Distributed and Collective Deep Reinforcement Learning for Computation Offloading: A Practical Perspective

Xiaoyu Qiu, Weikun Zhang, Wuhui Chen, *Member, IEEE*, and Zibin Zheng, *Senior Member, IEEE*

Abstract—Mobile edge computing (MEC) is a promising solution to support resource-constrained devices by offloading tasks to the edge servers. However, traditional approaches (e.g., linear programming and game-theory methods) for computation offloading mainly focus on the immediate performance, potentially leading to performance degradation in the long run. Recent breakthroughs regarding deep reinforcement learning (DRL) provide alternative methods, which focus on maximizing the cumulative reward. Nonetheless, there exists a large gap to deploy real DRL applications in MEC. This is because: 1) training a well-performed DRL agent typically requires data with large quantities and high diversity, and 2) DRL training is usually accompanied by huge costs caused by trial-and-error. To address this mismatch, we study the applications of DRL on the multi-user computation offloading problem from a more practical perspective. In particular, we propose a distributed and collective DRL algorithm called DC-DRL with several improvements: 1) a distributed and collective training scheme that assimilates knowledge from multiple MEC environments, which not only greatly increases data amount and diversity but also spreads the exploration costs, 2) an updating method called adaptive n-step learning, which can improve training efficiency without suffering from the high variance caused by distributed training, and 3) combining the advantages of deep neuroevolution and policy gradient to maximize the utilization of multiple environments and prevent the premature convergence. Lastly, evaluation results demonstrate the effectiveness of our proposed algorithm. Compared with the baselines, the exploration costs and final system costs are reduced by at least 43 and 9.4 percent, respectively.

Index Terms—Mobile edge computing, computation offloading, deep reinforcement learning, distributed and collective training, n-step return, deep neuroevolution, policy gradient

1 Introduction

In RECENT years, the proliferation of smart mobile devices has promoted the popularity of a multitude of new mobile applications. However, in general, mobile devices have limited computation capacity and power supply, directly affecting the computation performance and user experience. Driven by this, mobile edge computing (MEC) is emerging as a promising solution that integrates mobile devices with computational resources in the proximity, thereby alleviating the above challenges through computation offloading [1], [2].

Nonetheless, catering for varying user requests with high-performance demand is a non-trivial task even with the virtue of MEC. The significant challenge lies in the allocation of computational and network resources [3]. Although previous studies on computation offloading (e.g., linear programming and game-theory methods) have obtained several achievements [4], they have yet to reach the practical level because they mostly consider one-shot optimization and are based on the system model assumptions that may not be adequate to characterize real-world scenarios. Fortunately, recent breakthroughs in reinforcement learning (RL) provide a promising approach, designed to learn the optimal policy and maximize the long-term rewards without prior knowledge of system models. RL works as an agent to optimize the policy based on historical interactions with environments. Besides, deep reinforcement learning (DRL) leverages the powerful deep neural networks (DNNs) to enable RL to handle large state spaces. A large set of studies have proved its effectiveness [5], [6].

However, it is impractical to apply existing DRL algorithms directly to MEC. First, the DRL algorithmic paradigms proposed in most previous studies rely on numerous interactions with training environments to obtain experiences with large quantities and high diversity. In DRL, past experiences are used as training set data in supervised learning so as to obtain the optimal policy. Because these algorithms are designed for scenarios such as video games or neural architecture search [7], [8], where the environments can be created in memory by well-designed programming, experiences needed for DRL training can be obtained easily by increasing the computing resources. Due to the fact that it is impractical to build unbiased simulation models for many real-world problems, it is inevitable to train DRL agents in real-world environments, where the training experiences generated are not nearly as unbounded as they would be in the algorithm design. This mismatch makes it slow to uptake DRL applications in MEC. Second, DRL

- *The authors are with the School of Data and Computer Science, National Engineering Research Center of Digital Life, Sun Yat-sen University, Guangzhou 510006, China. E-mail: {qiuxy23, zhangwk8}@mail2.sysu.edu.cn, {chenwuh, zhzibin}@mail.sysu.edu.cn.*

Manuscript received 20 Feb. 2020; revised 28 Nov. 2020; accepted 1 Dec. 2020. Date of publication 9 Dec. 2020; date of current version 15 Dec. 2020.
(Corresponding authors: Wuhui Chen.)
Recommended for acceptance by jianfeng Zhan.
Digital Object Identifier no. 10.1109/TPDS.2020.3042599

training is usually accompanied by huge costs of trial-and-error (also known as exploration costs). For the computation offloading in MEC, the costs of trial-and-error are associated with the actual costs to the users, not the numerical values as in scenarios such as video games. Therefore, another difficulty lies in the huge training costs of a high-performance DRL agent, which is often unaffordable for a single MEC environment. To the best of our knowledge, these challenges have received little attention in previous studies.

To address the challenges, it is natural to consider sharing experiences and spreading the costs of trial-and-error over multiple MEC environments, and thereby achieving a larger number and greater diversity of experiences and lower exploration costs. In view of this, we explore distributed collective training as a tool to empower DRL with a more practical perspective. In such a distributed training system, multiple MEC environments collectively train DRL agents and update network parameters. However, it also presents unique challenges. On one hand, in view of the "policy gap" [9] caused by distributed training, it is important to coordinate multiple DRL agents deployed in the distributed system. On the other hand, to maximize the utilization of environments, it is important to design a system that can assimilate experiences and knowledge from multiple environments and DRL agents, so as to develop a collective, comprehensive, and high-performance policy.

Therefore, in this work, we propose a **D**istributed and **C**ollective **DRL**-based algorithm called DC-DRL, which can adaptively make the offloading and channel allocation decisions without awareness of the system model. In contrast with previous studies, the DC-DRL algorithm is empowered with a more practical perspective by leveraging a novel training scheme, where multiple MEC environments in the distributed system work collectively. In this way, the convergence speeds of DRL agents are significantly accelerated and the exploration costs of each environment are greatly reduced. In addition, we have enhanced the distributed and collective training scheme of the DC-DRL algorithm to maximize the benefits of multiple MEC environments. It should be noted that the DC-DRL algorithm is designed to address the above challenges and is not limited by computation offloading scenarios. It can be applied to many problems concerned with training intelligent agents for long-term performance maximization. Our contributions can be summarized as follows:

- We study the computation offloading problem for resource-intensive and deadline-sensitive applications in MEC. To meet the time-varying user requests and consider the long-term performance, we formulate the optimization problem as a Markov decision process (MDP) and take advantage of the recent advances in DRL to solve the problem.
- We propose the DC-DRL algorithm from a practical perspective, which can assimilate experiences and knowledge from the distributed system to obtain a collective DRL agent with high performance and low exploration costs.
- To remove the impact of the "policy gap" caused by distributed training, we follow the idea of n-step learning and off-policy correction [10], and further propose a new update method, namely adaptive n-step learning.
- We combine the advantages of deep neuroevolution [11] and policy gradient, which greatly improves the utilization of distributed system and the convergence results.

The remainder of this paper is structured as follows. In Section 2, we briefly present related work. In Section 3, we introduce the system model. In Section 4, we formulate the problem as an MDP. In Section 5, we introduce the design of the DC-DRL algorithm. Then, in Section 6, the numerical simulations and experiments are presented. Lastly, in Section 7, we conclude the paper.

2 Related Work

2.1 Computation Offloading for MEC

First, MEC has been widely recognized as a key technology to improve application performance [12], [13]. Extensive efforts have been made to optimize the computation offloading performance in MEC with approaches such as Mathematical Programming, Genetic Algorithm, and Lagrangian Relaxation. For example, Li *et al.* [14] proposed an opportunistic computation offloading algorithm to minimize communication overheads and maximize energy efficiency. A deep-learning-based response-time prediction framework was proposed in [15] for high-computation processes and real-time applications. Zhao *et al.* [16] proposed a distributed resource allocation algorithm that can significantly improve resource utilization, especially in the case of heavy computation load and insufficient edge resources.

However, most of the existing research approaches consider one-shot optimization, which may fail to achieve the expected performance in dynamically changing environments. The system dynamic plays an important role in long-term performance, such as for the time-varying user requests and network status [17]. There is no guarantee for these one-shot optimization algorithms to reach the expected performance over a long period.

2.2 DRL for Computation Offloading

Fortunately, the recent breakthroughs of DRL provide alternative approaches to address the problems in one-shot optimization [18]. Alam *et al.* [19] proposed an autonomic deep Q-learning-based algorithm to minimize the network delay and power consumption in the distributed edge/fog network. Qi *et al.* [20] proposed a knowledge-driven framework to obtain the optimal policy for multi-type edge computing nodes. Baek *et al.* [21] developed an effective algorithm based on DRL to achieve load balancing and minimize the execution time under different service rates and task arrival rates. Jiang *et al.* [22] proposed a multi-agent DRL-based caching strategy to achieve efficient content caching in mobile device-to-device networks and introduced an upper confidence bound algorithm to solve the slow convergence caused by large action space.

However, taking advantage of DRL into computation offloading optimization problems is different from that of video games, which is not taken into account in the above work. The major differences are the limited experience and the actual costs of trial-and-error in computation offloading.

In this regard, the vast amount of experiences required for agent training and the significant costs of trial-and-error do not hold up for the DRL deployment in MEC computation offloading.

2.3 Distributed DRL Training

While it may seem unrealistic to directly apply DRL algorithms in computation offloading problems, it turns out to be more realistic on the consideration of a possible future where multiple independent computation offloading systems are deployed and collectively train their DRL agents. This falls within the domain of distributed DRL training, which has long been regarded as a hot research topic of DRL. Ong *et al.* [23] presented the first attempt to empower the deep Q-learning [5] with large-scale distributed training across thousands of machines. Mnih *et al.* [24] and Barth-Maron *et al.* [25] proposed the Asynchronous Advantage Actor-Critic (A3C) algorithm and the Distributed Distributional Deterministic Deep Policy Gradient (D4PG) algorithm, respectively, which are the state-of-the-art distributed version of policy gradient-based DRL algorithms.

Note that the previous algorithms are designed to scale out DRL training to multiple machines (including GPUs and CPUs) and maximize machine utilization. This may work well in scenarios where an instance of the environment can be created directly in memory, such as video game AI and neural architecture search. But for some scenarios, such as computation offloading, it has its limitations. For instance, each parameter update of A3C requires to accumulate the gradients of each distributed DRL agent, which imposes heavy communication overhead for real-world computation offloading systems. D4PG relies on frequent network weight replication and relatively short trajectories to avoid data biases, which is also not impractical for computation offloading scenarios. The pressing question is the mismatch between the original intention of the design of traditional distributed DRL algorithms and the requirements for computation offloading scenarios. This work, to the best of our knowledge, represents the first attempt to apply DRL in a more practical and robust way to solve the computation offloading problems.

3 System Model

We first introduce our system model. As illustrated in Fig. 1, we consider a computation offloading model for MEC, which consists of an edge server (ES) and a set of mobile users (MUs) that are connected to ES through base stations. MUs are distributed within a certain range and belong to the same community/organization, such as smart home and smart factory. Let $\mathbb{U} = \{1, 2, \ldots, U\}$ be the set of MUs and $\mathbb{M} = \{1, 2, \ldots, M\}$ be the set of wireless channels, through which MUs can transmit the required data for task execution to the base stations. Both MUs and ES have a certain amount of computing capabilities. To simulate a dynamically changing environment, which is common in real-life scenarios, we adopt a uniform task generation model [26] to characterize user requests. In this model, each task is characterized by its required data (including the program code and input data), the required CPU cycles to finish the task, and the task deadline. In addition, due to the fact that the returning data is generally much smaller than the input data, such as object recognition, we assume that the size of returning data can be ignored. In this work, we mainly focus on the challenges of applying DRL to computation offloading. Therefore, for simplicity, we consider a system model with a simple assumption, that is, MUs can only offload the applications as a whole or decompose the applications into several independent sub-tasks.

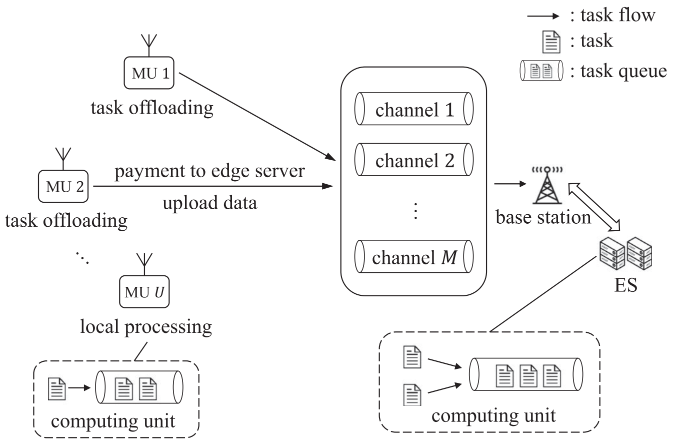


Fig. 1. An illustration of multi-user computation offloading and channel allocation model for MEC.

Without loss of generality, each MU has limited computation capacity, while ES can provide computational resources to support the requesting MUs. Therefore, to reduce the execution cost and time consumption, tasks can be offloaded to ES. As shown in Fig. 1, to offload tasks, MU 1 and MU 2 first upload the required data through the allocated wireless channels and pay to ES for the consumed computational resources. Then, the offloaded tasks are sent to the computing unit of ES, where each offloading task shares the same computational resources. Following the common methodology of studying the computation-offloading optimization problem, we assume that tasks are scheduled based on the First-In, First-Out (FIFO) principle. In this context, the task execution in ES can be modeled as a queuing model. Current tasks must wait in the task queue if the computational resources are occupied by other tasks. Similarly, each MU maintains a task queue for local processing tasks.

For tractable analysis, as with many other studies (e.g., [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]), we regard the computation offloading as a quasi-static process, that is, no MU will exit the MEC system abruptly within one computation offloading period, which is typically in a timescale of milliseconds [27]. In this work, we assume that the length of each period is the same for simplicity. Note that our work can be easily extended to scenarios with diverse period lengths. From the system model, it can be observed that if a large number of MUs choose to offload tasks, the network resources and computational resources of ES will be consumed so fast that there is no guarantee of finishing the subsequent tasks before the deadline. This seriously deteriorates the performance of MEC. Therefore, it is critical to design an algorithm that can optimize the allocation of the network and computational resources from a long-term perspective.

4 Problem Formulation

4.1 Decision-Making Description

As we mentioned above, we model the computation offloading as a quasi-static process. During each decision epoch, the offloading decisions are made based on the system state S, which consists of the brief information about the generated tasks, MUs, ES, and wireless channels. First, the generated tasks of MU i $(i \in \mathbb{U})$ are denoted as $G_i = (\nu_i, c_i, d_i)$, where ν_i is the size of the program code and input data, c_i is the number of CPU cycles required to finish the task, and d_i is the deadline of the task. Second, we assume current tasks must wait in the task queue if the computational resources are occupied by other tasks. Thus, we denote q_i as the queuing delay of MU i $(i \in \mathbb{U})$ and q_e as the queuing delay of ES. In addition, the computation capacities of MU i $(i \in \mathbb{U})$ and ES are denoted by f_i and f_e, respectively. For the wireless connection, we denote the channel gain and background noise power between MU i and base station as $g_{i,e}$ and $\varpi_{i,e}$, respectively. In this context, the system state can be defined as follows:

$$S = (\mathbb{G}, \mathbb{X}, \mathbb{C}, f_e, q_e). \tag{1}$$

Here, $\mathbb{G} = \{G_i | i \in \mathbb{U}\}$ is the collection of task information and $\mathbb{X} = \{(f_i, q_i, x_i, y_i) | i \in \mathbb{U}\}$ is the collection of MUs' states. The last two elements, x_i and y_i, indicate the coordinates of MU i. Further, $\mathbb{C} = \{(g_{i,e}, \varpi_{i,e}) | i \in \mathbb{U}\}$ denotes the wireless channel profile.

After obtaining the system state S, a joint action A for multiple MUs is made based on the observed system state. As we mentioned, it is necessary to jointly consider the allocation of network and computational resources. Thus, the action for MU $i (i \in \mathbb{U})$ can be represented as $a_i = (b_i, m_i)$, where $b_i \in \{0, 1\}$ and $m_i \in \mathbb{M}$ represent the offloading decision and the selected channel. Specifically, we have $b_i = 0$ if MU i chooses local processing. Likewise, we have $b_i = 1$ if MU i chooses to offload the current task through wireless channel m_i. Therefore, the joint action for multiple MUs can be defined as follows:

$$A = \{a_i | a_i = (b_i, m_i), i \in \mathbb{U}\}. \tag{2}$$

In the following, we will discuss the costs of choosing different actions. The main notations are summarized in Table 1 .

TABLE 1
List of Notation Definitions

Notation	Definition
$\mathbb{U}$	Set of MUs
$\mathbb{M}$	Set of wireless channels
G_i	Computational task of MU i
ν_i	Size of the required data of task G_i
c_i	Required CPU cycles of task G_i
d_i	Deadline of task G_i
q_i	Current queuing delay of MU i
q_e	Current queuing delay of ES
f_i	Computation capacity of MU i
f_e	Computation capacity of ES
(x_i, y_i)	Coordinates of MU i
β_i	Average energy cost per CPU cycle in MU i
β_e	Average energy cost per CPU cycle in ES
ζ	Unit price of energy
p_i	Transmission power of MU i
$g_{i,e}$	Channel gain between MU i and base station
$d_{i,e}$	Distance between MU i and base station
p_l	Path loss exponent
$\varpi_{i,e}$	Noise power between MU i and base station
$r_{i,e}$	Uplink data rate between MU i and base station
Θ	Penalty for task execution failure
C_i	Cost of MU i
C_{sys}	The overall system cost
S_t	System state at the decision epoch t
A_t	Action at the decision epoch t
$R(S_t, A_t)$	Immediate reward of state-action pair (S_t, A_t)
γ	Discount factor for future reward
ϕ, ϕ'	Parameters of actor-network and target actor-network
θ, θ'	Parameters of critic-network and target critic-network

4.2 Cost of Local Processing

For local processing, the generated task of MU i is sent to the computing unit of the local device. The time consumption of a task G_i depends on the computation capacity of the local device f_i, the required CPU cycles c_i, and the queuing delay of the current task queue q_i. We neglect the time for task initialization because it is typically small. Accordingly, the time consumption can be expressed as follows:

$$t_i^l = \frac{c_i}{f_i} + q_i. \tag{3}$$

In addition, the cost of energy consumption can be represented by the following:

$$e_i^l = c_i \beta_i \cdot \zeta. \tag{4}$$

Here, β_i is the average energy cost per CPU cycle in MU i, which is set according to the characteristics of computing devices. We apply the practical measurement in [28] and set $\beta_i = 10^{-27} (f_i)^2$. And ζ is the unit price of energy. In this model, each task should be completed by its deadline, otherwise, there will be a penalty, which is a pre-defined value and is typically far higher than the average cost of completing the task [29]. Therefore, the cost is determined by both the energy cost and time consumption. For MU i, the cost of local processing can be expressed as

$$C_i^l = \begin{cases} e_i^l & \text{if } t_i^l \le d_i \\ \Theta & \text{otherwise} \end{cases}, \tag{5}$$

where d_i is the task deadline and Θ is the fail penalty.

4.3 Cost of Edge Computing

Next, we analyze the cost of edge computing. To execute tasks in ES, MU i first needs to transmit the input data and program code with the size of ν_i to the ES in close proximity. Similar to many studies, the returning data is generally much smaller than the input data. Thus, we assume that the transmission time of returning data can be ignored. Consequently, the time consumption involves data uploading time and execution time. On one hand, the data uploading time is determined by the data size and transmission rate. In the model, multiple MUs share network resources to achieve efficient utilization. To accurately capture the average aggregate throughput of wireless links, we consider a wireless interference model as in [27], which is common for many channel access mechanisms, such as Code-Division Multiple Access (CDMA). Mathematically, the uplink data rate between MU $i (i \in \mathbb{U})$ and the base station can be calculated by

$$r_{i,e} = \omega \log_2 \left(1 + \frac{p_i g_{i,e}}{\varpi_{i,e} + \sum_{j \in \mathbb{U} \backslash \{i\}: a_j = a_i} p_j g_{j,e}} \right), \tag{6}$$

where ω is the channel bandwidth, p_i is the transmission power of MU i, and $\varpi_{i,e}$ is the background noise power. $g_{i,e} = d_{i,e}^{-p_l}$ is the channel gain between MU i and base station, where p_l is the path loss exponent and $d_{i,e}$ is the distance between MU i and base station. In addition, $\mathbb{U} \backslash \{i\} : a_j = a_i$ contains all the elements in set $\mathbb{U}$ that satisfy $a_j = a_i$ and are not in $\{i\}$. Given the above, the time consumption for MU i to upload data is as follows:

$$t_i^{e,tr} = \frac{v_i}{r_{i,e}}, \tag{7}$$

where v_i is the size of program code and input data. Here, we neglect the round-trip delay between base stations and ES because they are typically connected via fibre links and the time consumption is negligible [26].

After obtaining the data from MUs, ES performs the tasks with relatively powerful computation capacity. Similar to local processing, we neglect the task initialization time. We assume the incoming tasks are scheduled based on the FIFO principle. Thus, similar to local processing, the time consumption for MU i to complete the task can be expressed as

$$t_i^{e,com} = \frac{c_i}{f_e} + q_e, \tag{8}$$

where c_i is the total number of required CPU cycles, f_e and q_e are the computation capacity and queuing delay of ES, respectively. Following Equations (7) and (8), the total time consumption is the sum of the transmission time and computation time, which can be calculated by

$$t_i^e = t_i^{e,tr} + t_i^{e,com}. \tag{9}$$

For the energy cost of edge computing, it includes the payment to ES and the energy consumption of data transmission. Thus, we have

$$e_i^e = c_i \beta_e + t_i^{e,tr} p_i \cdot \zeta, \tag{10}$$

where β_e is the unit price for per CPU cycle in ES and is set according to the computation capacity of ES. In addition, p_i is the transmission power and ζ is the unit price of energy.

Lastly, similar to local processing, the tasks offloaded to ES should be completed by the deadline. For MU i, we define the cost of edge computing as

$$C_i^e = \begin{cases} e_i^e & \text{if } t_i^e \leq d_i \\ \Theta & \text{otherwise} \end{cases}. \tag{11}$$

4.4 MDP-Based Multi-User Computation Offloading

According to the system model, the actions taken by MUs are coupled and mutually affected. For instance, if too many MUs choose edge computing and upload their data through the same wireless channel simultaneously, severe interference may incur during data transmission, resulting in longer time to complete the tasks or even task failures. Meanwhile, intense competition for computational resources in ES can also lead to rapid resource consumption, long queuing time, and task failures. Based on this insight, it is important to consider the decision-making of multiple MUs simultaneously. As mentioned earlier, we consider a common scenario where all MUs belong to the same community/organization. Therefore, we formulate the multi-user computation offloading problem as a joint optimization problem, where the system cost can be defined as the sum of all MUs' costs

$$\begin{aligned} C_{\text{sys}} &= \sum_{i \in \mathbb{U}} C_i \\ &= \sum_{i \in \mathbb{U}} (1 - b_i) C_i^l + b_i C_i^e. \end{aligned} \tag{12}$$

Different from many previous studies (e.g., [12], [13], [14], [15], [16]), we are committed to minimizing the long-term cost of the MEC system to adapt to the dynamic environment and achieve better performance. For tractable analysis, we discretize the time horizon into multiple time steps (or decision epochs) and denote $\mathbb{T} = \{0, 1, \ldots, t, \ldots, T\}$ as a sequence of discrete decision epochs, where T may be ∞. At the beginning of a decision epoch t, a corresponding offloading decision A_t is made based on the current system state S_t. Next, we track the performance of the selected state-action pair (S_t, A_t) with the immediate system cost C_{sys}. In addition, the next state S_{t+1} is only determined by the current state S_t and the current action A_t. In this context, we define a tuple $< \mathbb{S}, \mathbb{A}, P, R, \gamma >$ and formulate the multi-user optimization problem as an MDP, which is widely used to address sequential stochastic decision-making problems. Here, $\mathbb{S}$ and $\mathbb{A}$ represent the state space and action space, respectively. $P : \mathbb{S} \times \mathbb{A} \rightarrow \Delta(\mathbb{S})$ represents the state transition probability, mapping each state-action pair to the transition probability to the next state over $\mathbb{S}$. In addition, $R : \mathbb{S} \times \mathbb{A} \rightarrow [R_{\min}, R_{\max}]$ is the immediate reward function, which is equal to $-C_{\text{sys}}$ in our model. Lastly, $\gamma \in (0, 1]$ is the discount factor that determines the weights of future rewards.

In an MDP model, the objective is to find the optimal policy π^* that maximizes the total discounted reward (or minimizes the total discounted cost) for an infinite/finite time horizon. In other words, the immediate reward may be smaller, but the optimal policy π^* leads to more profitable rewards from a long-term point of view. As such, the considerations we discussed above are automatically considered in an MDP model. Accordingly, we define the optimal policy as the following:

$$\pi^* = \arg \max_{\pi} \mathbb{E}_{\pi, S} \left[\sum_{t \in \mathbb{T}} R(S_t, \pi(S_t)) \right]. \tag{13}$$

As the long-term performance optimization problem can be formulated as an MDP, the optimal policy π^* can be

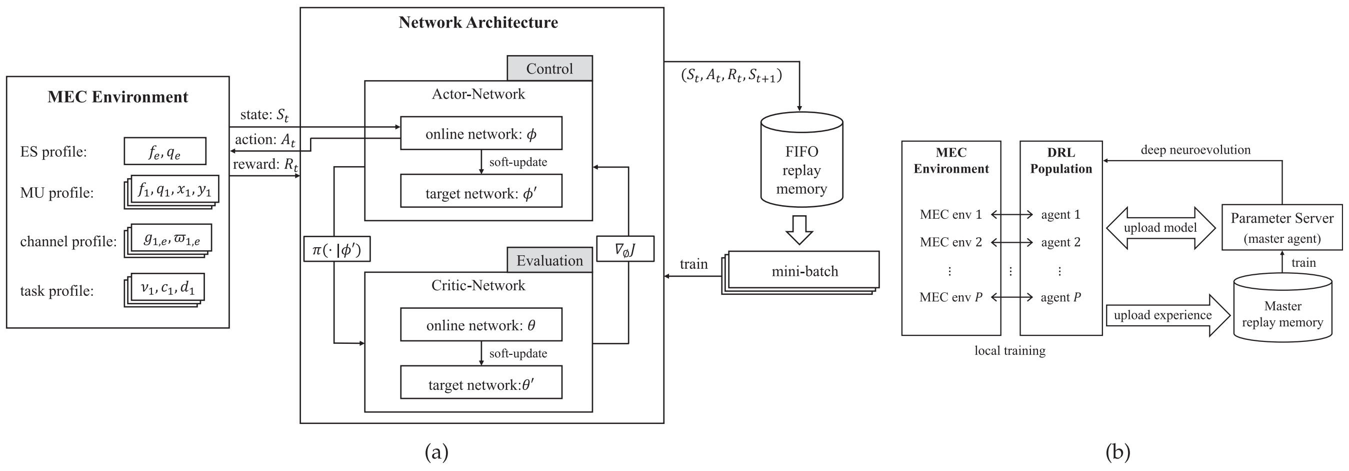


Fig. 2. (a) The actor-critic framework of the DC-DRL algorithm for each independent MEC environment. (b) A high-level schematic of the distributed and collective training scheme of the DC-DRL algorithm. In such a scheme, the training of the distributed DRL population is decoupled from the training of master DRL agent. Multiple MEC environments and DRL agents collect experiences concurrently to promote maximum utilization.

obtained by value iteration. However, in multi-user MEC, the state space and action space grow exponentially as the number of MUs increases, making it infeasible to solve the optimization problem in polynomial time. It has been proven that a multi-user computation offloading problem involving offloading decision-making and channel allocation is NP-hard [27]. In addition, the problem is further complicated by the effect of current action on the future reward and the unawareness of state transition probability.

5 Algorithm Design

To tackle the above issues, we leverage the experience-driven DRL and propose the DC-DRL algorithm based on the actor-critic framework. In particular, from a practical perspective, we consider to use multiple MEC environments to obtain knowledge beyond the individual and spread the exploration costs. Considering the "policy gap" caused by distributed training, we propose the adaptive n-step learning, which can improve training efficiency without affecting the convergence stability. In addition, we combine the advantages of deep neuroevolution and policy gradient to maximize the utilization of multiple MEC environments.

5.1 Actor-Critic Framework

In this work, we develop the DC-DRL algorithm from the actor-critic framework [30], which is the fundamental of many start-of-the-art DRL algorithms, from Deep Deterministic Policy Gradient (DDPG) [31] to Proximal Policy Optimization (PPO) [32]. We first introduce the DC-DRL agent deployed in each independent MEC environment, and how the DC-DRL exploits the actor-critic framework to deal with the computation offloading problem.

As illustrated in Fig. 2a, the network architecture of the actor-critic framework consists of two parts: an actor-network for control and a critic-network for evaluation. The actor-network is used to predict the optimal action given the system state S_t at current decision epoch t. In the proposed multi-user MEC model, the system state S_t contains the profiles of ES, MUs, wireless channels, and the generated tasks. According to the current policy and system state, the actor-network outputs an action, which consists of the computation offloading and channel allocation decisions for all requesting MUs. In addition, to explore the optimal strategy, a noise derived from the Ornstein-Uhlenbeck process is added to the original action output as in [31], which is a stationary Gauss-Markov process and is temporally homogeneous. By this point, a noisy action A_t is obtained and applied to the MEC environment.

Next, after the MEC environment takes the noisy action A_t, the actor-critic agent obtains the immediate reward R_t and the next observed state S_{t+1}. Here, the immediate reward R_t is equal to the negative value of system cost, namely $-C_{\text{sys}}$, which reflects the performance of the action taken. After the interaction, we have an experience tuple, (S_t, A_t, R_t, S_{t+1}), which will be stored in the replay memory for parameter update. In particular, we organize the replay memory in a FIFO method. The key idea underlying this is to facilitate the implementation of the adaptive n-step learning, which is analyzed in detail in Section 5.3. Algorithm 1 summarizes the interactions between each independent environment and DRL agent.

The parameters of the actor-network and critic-network are updated through periodic training. At each training iteration, we randomly select samples from the replay memory to update the network parameters. For a state-action pair (S_t, A_t), the critic-network attempts to map to the corresponding expected long-term reward following current policy π, which is also referred to as Q-value, where "Q" stands for quality. Hence, the Q-value can be expressed as

$$Q^{\pi}(S_t, A_t) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k}\right], \tag{14}$$

where $(S_t, A_t, R_t, S_{t+1}, A_{t+1}, R_{t+1}, \ldots)$ is a complete trajectory following policy π. However, the complete trajectory is generally unreachable beforehand for a MEC system. In consequence, many DRL algorithms use one-step Temporal Difference (TD) learning to approximate the long-term reward in Equation (14). By applying the Bellman operator, we have

$$\begin{aligned}\mathcal{T}Q(S_t, A_t|\theta) = {} & R(S_t,A_t) \\ & + \gamma \cdot \mathbb{E}[Q(S_{t+1}, \pi(S_{t+1}|\phi)|\theta)].\end{aligned} \tag{15}$$

Here, $R(S_t, A_t)$ is the immediate reward of state-action pair (S_t, A_t) and is equal to the R_t in Equation (14). $\pi(S_{t+1}|\phi)$ denotes the output of the actor-network given input S_{t+1}, where ϕ denotes the parameters of the actor-network. In addition, $Q(S_{t+1}, \cdot|\theta)$ denotes the estimated long-term reward of the state-action pair $(S_{t+1}, \cdot)$ using the critic-network, where θ denotes the network parameters. In this work, we use the Mean-Squared Error (MSE) as the loss function. The resulting loss function used to update the critic-network can be written as

$$L(\theta) = \mathbb{E}\Big[\big(Q(S_t, A_t|\theta) - \mathcal{T}Q(S_t, A_t|\theta)\big)^2\Big]. \tag{16}$$

For the actor-network, according to the deterministic policy gradient theorem [31], the parameters are updated with the policy gradient from the backpropagation of value function (i.e., critic-network). By applying the chain rule of the expected long-term reward, we update the parameters of the actor-network with the following equation:

$$\begin{aligned}\nabla_\phi J &\approx \mathbb{E}\Big[\nabla_\phi Q(S, A|\theta)|_{S=S_t, A=\pi(S_t|\phi)}\Big] \\ &= \mathbb{E}\Big[\nabla_A Q(S, A|\theta)|_{S=S_t, A=\pi(S_t|\phi)} \cdot \nabla_\phi \pi(S|\phi)|_{S=S_t}\Big],\end{aligned} \tag{17}$$

where ∇_ϕ and ∇_A denote the gradient vectors with respect to ϕ and A, respectively. During the entire training process, the critic-network constantly improves its estimation of the Q-value for any given state-action pair (S, A) by minimizing Equation (16), and the actor-network constantly improves its strategy based on the gradient of the critic-network $\nabla_\phi J$. By periodically updating network parameters, the actor-network eventually converges to the optimal policy.

In addition, two significant improvements are integrated into the actor-critic framework. On one hand, the DNN structure of the actor-network and critic-network manifests instability because it is a non-linear function approximator. To avoid the oscillation of DNN and stabilize the convergence, we adopt additional DNNs for both the actor-network and critic-network, which have the same structure as the original networks and are used to provide relatively stable reference values [5]. For clarity, we refer to the original network as the online network, and the additional network as the target network. In this way, the expected long-term reward in Equation (15) can be rewritten as

$$\begin{aligned}\mathcal{T}Q(S_t, A_t|\theta) = R(S_t, A_t) \\ + \gamma \cdot \mathbb{E}[Q(S_{t+1}, \pi(S_{t+1}|\phi')|\theta')],\end{aligned} \tag{18}$$

where ϕ' and θ' denote the parameters of target actor-network and target critic-network. The application of target networks ensures the Q-values of state-action pairs are fairly evaluated and stabilizes the convergence. To achieve this, the parameters of target networks are constrained to fluctuate slightly during the update of the original online networks. In practice, they slowly track the learned online networks with the following equations:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta', \tau \ll 1, \tag{19}$$

$$\phi' \leftarrow \tau\phi + (1-\tau)\phi', \tau \ll 1. \tag{20}$$

The other improvement is the use of experience replay, which is the mainstay technique in off-policy DRL algorithms. After each decision epoch, the interaction (i.e., experience) is stored in a FIFO replay memory. Then, at each training iteration, the DRL agent updates its parameters by randomly selecting a mini-batch of experiences rather than using the samples just collected. There are two advantages to using the experience replay. On one hand, it breaks the strong correlation in the trajectory of MDP and provides an accurate gradient estimation. On the other hand, data stored in the replay memory can be reused, greatly improving the sample efficiency and empowering DRL agents with strong learning capacity. This is critical for applying DRL algorithms to many real-world problems, which is generally limited in experience, such as computation offloading optimization.

Algorithm 1. Interaction (in Each Local Environment)

```
1: for $t = 0, \ldots, \infty$ do
2:     DRL agent takes system state $S_t$ as input and outputs a noisy action $A_t$.
3:     The environment executes action $A_t$.
4:     Observe the immediate reward $R_t$ and the next state $S_{t+1}$.
5:     Store $(S_t, A_t, R_t, S_{t+1})$ in the replay memory based on the FIFO principle.
6: end for
```

5.2 Distributed Collective Learning With Experience-Sharing

The actor-critic-based DRL algorithm shows promising prospects in long-term performance optimization, yet it suffers from slow learning speeds and huge exploration costs, especially for MDPs with high-dimensional state/action space. Therefore, the DC-DRL considers leveraging multiple MEC environments to obtain knowledge beyond the individual and spread the costs across multiple independent environments. Based on the characteristics of the DRL algorithms and MEC systems, we summarize the three requirements of a distributed collective learning scheme should fulfil:

1) It should be communication-efficient and not affect the operation of MEC systems.
2) It should be robust against non-Independent and Identically Distributed (non-IID) data distribution.
3) It should be highly scalable to support a large number of MEC environments for collective training.

In the broader realm of distributed DRL, a wide array of training schemes are proposed in recent years, which can be divided into three categories: gradient-sharing, parameter-sharing, and experience-sharing. The experience-sharing means DRL agents share their data stored in the replay memory. In the following, we will analyze the characteristics of these three training schemes and demonstrate their respective feasibilities.

5.2.1 *Communication Overhead*

From the hierarchical architecture shown in Fig. 2b, the additional communication overhead arises from the master agent in the parameter server accessing the distributed agents and exploiting their knowledge. In practice, the amount of communication overhead is determined by the size and frequency of the data transmitted. For the parameter-sharing scheme, after several rounds of training over their local datasets, distributed agents upload all the parameters of DNNs (whose size ranges from several megabytes to gigabytes) to the parameter server for model aggregation, as in [33]. In effect, parameter-sharing is a communication-efficient training scheme because the model aggregation can proceed at a relatively low frequency. For the gradient-sharing scheme, despite the amount of data per transmission is small, it still brings a huge communication overhead because the transmission interval is typically at a time scale of milliseconds, as the A3C algorithm in [24]. Such highly frequent data transmission constraints the gradient-sharing scheme to scenarios where an instance of the environment can be created directly in memory, such as video games. For the experience-sharing scheme, uploading an experience tuple to the parameter server typically takes much less time than generating an experience for a MEC system, which typically includes transmitting the program code and input data, queuing, and task execution. In addition, the experience-sharing scheme considerably improves the sample efficiency by experience replay. Therefore, the experience-sharing scheme is applicable to the deployment of DRL agents without affecting the operation of MEC systems.

Algorithm 2. DRL Training

```
Require: replay memory capacity $D$, batch size $N$, actor-network learning rate $\alpha_a$, critic-network learning rate $\alpha_c$, soft-update parameter $\tau$, upload interval $j_{\text{upload}}$, local training times $\Upsilon$, agent id $p$.
1: Initialize online actor-network and online critic-network with random network parameters $\phi$ and $\theta$.
2: Initialize target actor-network and target critic-network with network parameters $\phi' \leftarrow \phi$ and $\theta' \leftarrow \theta$.
3: for $g = 1, \ldots \infty$ do
4:     Apply additive Gaussian noises with mutation strength to network parameters.
5:     for $j = 1, \ldots, \Upsilon$ do
6:         Randomly select $N$ transitions with length $n_{\max}$.
7:         Compute the adaptive n-step return using Equation (22).
8:         Compute the loss values and gradient vectors using Equations (21), (16), and (17).
9:         Update the network parameters of the online actor-network and online critic-network with the computed loss values and gradient vectors.
10:        Soft-update the parameters of the target networks using Equations (19) and (20).
11:        if j mod $j_{\text{upload}} = 0$ and $p \neq 0$ then
12:            Upload the collected experience batch to the master agent for experience-sharing.
13:        end if
14:    end for
15:    if $p = 0$ then
16:        Calculate $h_{g+1}$ and $f_g^{\text{avg}}$ using Equations (25) and (26).
17:    else
18:        Obtain $h_{g+1}$ and $f_g^{\text{avg}}$ from the master agent.
19:    end if
20:    if $F(h_g^p) < f_g^{\text{avg}}$ then
21:        Replace the network parameters with $h_{g+1}$.
22:    end if
23: end for
```

TABLE 2
Comparison of Different Distributed Training Schemes

	Parameter	Gradient	Experience
Size	large	small	medium
Frequency	low	extremely high	medium
Robustness	low	high	high
Scalability	low	low	high

5.2.2 Robustness Against Non-IID Data

In a MEC system, because the training data collected by a distributed DRL agent is derived from the interactions with its local environment, the distribution and pattern of local datasets vary strongly among different distributed DRL agents. More concretely, the local datasets are typically based on the characteristics of MU devices and behavior patterns. As a consequence, the local dataset of a distributed environment does not represent the true data distribution [34]. It can be observed that the gradient-sharing and experience-sharing schemes possess high robustness against non-IID data. This is because the former can be approximated as training on IID data through highly frequent data transmission [35], and the latter frames a dataset that can represents the true data distribution by sharing the experience. However, the parameter-sharing scheme is sensitive to non-IID data, which conducts several rounds of training on the model locally before uploading the model to the parameter server. Due to the skewness of data distribution, the weight divergence is inevitably [36], constraining it to scenarios with IID (or near-IID) data.

5.2.3 Scalability

In this work, we envision a bright future where a massive number of MEC environments participate in the training of DRL agents. The challenge in achieving such a vision lies in the scalability of the system. For the parameter-sharing scheme, different training individuals have varying computation capacities and data generation rates, result in huge differences in learning progress. The parameters of some resource-constrained training individuals may be outdated, thereby aggregating these parameters may instead degrade the performance. The synchronization of distributed DRL agents is an obstacle to the parameter-sharing scheme in practice [33]. For the gradient-sharing scheme, it is thorny to handle the high communication overhead brought by massive training participants. Individuals lacking communication resources are the potential bottleneck to maximize the utilization of distributed environments [36]. Fortunately, for the experience-sharing scheme, since the training of the master agent is based on the aggregated experience, not the parameters of distributed DRL agents, there is no outdated issue as in the parameter-sharing scheme. In addition, the training process and the experience collection process can be easily implemented asynchronously, thus eliminating potential communication bottlenecks as in the gradient-sharing scheme.

According to the above analysis, we observe that for the MEC system we considered, the experience-sharing scheme is superior to other distributed learning schemes in that it is communication-efficient, robust against non-IID data, and highly scalable to support a large number of training participants. Table 2 gives a comparison of different distributed training schemes. Following this, we design the DC-DRL algorithm based on the experience-sharing scheme. Noted that in the implementation of the DC-DRL algorithm, we combined the experience-sharing and parameter-sharing schemes to further enhance the DC-DRL algorithm, which is explained in detail in Section 5.4.

5.3 Adaptive N-Step Learning

Subsequently, we introduce the adaptive n-step learning, which is a modification of the long-term reward estimate in DRL training. As shown in Equation (18), the standard actor-critic framework adopts one-step TD-learning to approximate the long-term reward. However, it inevitably leads to high bias and low efficiency in the early stages of training because $Q(S_{t+1}, \pi(S_{t+1}|\phi')|\theta')$ in Equation (18) is an imprecise estimate of the future reward. Given the impracticality of obtaining the complete trajectories in real-world MEC environments, it is possible to reduce the bias by leveraging the information of multiple steps as a compromise. Motivated by this, we consider the n-step TD methods in [25] to obtain a better estimate of the long-term reward. In contrast with Equation (18), the parameters are updated with n-step return, where the update target can be represented as

$$\begin{aligned}\mathcal{T}Q(S_t, A_t|\theta) = &R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots + \gamma^{n-1} R_{t+n-1}\\ &\gamma^n \cdot \mathbb{E}[Q(S_{t+n}, \pi(S_{t+n}|\phi')|\theta')].\end{aligned} \tag{21}$$

Here, $(S_t, A_t, R_t, S_{t+1}, A_{t+1}, R_{t+1}, \ldots, S_{t+n}, A_{t+n}, R_{t+n})$ is a truncated trajectory stored in the replay memory, and n is a preset value.

However, empirical studies show that n-step TD methods exhibit great limitations in the off-policy DRL algorithms, such as the distributed settings considered in this work. The reason is that the selected truncated trajectories are collected with the previous policy, which may greatly deviate from the policy to be updated. As a result, directly introducing n-step TD methods in the distributed settings greatly suffers from high variance. Therefore, it is critical to correct the "policy gap" when updating parameters with the experiences collected from distributed DRL agents.

To address the above challenge, we follow the idea of the off-policy correction [10] and propose a method that can dynamically adjust the value of n based on the policy deviation, namely adaptive n-step learning. At the initialization process, the maximum value of n is set as $n_{\max}$. At each training iteration, trajectories of length $n_{\max}$ are selected to update parameters. Assume $(S_t, A_t, R_t, S_{t+1}, A_{t+1}, R_{t+1}, \ldots, S_{t+n_{\max}}, A_{t+n_{\max}}, R_{t+n_{\max}})$ is a selected trajectory. Following the idea of off-policy correction [10], it is straightforward to set n as follows:

$$n = 1 + \sum_{k=t+1}^{t+n_{\max}} \prod_{j=t+1}^{k} \mathbf{1}_{\{A_j = \pi(S_j|\phi')\}}, \tag{22}$$

where $\mathbf{1}_{\{A_j = \pi(S_j|\phi')\}}$ is the indicator function, which equals to 1 if the condition $A_j = \pi(S_j|\phi')$ satisfies. In this way, the DC-DRL algorithm can update parameters with corrected trajectories, which improves sample efficiency without introducing high variance. In the following, we provide theoretical analysis for the policy improvement of the adaptive n-step learning.

Theorem 1. *Using adaptive n-step learning, the worst error of the new long-term reward estimate is less than or equal to the worst error of the old estimate, that is,*

$$\begin{aligned}\max_{S_t, A_t} \Big| \mathbb{E}[Q'(S_t, A_t)] - Q^*(S_t, A_t) \Big| \\ \leq \max_{S_t, A_t} \Big| \mathbb{E}[Q(S_t, A_t)] - Q^*(S_t, A_t) \Big|,\end{aligned}$$

where $Q(S_t, A_t)$, $Q'(S_t, A_t)$, and $Q^*(S_t, A_t)$ stand for the old, new, and unbiased estimates of the long-term reward of state-action pair (S_t, A_t), respectively.

Proof. According to the definition of $Q^*(S_t, A_t)$, we have

$$Q^*(S_t, A_t) = \sum_{k=0}^{\infty} \gamma^k R(S_{t+k}, \pi(S_{t+k}|\phi')),$$

where $\pi(\cdot|\phi')$ denotes the current policy to be updated and $R(\cdot,\cdot)$ denotes the immediate reward function. Intuitively, $Q^*(S_t, A_t)$ is the cumulative discounted reward of an undeviated trajectory.

On the other hand, the new long-term reward estimate for state-action pair (S_t, A_t) can be represented as

$$Q'(S_t, A_t) = R_{t:t+n} + \gamma^n Q(S_{t+n}, A_{t+n}).$$

For simplicity, we define

$$R_{t:t+n} = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots + \gamma^{n-1} R_{t+n-1}.$$

From the setting of adaptive n-step learning and Equation (21), we can see that the selected truncated trajectory of length n is undeviated. In this context, the following equations hold:

$$\begin{aligned} R_{t+1} &= R(S_{t+1}, \pi(S_{t+1}|\phi')),\\ R_{t+2} &= R(S_{t+2}, \pi(S_{t+2}|\phi')),\\ &\vdots\\ R_{t+n-1} &= R(S_{t+n-1}, \pi(S_{t+n-1}|\phi')).\end{aligned}$$

Therefore, we can proceed by extending $Q'(S_t, A_t)$ and $Q^*(S_t, A_t)$

$$
\begin{aligned}
&\max_{S_t,A_t}\left|\mathbb{E}[Q'(S_t,A_t)]-Q^*(S_t,A_t)\right|\\
=&\max_{S_t,A_t}\left|\mathbb{E}[R_{t:t+n}+\gamma^n Q(S_{t+n},A_{t+n})]-Q^*(S_t,A_t)\right|\\
=&\max_{S_t,A_t}\Big|\mathbb{E}[R_{t:t+n}+\gamma^n Q(S_{t+n},A_{t+n})]\\
&-R_{t:t+n}-\gamma^n\sum_{k=0}^{\infty}\gamma^k R(S_{t+n+k},\pi(S_{t+n+k}|\phi'))\Big|\\
=&\max_{S_t,A_t}\Big|\mathbb{E}[Q(S_{t+n},A_{t+n})]\\
&\qquad-\gamma^n\sum_{k=0}^{\infty}\gamma^k R(S_{t+n+k},\pi(S_{t+n+k}|\phi'))\Big|\\
=&\gamma^n\max_{S_t,A_t}\left|\mathbb{E}[Q(S_{t+n},A_{t+n})]-Q^*(S_{t+n},A_{t+n})\right|.
\end{aligned}
$$

Intuitively, the left-side of Theorem 1 involves searching a state-action pair (S_t, A_t) that satisfies the condition: following the current policy, the state-action pair after its n steps maximizes the function $|\mathbb{E}[Q(\cdot)]-Q^*(\cdot)|$. In this way, the domain of function $|\mathbb{E}[Q(\cdot)]-Q^*(\cdot)|$ is the subspace of $\mathbb{S}\times\mathbb{A}$. In view of this, it is easy to derive that

$$
\begin{aligned}
&\gamma^n\max_{S_t,A_t}\left|\mathbb{E}[Q(S_{t+n},A_{t+n})]-Q^*(S_{t+n},A_{t+n})\right|\\
\leq&\gamma^n\max_{S_t,A_t}\left|\mathbb{E}[Q(S_t,A_t)]-Q^*(S_t,A_t)\right|.
\end{aligned}
$$

Because γ is a discounted factor that satisfies $\gamma\in(0,1]$, the inequality in Theorem 1 holds. This completes the proof. □

Theorem 1 underpins the policy improvement property of our proposed algorithm. It is safe to say that the adaptive n-step learning updates the parameters towards a better estimate concerning the long-term reward.

5.4 Deep Neuroevolution and Policy Gradient

5.4.1 Motivation

Although the gradient descent method is effective in the parameter optimization of many deep learning algorithms, it has not achieved the expected performance when applied to DRL algorithms. The reason in large part lies in the characteristics of DRL training. More concretely, a DRL agent updates the parameters of the actor-network and critic-network by calculating Equations (17) and (16), respectively. However, as mentioned above, we use a recursive method to approximate the expected long-term reward in Equation (14). This is because the actual gradients for DRL training are untraceable without knowing the complete trajectories. As a compromise, we apply the surrogate gradients in Equations (17) and (16), which is in contrast with deep learning algorithms where the true gradients can be easily obtained by using labeled data. Due to the inaccuracy of surrogate gradients in DRL, the traditional gradient-based optimization methods exhibit brittle convergence property and even divergence [37].

In addition, traditional gradient-based DRL algorithms are notorious for necessitating meticulous hyper-parameter tuning. In practice, a DRL agent searches for the optimal parameters through trial-and-error, which is typically in a time-consuming and expensive fashion. In the situation of high-dimensional parameter space, traditional trial-and-error methods may be insufficient to compensate for the poor convergence characteristics of gradient-based algorithms, which makes them liable to fall into local optima.

On the whole, traditional distributed DRL algorithms tend to focus on using multiple environments to increase the training speed, but fail to fully exploit the advantages of their diversities. In particular, DRL agents distributed in multiple environments can apply varying settings in terms of parameters and exploration strategies, