consumption encompasses the energy consumption during input data transmission and the energy consumption during idle periods. The transmission energy consumption E_t is calculated as

$$E_t = p_t \times \sum_{i=0}^{N-1} T_{i_in}, \quad (11)$$

where p_t is the transmission power of UD. The idle energy consumption E_i is defined as

$$E_i = p_i \times \left(T - \sum_{i=0}^{N-1} T_{i_in} \right), \quad (12)$$

where p_i is the idle time power of UD. Therefore, the total energy consumption E can be defined as

$$E = E_t + E_i. \quad (13)$$

F. User Cost Model

When functions are executed on either edge servers or cloud center, users are required to pay fees to the network provider based on pricing schemes. In industry and academia, AWS

Lambda and Lambda@edge pricing schemes are widely used for serverless computing. The total user cost primarily depends on three factors: i) the number of function requests, ii) the memory size allocated by the platform, and iii) the execution time of the function. We define C_{r_i} as the request cost of v_i , which is calculated as

$$C_{r_i} = N_i \times \rho, \quad (14)$$

where N_i is the number of requests for v_i , and ρ is the request cost coefficient, which represents the cost per million requests. Let C_{e_i} is the computing cost of v_i , and it is defined as

$$C_{e_i} = N_i \times mem_i^a \times T_{i_e} \times \psi, \quad (15)$$

where ψ represents the computing cost coefficient and mem_i^a is the actual execution memory allocated for the function v_i .

Following the AWS Lambda rules [35], the platform allocates the memory resources for each function within a specific range, where the minimum is 128 MB and the maximum is 3008 MB. Besides, the platform enforces the allocations in increments of 64 MB, resulting in a discrete set of available memory values. This means that functions can only be allocated memory in the form of {128 MB, 192 MB, 256 MB, ..., 3008 MB}. If the required memory size mem_i for v_i does not correspond to any of the allowed discrete values, the platform will allocate an actual execution memory size mem_i^a to v_i , which is the closest available option, even if it is slightly larger than the required size. The mem_i^a is defined as

$$mem_i^a = \begin{cases} 128 & mem_i \leq 128, \\ 128 + 64 \times \lceil \frac{mem_i - 128}{64} \rceil & mem_i > 128. \end{cases} \quad (16)$$

For example, when the mem_i is 186 MB, the platform allocates mem_i^a as 192 MB (i.e., the closest allowed value).

Let C_i be the cost for processing v_i , and it is calculated as

$$C_i = C_{r_i} + C_{e_i}. \quad (17)$$

As a result, the total user costs C for completing the serverless application is defined as

$$C = \sum_{i=0}^N C_i. \quad (18)$$

IV. PROPOSED APPROACH

In this section, we first establish a Multi-objective Markov Decision Process (MOMDP) to describe the multi-objective optimization problem defined by (1). Afterward, we offer a detailed description of our proposed MOSEC method.

A. Modeling MOMDP

By defining a priority sequence for each function in the application, we transform the function offloading decision problem of DAG applications into a sequential decision problem. The offloading decision for each function depends solely on the current environmental state and objectives preference, without being influenced by previous states. As defined in (1), our optimization objectives include application completion time, UD energy consumption, and user costs. These objectives are conflicting and

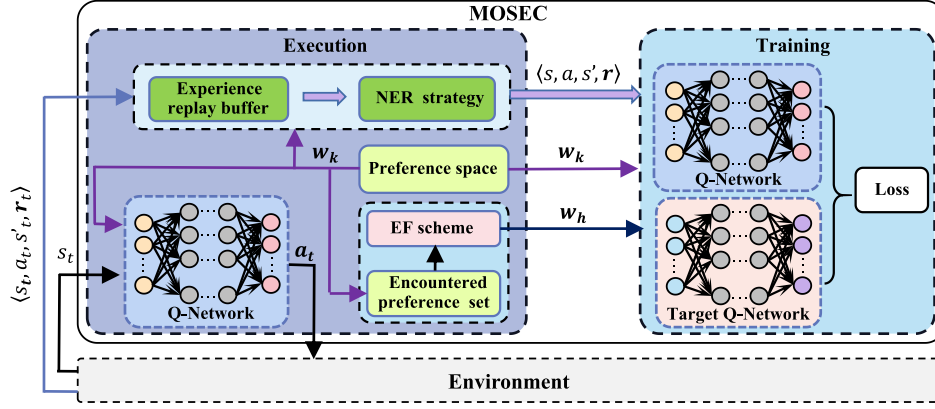


Fig. 2. Architecture of MOSEC.

require multi-objective trade-offs using a preference vector. The preferences of these objectives change dynamically over time. To address the multi-objective optimization problem in dynamic SEC environments, we model the problem as an MOMDP.

According to [28], an MOMDP can be defined as a 6-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathbf{r}, \Omega, f \rangle$, where \mathcal{S} represents the state space, \mathcal{A} denotes the action space, \mathcal{P} represents the transition probability matrix of states, $\mathbf{r}(s, a)$ is the vector of the reward function, Ω represents the space of multi-objective preference vectors, and f is a linear scalarization function that maps the value function V^π to a scalar value for policy π . Each element in Ω is a preference vector \mathbf{w} , which has been defined in Section III-C. With the linear function f , we denote $f(V^\pi, \mathbf{w}) = \mathbf{w} \cdot V^\pi$, where V^π is the value function of policy π .

States: We use S to denote the state space of the SEC environment. Specifically, we define S as

$$S = \{s_t | s_t = (\mathbf{u}_t, \mathbf{g}_t, \mathbf{c}_t), t = 0, \dots, N - 1\}, \quad (19)$$

where s_t denotes the state at step t . \mathbf{u}_t represents the features of the function to be offloaded, while \mathbf{g}_t represents the channel conditions between UD and the servers. Furthermore, \mathbf{c}_t indicates the computing capacity of the servers.

Actions: The action space is $A = \{a_t | a_t \in \{0, 1, \dots, |M|\}\}$, where a_t denotes the execution location of the function v_t . Note that v_t is the function to be offloaded at step t . Let $a_t = 0$ indicate that v_t is offloaded to the cloud center; otherwise, v_t is offloaded to an edge server.

Rewards: We use $\mathbf{r}_t = (r_t^T, r_t^E, r_t^C)$ to define the reward function, where r_t^T , r_t^E , and r_t^C are scalar rewards for three objectives, i.e., completion time, UD energy consumption, and user cost. Assume that U is the sequence of functions with priorities, and U_t is the subsequence, including the first t functions in the U . We denote $T(U_t)$ as the completion time of U_t . To minimize $T(U_t)$, we define $r_t^T = T(U_{t-1}) - T(U_t)$, which is the negative increment of $T(U_{t-1})$ after executing function v_t . For the reward r_t^E , we define E_t as the UD energy consumption and denote $r_t^E = -E_t$. Similarly, we set $r_t^C = -C_t$, where C_t is the usage charge for executing v_t .

B. Architecture of MOSEC

Fig. 2 depicts the architecture and learning process of MOSEC. The architecture of MOSEC includes an experience replay buffer with NER strategy, a preference space, an encountered preference set with EF scheme, and neural networks. The experience replay buffer is responsible for storing transitions generated during the execution phase, and the NER strategy is employed to sample a batch of transitions from the experience replay buffer. The preference space contains all preference vectors associated with three objectives, while the encountered preference set stores history-trained preference vectors alongside their corresponding latest episode numbers. We use a Double Deep Q-network (DDQN) to implement the MOSEC. The network structure of DDQN is described in Appendix A, available online.

C. Learning Process of MOSEC

The learning process of MOSEC includes the execution phase and the training phase. In the execution phase, MOSEC collects transitions and stores them in the experience replay buffer while updating the encountered preference set. In the training phase, MOSEC samples batch transformations from the experience replay buffer through **NER strategy** and selects the historically trained preference vector \mathbf{w}_h from the encountered preference set with **EF scheme**. The current preference \mathbf{w}_k , the historical preference \mathbf{w}_h , and the sampled transitions are combined as the input of the Q network and the target Q network to calculate the Q value and the target Q value. A more detailed description is supplied in the Appendix B, available online.

With the Q-values and target Q-values, MOSEC calculates the loss function and updates the Q-network parameters through gradient descent, synchronizing the target Q-network. The loss function for any transition $(s_i, a_i, s'_i, \mathbf{w}_k, \mathbf{r}_i)$ is defined as:

$$L_i = \frac{[|Q_i - Q(s_i, a_i | \mathbf{w}_h)| + |Q'_i - Q(s_i, a_i | \mathbf{w}_k)|]}{2}, \quad (20)$$

where Q_i represents the Q-value vector obtained by taking action a_i under $[s_i, \mathbf{w}_k]$. Q_i and Q'_i are defined as:

$$Q_i = r_i + \gamma \cdot \hat{Q}(s_{i+1}, \operatorname{argmax}_{a \in \mathcal{A}} Q(s_{i+1}, a | \mathbf{w}_h) \mathbf{w}_h : \mathbf{w}_h), \quad (21)$$

Algorithm 1: NER Sample Strategy.

Input: i) λ , sample coefficient; ii) b , batch size; iii) B , experience replay buffer; iv) w_t , current preference; v) s_t , current state;

Output: H , a set for collection the sampled transitions;

- 1: **for** $k = 0, \dots, \lambda b$ **do**
- 2: $\langle s_k, w_k, a_k, s'_k, r_k \rangle \leftarrow \text{Select}(B)$;
- 3: $p_k \leftarrow \|s_k - s_t\|_2 + \|w_k - w_t\|_2$;
- 4: $I \leftarrow \text{Store}(\langle s_k, w_k, a_k, s'_k, r_k \rangle, p_k)$;
- 5: **end**
- 6: $\text{Sort}(I, p)$;
- 7: **for** $j = 0, \dots, b$ **do**
- 8: $H \leftarrow \text{Store}(I_j)$;
- 9: **end**
- 10: **return** H ;

$$Q'_i = r_i + \gamma \cdot \hat{Q}(s_{i+1}, \text{argmax}_{a \in A} Q(s_{i+1}, a | w_k) w_k : w_k). \quad (22)$$

1) *NER Strategy:* Since our MOSEC is based on an off-policy learning method, it faces a state distribution shift issue, which derives from a state distribution mismatch in the data gathered from the behavior policies and target policies [38]. This data distribution shift introduces extrapolation errors, which can significantly impact the performance of off-policy algorithms. Similar to the above statement, in the dynamic multi-objective scenarios, a buffer with transitions associated with older preference vectors might not encompass any state-action pairs that a policy derived from a value network conditioned on the current preference vector [27]. This discrepancy gives rise to substantial extrapolation errors.

To effectively reduce the extrapolation errors, we adopt the NER sample strategy to our MOSEC, which is mentioned in [27]. The NER strategy considers the similarity when sampling transitions from the experience replay buffer. The similarity is calculated as the sum of the state-similarity and preference-similarity. The state-similarity is defined as the second-order norm between the current state and the state of the sampled transition, while the preference-similarity is defined as the second-order norm between the current preference vector and the preference vector at the time of the sampled transition produced. Transitions with higher similarity are closer to the current policy distribution, enabling the current policy network to rapidly adapt to new preferences.

Algorithm 1 describes the NER strategy in detail. At first, the NER strategy samples λb transitions from the replay buffer and calculates similarity scores for each of these transitions (lines 2-3), where λ is a positive integer representing the sample coefficient. Second, it adds the transition and its associated similarity score to the set I (line 4). Third, the NER strategy sorts the similarity scores in ascending order and selects a batch of b transitions with the highest similarities for network training (lines 6-9), where b is the batch size.

2) *EF Scheme:* In MOSEC, the agent aims to learn and maintain a set of optimal policies for distinct preference vectors of three objectives. However, when the agent learns a policy

Algorithm 2: Learning Process of MOSEC.

Input: i) K , # of episodes; ii) N , # of functions; iii) Ω , preference space; iv) E , encountered preference set; v) B , replay buffer; vi) b , batch size; vii) λ , sample coefficient; viii) θ , Q-network parameters; ix) θ' , parameters of target Q-network;

Output: Q-network model θ_{opt} ;

- 1: **for** $k = 1, \dots, K$ **do**
- 2: $w_k \leftarrow \text{RandomWeight}(\Omega)$;
- 3: $E \leftarrow (w_k, k)$;
- 4: **for** $t = 1, \dots, N$ **do**
- 5: $s_t \leftarrow \text{Observation}()$;
- 6: $Q \leftarrow \text{Q-network}(s_t, w_k)$;
- 7: $a_t \leftarrow \epsilon\text{-greedy}()$;
- 8: $(s_{t+1}, r_t) \leftarrow \text{EnvNextStep}(a_t)$;
- 9: $B \leftarrow (s_t, w_k, a_t, r_t, s_{t+1})$;
- 10: $\text{Transitions} \leftarrow \text{NERsample}(s_t, w_k, \lambda)$;
- 11: $w_h \leftarrow \text{EFscheme}(E)$;
- 12: **for each sampled transition** (s_i, a_i, r_i, s'_i) **do**
- 13: $Q_{\theta, i}^k \leftarrow \text{Q-network}(s_i, w_k)$;
- 14: $Q_{\theta, i}^h \leftarrow \text{Q-network}(s_i, w_h)$;
- 15: $Q_{\theta', i}^k \leftarrow \text{target Q-network}(s'_i, w_k)$;
- 16: $Q_{\theta', i}^h \leftarrow \text{target Q-network}(s'_i, w_h)$;
- 17: $y_i^k \leftarrow r_i + \gamma Q_{\theta', i}^k$;
- 18: $y_i^h \leftarrow r_i + \gamma Q_{\theta', i}^h$;
- 19: $\text{loss}_i \leftarrow \frac{1}{2} [mse(y_i^k, Q_i^k) + mse(y_i^h, Q_i^h)]$;
- 20: **end**
- 21: Update Q-network parameters θ_{opt} ;
- 22: Synchronize target Q-network each N^- step with θ_{opt} ;
- 23: **end**
- 24: **end**
- 25: **return** θ_{opt} ;

for current preferences, it can potentially result in overfitting, causing the model to deviate from previously learned policies for other preferences. This deviation triggers a catastrophic policy forgetting issues. For all the preferences stored in the encountered preference set, the policies belonging to preferences learned earlier tend to be forgotten to a greater extent as training progresses. Consequently, the model is more likely to produce larger adaptation errors on these preferences.

To address this issue, we propose an EF scheme, which chooses the earliest history-trained preference vector (i.e., the preference vector with the smallest episode number in the encountered preference set) to participate in the training of current preferences. By adopting the EF scheme, our method effectively prevents the model from overfitting on the current preference and ensures that no optimal policy is forgotten excessively. Consequently, MOSEC effectively maintains the learned optimal policies and reduces adaptation errors.

Algorithm 2 describes the learning process of MOSEC in detail. During each episode, the agent randomly selects a preference vector w_k from the set Ω and updates E with the pair

(\mathbf{w}_k, k) (lines 2-3). Each episode consists of N steps representing the number of application functions. In each step, the agent performs a function offloading. At every step, the agent first retrieves the state s_t from the environment, combining it with \mathbf{w}_k , and feeds them as inputs to the Q-network to calculate the Q-values $Q(s_t, a_t)$ (lines 5-6). It then selects an action a_t with a ϵ -greedy strategy (line 7). The agent takes a_t to interact with the environment and receives a reward vector \mathbf{r}_t and a next state s'_t (line 8). Afterwards, the experience replay buffer B records the tuple $(s_t, \mathbf{w}_k, a_t, \mathbf{r}_t, s'_t)$ (line 9). After sampling a set of transitions from B with the NER strategy, the agent selects a history-trained preference \mathbf{w}_h from E using the EF scheme (lines 10-11). For each sampled transition, the Q-network takes (s_i, \mathbf{w}_k) and (s_i, \mathbf{w}_h) as inputs, and outputs the action value function $Q_{\theta_i}^k$ and $Q_{\theta_i}^h$, respectively (lines 13-14). On the other hand, the target Q-network takes (s'_i, \mathbf{w}_k) and (s'_i, \mathbf{w}_h) as inputs and estimates the value function y_i^k and y_i^h , respectively (lines 17-18). Finally, it calculates the loss function using the mean square error function (line 19). With the loss function, the agent updates the parameters θ_{opt} of the Q-network through gradient descent and synchronizes to the target Q-network every N^- episode (lines 21-22). After the training is finished, the optimal parameters of the networks are returned (line 25).

D. Computation Complexity

The Q-network consists of an input layer, J fully connected layers, and an output layer. We use h_j to represent the number of neurons in the j -th layer, where $j \in [0, J + 1]$. Note that h_0 and h_{J+1} represent the number of neurons in the input and output layers, respectively. Assume that E_{\max} is the maximum number of episodes, and N is the number of time steps per episode. Additionally, It is worth noting that N is equal to the number of functions within an application. We denote b as the batch size. The computation complexity of MOSEC for the training model is $O(E_{\max} N b (\sum_{j=0}^J h_j h_{j+1}))$. Once the training process is finished, the trained Q-network is used to make the function offloading decisions. In each time step, the execution location of each function in an application can be generated through the Q-network. Therefore, the computation complexity of MOSEC for testing is $O(N (\sum_{j=0}^J h_j h_{j+1}))$.

V. EXPERIMENTAL RESULTS

This section presents the results of our experiments and evaluates the performance of MOSEC. First, we describe the experimental setup, including environmental settings and parameter configurations. second, the details of the performance metrics are provided. Finally, we present the experimental results, encompassing parameter studies, ablation experiments, and comparative experiments, along with the analyses of the results.

A. Experimental Setup

1) *Platform and Environment Settings*: We conducted our experiments on a generic Ubuntu server, which was equipped with a 3.7 GHz Intel CPU, 32 GB of RAM, and an NVIDIA

RTX 3080 GPU. The SEC network comprises a cloud center server and 6 edge servers. The cloud center server is equipped with a multi-core CPU and has unrestricted memory capacity. The CPU of the cloud server operates at a clock frequency of 3 GHz. Similarly, the edge servers are also multi-core machines with CPU clock frequencies ranging from 1 GHz to 3 GHz. The cold start time for containers on the edge servers is consistently set to 1s. Wireless communication between UD and the edge domain utilizes 1MHz bandwidth links, with SNRs ranging from 1.5 to 5. The UD has a transmission power of 1.5 watts and an idle power of 0.5 watts. The system is capable of processing 5,000 requests per second. In accordance with the pricing models of AWS Lambda and Lambda@edge [36], we set the request cost coefficient at 0.2 \$, and the computing cost coefficient at 0.00001667 \$ for the cloud center. Conversely, the request cost coefficient for the edge servers was set at 0.6 \$, with a computing cost coefficient of 0.00005001 \$.

2) *DDQN Agent Settings*: We employed a DDQN agent to implement MOSEC, which integrates two networks with identical structures. Except for the input layer and the output layer, each network contains two hidden layers, each with 64 neurons. The hidden layers utilize the *Tanh* as an activation function. To enhance the performance of MOSEC, we employed the *Adam* optimizer with a learning rate of 0.001. We set the discount factor to 0.99. The size of the experience replay buffer was limited to 5000, and the sample batch size was set at 64.

3) *Test Applications Settings*: Similar to [28], we generated 6 distinct DAGs to simulate the applications, labeled as App-1 to App-6. The input data size of the functions ranges from 500 KB to 600 KB, while the output data size varies between 50 KB and 100 KB. The number of CPU cycles required for function execution ranges from 1G to 5G. The memory size required for functions ranges from 64 MB to 512 MB. Both App-1 and App-2 consist of 10 distinct functions, each with varying levels of dependency complexity, as do App-3 and App-4, which feature 20 separate functions. Similarly, App-5 and App-6 each include 30 unique functions with diverse functional dependencies.

4) *Compared Algorithms*: Due to no research based on MORL for function offloading optimization problems in SEC, we select three state-of-the-art MORL-based algorithms (**Naive** [37], **CN-DWS** [19] and **ODT** [28]) to evaluate our method. These methods have been applied to address multi-objective optimization problems in other scenarios. All four methods obtain state information from the same environment to ensure fairness. The details of the compared algorithms are described as follows.

- *Naive* [37]: This approach solves the MORL problem by designing a synthetic objective function, which can represent the overall preferences. Hence, this method does not select any historically encountered preferences to participate in the training.
- *CN-DWS* [19]: This method employs conditional networks (CNs) to acquire multi-objective optimization policies in dynamic preference vector settings. To improve the dynamic adaptability of the model, CN-DWS randomly selects a previously trained preference vector and integrates it into the current training process.

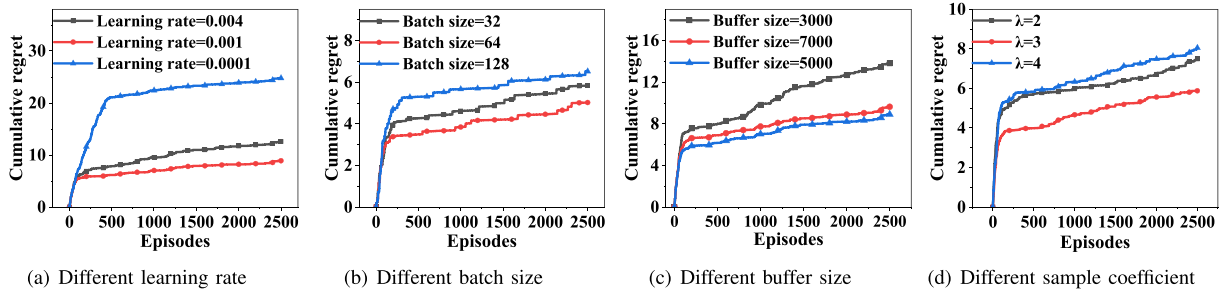


Fig. 3. Comparison of cumulative regret for different parameters of MOSEC.

- *ODT* [28]: This is an improved CN-DWS method used to optimize three objectives for the task offloading problem in multi-access edge computing. Compared to CN-DWS, ODT uses a Tournament Selection scheme to choose a history-trained preference vector to participate in the current training process.
- *MOSEC*: Our proposed method for the function offloading problem in SEC uses a NER strategy to sample the more similar experience data to train the model. Furthermore, it adopts an EF scheme to select a history-learned preference vector to participate in current training.

B. Performance Metrics

Cumulative regret: Cumulative regret represents the sum of the differences between the actual cumulative rewards in each episode and the optimal cumulative reward during the training process. As training progresses, the model should decrease the regret, indicating improved performance across multiple objectives and the ability to balance relationships between different objectives. Cumulative regret is typically employed in MORL algorithms to represent the convergence performance of the model.

Adaptation error: For a specific preference vector, the adaptation error is defined as the relative error between the actual cumulative reward in the test phase and the optimal cumulative reward in the training phase. A smaller adaptation error indicates that the model is more adaptive to the dynamic preferences, and no optimal policy is forgotten excessively.

C. Results and Analysis

Based on the experimental setup mentioned above, we first conducted a parameter study to investigate the influence of the learning rate, experience replay buffer size, batch size, and sample coefficient on the performance of MOSEC. Furthermore, we conducted ablation experiments on MOSEC to validate the effectiveness of the NER strategy and the EF scheme. After that, we evaluated and compared the overall performance of our MOSEC approach with state-of-the-art dynamic multi-objective optimization methods. Finally, we analyzed the sensitivity and scalability of MOSEC.

1) *Parameters Study on MOSEC*: This study aimed to analyze the impact of various parameters on the performance of MOSEC. As the scale of App-4 is mid-size compared to other tested applications, we have chosen it as a case to assess the

impact of the learning rate, experience buffer size, batch size, and sample coefficient on the performance of MOSEC. The results for the cumulative regret are shown in Fig. 3.

Fig. 3(a) demonstrates the impact of various learning rates on the performance of MOSEC. It is evident that both larger and smaller learning rates result in poorer cumulative regret curves. For the experiment, we set the learning rate to 0.001 since it produces the best performance.

As shown in Fig. 3(b), a smaller batch size implies selecting fewer transitions for training the model, which limits the diversity of training data and slows down the convergence of the model. On the other hand, a larger batch size means sampling more low-similarity transitions, increasing extrapolation errors, and decreasing the quality of cumulative regret. Therefore, we set the batch size to 64.

In Fig. 3(c), a smaller experience replay buffer size leads to an increase in cumulative regret. This phenomenon can be attributed to the nature of the experience replay buffer, which operates as a first-in-first-out queue. A smaller buffer size restricts the storage capacity for the experiences, causing valuable transition data to be prematurely discarded. Conversely, when the buffer size is increased, the impact of buffer size on performance becomes insignificant. Based on these findings, we set the buffer size to 5000 for our experimental setup.

Fig. 3(d) illustrates the impact of the sample coefficient λ on the performance of MOSEC. The value of λ affects the size of the sampled data, and as the λ increases, the size of the sampled data also grows, resulting in a large amount of data with similar experiences being sampled. When the sampled data is processed by the NER strategy, a large number of transitions with similar similarity scores may be selected for training, which reduces the training efficiency. A smaller λ value corresponds to smaller sampled data sizes, negatively affecting the diversity of data involved in training and slowing the convergence, consequently resulting in an increase in cumulative regret. Therefore, we set the NER sample coefficient to 3.

2) *Ablation Experiment*: During the model training phase, MOSEC utilizes the NER strategy to mitigate extrapolation error, thereby reducing cumulative regret. Meanwhile, it incorporates the EF scheme to select a historically trained preference vector for participation in current training, ensuring the adaptability of the network to dynamic preference vectors. We conducted a series of ablation experiments on 6 test applications to validate the effectiveness of the NER strategy and EF scheme.

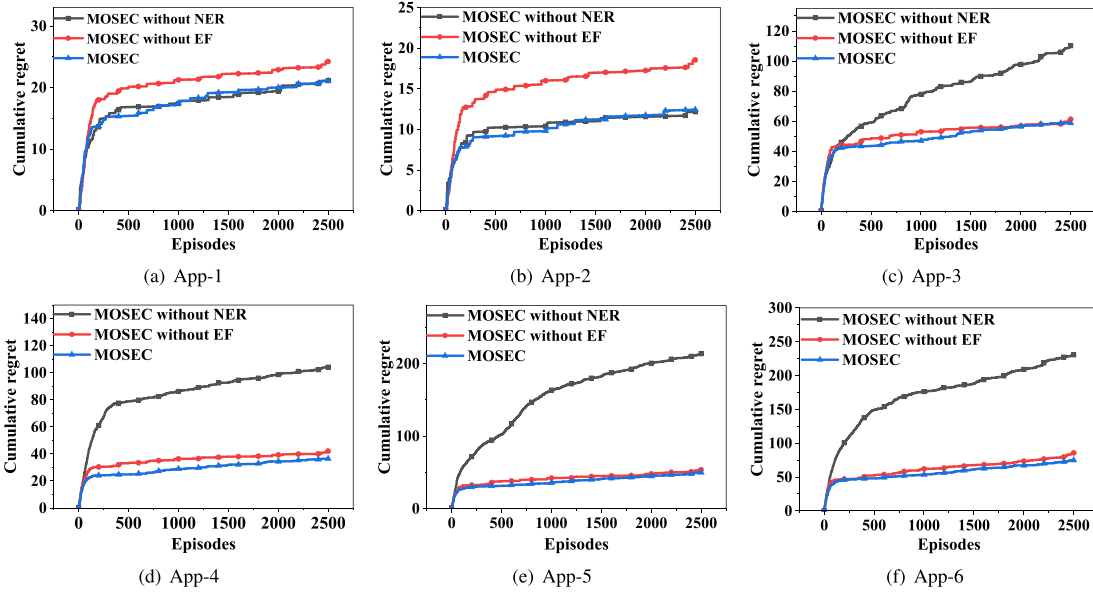


Fig. 4. Comparison of cumulative regret for different MOSEC variants.

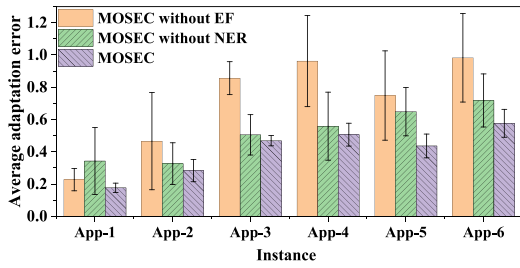


Fig. 5. Comparison of adaptation error for different MOSEC variants.

In these experiments, the number of training steps in each episode is equal to the number of functions in the application. We compared the cumulative regrets and average adaptation errors across three variants, i.e., the standard MOSEC (with both NER and EF), MOSEC without NER (utilizing a standard sample strategy instead of NER), and MOSEC without EF (employing a random preference vector selection scheme instead of EF). The experimental results are presented in Figs. 4 and 5.

Fig. 4 compares the cumulative regret of different MOSEC variants. For applications with fewer functions, as shown in Fig. 4(a) and (b), the limited number of steps per training episode results in minimal distribution shift and relatively smaller extrapolation errors. The NER strategy is unable to demonstrate its advantages. Consequently, the performance of MOSEC is comparable to MOSEC without NER. Meanwhile, MOSEC without EF exhibits the least favorable cumulative regret curves. This is attributed to the fewer steps per episode leading to lower policy update frequency, exacerbating the tendency to forget the policies of preferences that had been trained, thereby increasing cumulative regret. In contrast, for applications with more functions, such as in Fig. 4(c) and (f), more steps per episode bring a greater accumulation of extrapolation errors,

leading to the worst cumulative regret curve for MOSEC without NER. The other two variants with the NER strategy produce better cumulative regret curves, demonstrating the advantage of the NER method on applications with more functions. The performance of MOSEC without EF is comparable to MOSEC. This is because more training steps help the model maintain a better balance among different trained preferences, reducing long-term forgetting and thus decreasing long-term cumulative regret.

Fig. 5 compares average adaptation errors and Standard Deviation (SD) for different MOSEC variants. In most applications, MOSEC without EF exhibits the highest adaptability error, while standard MOSEC demonstrates the best performance in terms of adaptability error. This indicates that standard MOSEC gains greater benefits from the EF scheme in reducing adaptability errors. The EF scheme addresses the issue of overfitting the current preference vector while preventing the optimal policy for a specific preference vector from being forgotten, thereby significantly enhancing the model to adapt to different preference vectors.

3) *Overall Performance Evaluation*: First, we assessed the performance of MOSEC in optimizing three objectives, i.e., completion time, UD energy consumption, and user cost. Second, we conducted a series of experiments on 6 test applications to compare MOSEC with three state-of-the-art multi-objective optimization algorithms regarding cumulative regrets, adaptation errors, and the average optimization performance for three objectives.

Optimization performance for multi-objectives: We took App-4 as an instance to evaluate the optimization performance for three objectives with various preference vectors. As shown in Fig. 6, when MOSEC optimizes only one objective and forsakes the other two, the optimized objectives all reach their respective minimum values. Furthermore, when MOSEC emphasizes optimizing UD energy consumption (reducing energy

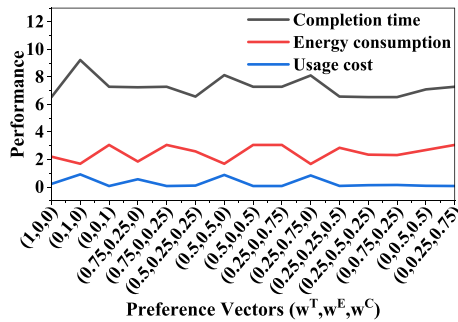


Fig. 6. Performance for three objectives with different preferences.

consumption), it consistently maintains or worsens completion time and user cost. When MOSEC prioritizes the optimization of completion time (reducing completion time), it invariably leads to an improvement or stability in UD energy consumption, but the trend in user cost follows the completion time (with different slopes). This arises from the fact that both completion time and user cost are associated with the computing time of functions. Similarly, when MOSEC emphasizes optimizing user cost (reducing user cost), it consistently results in degradation or stability of UD energy consumption, while the change in completion time follows the trend in user cost.

Cumulative regrets: Fig. 7 compares cumulative regret among various methods. In all the test applications, MOSEC consistently outperforms other comparative methods, showing the smallest cumulative regret in each episode. The advantage contributing to this superior performance is that MOSEC utilizes the NER method to sample the more similar transitions from the experience replay buffer, effectively mitigating the impact of extrapolation errors on its performance. In contrast, the Naive method exhibits the highest cumulative regret. CN-DWS and ODT demonstrate similar cumulative regrets across all the test applications since they randomly retrieve data from the experience replay buffer, leading to increased extrapolation errors and higher cumulative regret. The statistical results of cumulative regret are shown in Fig. 8.

Fig. 8 presents the average episodic regrets and SDs for four MORL algorithms across 5 experimental rounds. For applications with a limited number of functions, such as APP-1 and APP-2, Naive exhibits the highest average episodic regret and SD, and the performance of the other three methods is relatively comparable. However, as the number of functions increases, such as APP-4 to APP-6, the complexity of functional dependencies within the applications also increases. Under these conditions, MOSEC consistently demonstrates the lowest average episodic regret and SD. This observation underscores the superior performance of MOSEC compared to the other three algorithms.

Average adaptation errors: Fig. 9 displays the average adaptation errors and SDs obtained by four methods in 5 rounds of experiment, in which MOSEC consistently demonstrates the lowest average adaptation errors and SDs across all the test applications. These results indicate that MOSEC effectively enables the model to adapt to dynamic preferences stably. The superiority of MOSEC can be attributed to the EF scheme,

TABLE I
RESULT OF AVERAGE COMPLETION TIME (SEC.)

Instance	Naive	CN-DWS	ODT	MOSEC
App-1	4.1701	3.9657	3.9669	3.9390
App-2	3.6748	3.3964	3.4502	3.4595
App-3	14.9312	12.7442	13.7156	13.0775
App-4	11.7186	11.5553	11.5377	11.2324
App-5	21.6620	20.9757	20.8294	20.7487
App-6	21.7473	19.6067	19.8094	19.4691

TABLE II
RESULT OF AVERAGE UD ENERGY CONSUMPTION (J)

Instance	Naive	CN-DWS	ODT	MOSEC
App-1	4.1087	4.0065	4.0071	3.9931
App-2	3.8353	3.6961	3.7230	3.7277
App-3	11.2177	10.1242	10.6099	10.3265
App-4	9.568	9.4863	9.4775	9.3249
App-5	16.4384	16.0952	16.0952	15.9817
App-6	16.3985	15.2568	15.4296	15.3761

TABLE III
RESULT OF AVERAGE USER COST (\$)

Instance	Naive	CN-DWS	ODT	MOSEC
App-1	0.1902	0.1737	0.1767	0.1728
App-2	0.1536	0.1338	0.1379	0.1385
App-3	1.0182	0.8082	0.9294	0.8070
App-4	0.7640	0.7008	0.7008	0.6875
App-5	1.3131	1.3221	1.2630	1.2576
App-6	1.2545	1.1030	1.1098	1.0805

which selects the earliest history-trained preference vectors from the encountered preference set to participate in training for the current preference vector. The EF scheme prevents the model from overfitting during training for the current preference, which results in the forgetting of previously learned optimal policies. This ensures that the model can adapt to various multi-objective preferences.

Average optimization performance for multiple objectives: We conducted a series of additional experiments to validate the performance of MOSEC in optimizing three objectives, including average completion time, UD energy consumption, and user cost across all preference vectors. The results are presented in Tables I, II, and III, with the best results highlighted in bold. In Table I, MOSEC consistently demonstrates shorter completion times compared to other algorithms in the majority of tested applications, except for App-2. Regarding average UD energy consumption in Table II, MOSEC performs the best in App-1, App-4, and App-5, while CN-DWS achieves the best results in App-3 and App-6. However, CN-DWS exhibits poor performance in terms of completion time and user cost. Concerning the average user cost, MOSEC outperforms other algorithms in all applications except for App-2. In summary, Naive performs the worst across all tested applications, while CN-DWS and ODT show good optimization performance only in specific cases. MOSEC emerges as the best-performing algorithm in most tested applications, surpassing other algorithms in various aspects.

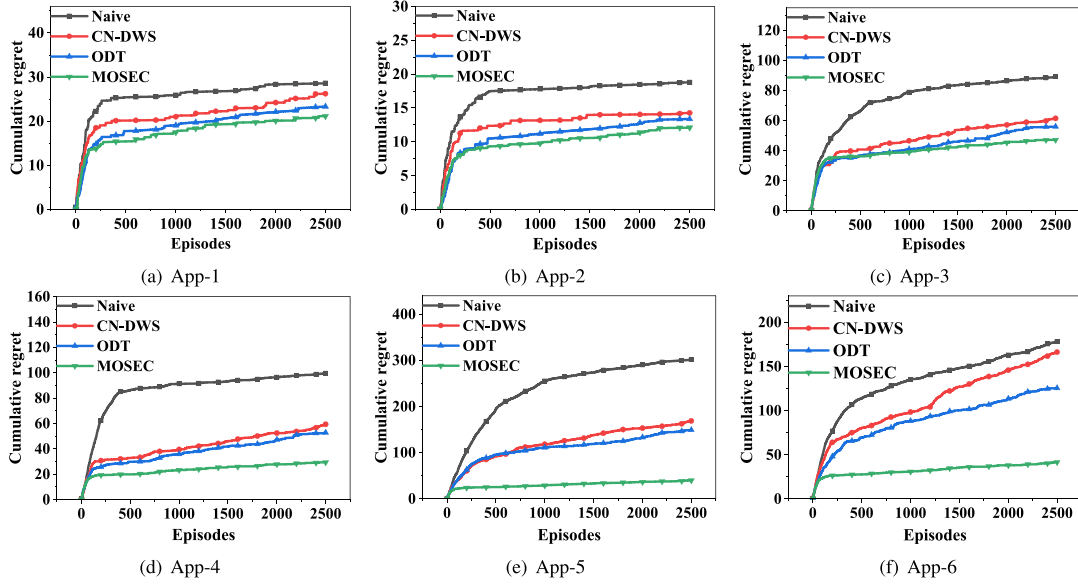


Fig. 7. Comparison of cumulative regret for different methods.

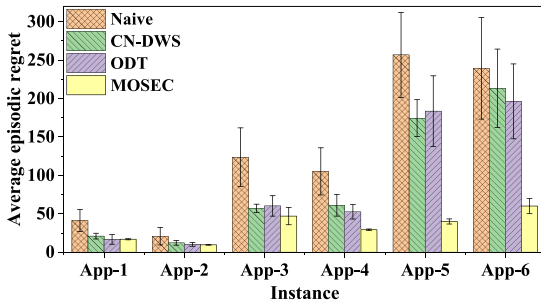


Fig. 8. Comparison of average episodic regret for different methods.

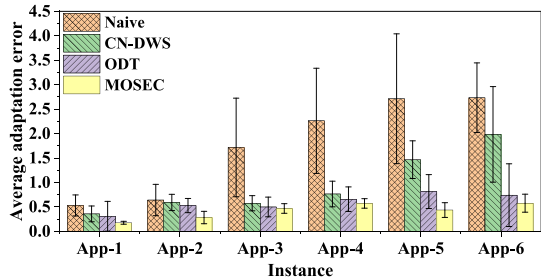


Fig. 9. Comparison of adaptation error for different methods.

4) *Sensitivity Analysis*: In the first part of this subsection (i.e., the parameter study on MOSEC), we discussed MOSEC’s sensitivity to the training parameters. Here, we will explore MOSEC’s sensitivity to server processing capacity.

Taking App-4 as an example, we conducted experiments by scaling the processing capacity of servers to 0.6, 0.8, 1.2, and 1.4 times their default values to investigate the impact of server processing capability on performance. The average performance of MOSEC with different processing capacities is summarized in Table IV. The results show that increasing the processing capabilities of servers leads to an improvement in average performance. This improvement can be attributed to the enhanced

 TABLE IV
 THE PERFORMANCE ON DIFFERENT PROCESSING CAPACITIES OF SERVERS

Objectives	$0.6 \times \omega$	$0.8 \times \omega$	$1 \times \omega$	$1.2 \times \omega$	$1.4 \times \omega$
Average CT (Sec.)	17.1634	16.4369	15.1111	14.6542	13.7007
Average EC (J)	12.9379	12.5747	11.9177	11.6833	11.6833
Average UC (\$)	0.9961	0.956	0.7149	0.695	0.6911

server processing capacity, which reduces the execution time of each function and leads to reduced application completion times, lower UD energy consumption, and decreased application user costs.

5) *Scalability Analysis*: In Section VI-C, we generated 6 test applications with varying numbers of functions and different levels of complexity. We applied our algorithm for function offloading, and the results illustrate its capability to offload applications with diverse specifications. As depicted in Figs. 7, 8, and 9, while our approach outperforms other algorithms in achieving function offloading, increasing the number of functions within applications leads to higher cumulative regrets and adaptation errors. Consequently, this results in decreased performance.

VI. DISCUSSION

The advantages of this work can be summarized in two main aspects. First, it is the first attempt to redefine the function offloading problem in SEC as a multi-objective optimization problem, in which the optimization objectives are completion time, UD energy consumption, and user cost. Even more, the preference vector of the three objectives changes over time. Second, since the dynamic nature of preferences introduces complexity, this work proposed a MORL-based method that incorporates the NER sample strategy and EF scheme for the first time to enhance the performance.

In practice, service providers can deploy the MOSEC algorithm on edge servers. When a serverless application is deployed on a cloud-edge collaborative platform, UD can request the application from the servers and operate in energy-saving mode, real-time mode, or economic mode. By assigning different preferences to these three objectives, UD can also operate in a hybrid mode. Our algorithm selects the most suitable execution location for each application function based on the operating mode and preference vector of the UD, thereby meeting their performance requirements. Furthermore, our approach can be employed to intelligently schedule dependent workflows in factory green workshops with the requirement of optimizing multiple objectives.

However, our experimental results imply that MOSEC shows growth in both cumulative regret and adaptation error as the scale and complexity of applications increase. In our experiments, we tested MOSEC with applications containing up to 30 functions. It is worth noting that as the number of functions in applications continues to grow, the increasing cumulative regret and adaptation errors may lead to MOSEC not performing optimally.

In future work, we plan to explore more effective methods to reduce adaptation errors and improve the algorithm's performance. Furthermore, we intend to establish a cloud server to create a cloud-center serverless platform and acquire Raspberry Pi devices to establish edge serverless platforms. With these setups, we will evaluate the performance of our approach by offloading real-world applications.

VII. CONCLUSION

Function offloading problem in Serverless Edge Computing (SEC) plays a crucial role in enhancing application performance. Current research has extensively explored offloading strategies primarily focused on optimizing a single performance objective. However, a significant challenge arises when applications need to optimize multiple objectives, especially when the relative importance of these objectives dynamically changes. To address this challenge, this paper proposed a novel approach for function offloading based on Multi-objective Reinforcement Learning (MORL), named MOSEC, which aims to optimize application completion time, User Device (UD) energy consumption, and user cost according to the relative importance of three objectives. To reduce extrapolation errors, MOSEC incorporates a Near-on Experience Replay (NER) strategy in model training. Furthermore, we introduced the Earliest First (EF) scheme to efficiently maintain previously learned policies, thereby mitigating the adaptation errors. Comprehensive experiments conducted across various generated applications confirmed the effectiveness of MOSEC. It consistently outperforms state-of-the-art multi-objective optimization algorithms, demonstrating its ability to effectively address the function offloading problem in SEC.

REFERENCES

- [1] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–34, 2022.
- [2] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–32, 2022.
- [3] H. Ko, S. Pack, and V. C. Leung, "Performance optimization of serverless computing for latency-guaranteed and energy-efficient task offloading in energy harvesting industrial IoT," *IEEE Internet Things J.*, vol. 10, no. 3, pp. 1897–1907, Feb. 2023.
- [4] 2024. [Online]. Available: <https://aws.amazon.com/cn/campaigns/lambda/>
- [5] 2024. [Online]. Available: <https://cloud.google.com/functions/docs/concepts/overview?hl=zh-cn>
- [6] 2024. [Online]. Available: <https://azure.microsoft.com/en-us/products/functions/>
- [7] M. S. Aslanpour et al., "Serverless edge computing: Vision and challenges," in *Proc. Australas. Comput. Sci. Week Multiconference*, 2021, pp. 1–10.
- [8] P. Mendki, "Evaluating web assembly enabled serverless approach for edge computing," in *Proc. Cloud Summit*, 2020, pp. 161–166.
- [9] 2024. [Online]. Available: <https://aws.amazon.com/cn/lambda/edge/>
- [10] Q. L. Trieu, B. Javadi, J. Basilakis, and A. N. Toosi, "Performance evaluation of serverless edge computing for machine learning applications," 2022, *arXiv:2210.10331*.
- [11] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang, "When serverless computing meets edge computing: Architecture, challenges, and open issues," *IEEE Wireless Commun.*, vol. 28, no. 5, pp. 126–133, Oct. 2021.
- [12] Y. Li, D. Zeng, L. Gu, K. Wang, and S. Guo, "On the joint optimization of function assignment and communication scheduling toward performance efficient serverless edge computing," in *Proc. Int. Symp. Qual. Serv.*, 2022, pp. 1–9.
- [13] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and R. Gaire, "Energy-aware resource scheduling for serverless edge computing," in *Proc. Int. Symp. Cluster Cloud Internet Comput.*, 2022, pp. 190–199.
- [14] R. Xie, D. Gu, Q. Tang, T. Huang, and F. R. Yu, "Workflow scheduling using hybrid PSO-GA algorithm in serverless edge computing for the Internet of Things," in *Proc. Veh. Technol. Conf.*, 2022, pp. 1–7.
- [15] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *Proc. Int. Symp. Cluster Cloud Internet Comput.*, 2020, pp. 41–50.
- [16] R. Xie, D. Gu, Q. Tang, T. Huang, and F. R. Yu, "Workflow scheduling in serverless edge computing for the industrial Internet of Things: A learning approach," *IEEE Trans. Ind. Inform.*, vol. 19, no. 7, pp. 8242–8252, Jul. 2023.
- [17] I. Rahimi, A. H. Gandomi, F. Chen, and E. Mezura-Montes, "A review on constraint handling techniques for population-based algorithms: From single-objective to multi-objective optimization," *Arch. Comput. Methods Eng.*, vol. 30, no. 3, pp. 2181–2209, 2023.
- [18] W. Gao, Y. Wang, L. Liu, and L. Huang, "A gradient-based search method for multi-objective optimization problems," *Inf. Sci.*, vol. 578, pp. 129–146, 2021.
- [19] A. Abels, D. Roijers, T. Lenaerts, A. Nowé, and D. Steckelmacher, "Dynamic weights in multi-objective deep reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 11–20.
- [20] X. Nian, A. A. Irissappane, and D. Roijers, "DCRAC: Deep conditioned-recurrent actor-critic for multi-objective partially observable environments," in *Proc. Int. Conf. Auton. Agents Multiagent Syst.*, 2020, pp. 931–938.
- [21] T. Basaklar, S. Gumussoy, and U. Y. Ogras, "PD-MORL: Preference-driven multi-objective reinforcement learning algorithm," 2022, *arXiv:2208.07914*.
- [22] H. Lu, D. Herman, and Y. Yu, "Multi-objective reinforcement learning: Convexity, stationarity and pareto optimality," in *Proc. Int. Conf. Learn. Representations*, 2023, pp. 1–27.
- [23] J. Skalse, L. Hammond, C. Griffin, and A. Abate, "Lexicographic multi-objective reinforcement learning," 2022, *arXiv:2212.13769*.
- [24] Y. Ma et al., "Multi-objective congestion control," in *Proc. Eur. Conf. Comput. Syst.*, 2022, pp. 218–235.
- [25] L. Liu, H. Chen, and Z. Xu, "SPMOO: A multi-objective offloading algorithm for dependent tasks in IoT cloud-edge-end collaboration," *Information*, vol. 13, no. 2, 2022, Art. no. 75.
- [26] S. Almasri, M. Jarrah, and B. Al-Duwairi, "Multi-objective optimization of task assignment in distributed mobile edge computing," *J. Reliable Intell. Environments*, vol. 8, no. 1, pp. 21–33, 2022.
- [27] S. Wang, M. Reymond, A. A. Irissappane, and D. M. Roijers, "Near on-policy experience sampling in multi-objective reinforcement learning," in *Proc. Int. Conf. Auton. Agents Multiagent Syst.*, 2022, pp. 1756–1758.
- [28] F. Song, H. Xing, X. Wang, S. Luo, P. Dai, and K. Li, "Offloading dependent tasks in multi-access edge computing: A multi-objective reinforcement learning approach," *Future Gener. Comput. Syst.*, vol. 128, pp. 333–348, 2022.

- [29] 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.
- [30] L. Liu, H. Tan, S. H. C. Jiang, Z. Han, X. Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *Proc. Int. Symp. Qual. Serv.*, 2019, pp. 1–10.
- [31] S. Deng et al., "Dependent function embedding for distributed serverless edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2346–2357, Oct. 2022.
- [32] S. Zheng, B. Liu, W. Lin, X. Ye, and K. Li, "A package-aware scheduling strategy for edge serverless functions based on multi-stage optimization," *IEEE Trans. Ind. Inform.*, vol. 13, no. 2, pp. 1–9, Mar. 2021.
- [33] 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, no. 2, pp. 74–86, 2023.
- [34] L. Pan, X. Liu, Z. Jia, J. Xu, and X. Li, "A multi-objective clustering evolutionary algorithm for multi-workflow computation offloading in mobile edge computing," *IEEE Trans. Cloud Comput.*, vol. 11, no. 2, pp. 1334–1351, Second Quarter 2023.
- [35] T. Elgamel, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *Proc. ACM/IEEE Symp. Edge Comput.*, 2018, pp. 300–312.
- [36] 2024. [Online]. Available: <https://aws.amazon.com/cn/lambda/pricing/>
- [37] C. Liu, X. Xu, and D. Hu, "Multiobjective reinforcement learning: A comprehensive overview," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 45, no. 3, pp. 385–398, Mar. 2015.
- [38] R. Islam, K. K. Teru, D. Sharma, and J. Pineau, "Off-policy policy gradient algorithms by constraining the state distribution shift," 2019, *arXiv: 1911.06970*.
- [39] X. Liu, Z. Y. Chai, Y. L. Li, Y. Y. Cheng, and Y. Zeng, "Multi-objective deep reinforcement learning for computation offloading in UAV-assisted multi-access edge computing," *Inf. Sci.*, vol. 642, 2023, Art. no. 119154.
- [40] N. Yang, J. Wen, M. Zhang, and M. Tang, "Multi-objective deep reinforcement learning for mobile edge computing," in *Proc. IEEE Int. Symp. Model. Optim. Mobile Ad Hoc Wireless Netw.*, 2023, pp. 1–8.



Yaning Yang (Student Member, IEEE) is an associate professor with the School of Physics and Electronic Information Engineering, Ningxia Normal University, Guyuan, China. She is currently working toward the PhD degree with the Software Engineering Institute, East China Normal University, Shanghai, China. Her research interests include reinforcement learning, embedded systems, cloud/edge computing, and serverless computing.



Xiao Du (Student Member, IEEE) received the BS degree in electronic information engineering from Leshan Normal University, China, in 2015. Currently, he is currently working toward the PhD degree with Software Engineering Institute, East China Normal University, China. His research interests include multi-agent reinforcement learning and resource management in massive random access.



Yutong Ye (Student Member, IEEE) received the BS degree from the School of Computer Science and Information Engineering, Guangxi Normal University, Guilin, China, in 2020. He is currently working toward the PhD degree with the Software Engineering Institute, East China Normal University, Shanghai, China. His research interests include reinforcement learning, embedded systems and Internet of Things.



Jiepin Ding received the MS degrees from the Department of Information Management and Artificial Intelligence, Zhejiang University of Finance and Economics, Hangzhou, China, in 2016 and 2020, respectively. She is currently working toward the PhD degree with the Software Engineering Institute, East China Normal University, Shanghai, China. Her research interests include production scheduling, heuristic algorithm, and reinforcement learning.



Ting Wang (Senior Member, IEEE) received the PhD degree in computer science and engineering from the Hong Kong University of Science and Technology, Hong Kong, China, in 2015. He is currently an associate professor with the Software Engineering Institute, East China Normal University, Shanghai, China. Before joining ECNU in 2020, he worked with the Bell Labs as a research scientist from 2015 to 2016, and with Huawei as a senior engineer from 2016 to 2020. His research interests include cloud/edge computing, serverless computing, federated learning, data center networks



Mingsong Chen (Senior Member, IEEE) received the PhD degree in computer engineering from the University of Florida, Gainesville, in 2010. He is currently a professor with the Software Engineering Institute, East China Normal University. His research interests are in the area of design automation of cyber-physical systems, EDA, embedded systems, and formal verification techniques. Currently he serves as the director of Engineering Research Center of Software/Hardware Co-design Technology and Application affiliated to the Ministry of Education, China, and the vice director of technical committee of embedded systems of China Computer Federation (CCF). He is an associate editor of *IET Computers & Digital Techniques*, and *Journal of Circuits, Systems and Computers*.



Keqin Li (Fellow, IEEE) is a SUNY distinguished professor of computer science with the State University of New York. He is also a distinguished professor with Hunan University, China. His current research interests include cloud computing, fog/edge computing and serverless computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, Big Data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing.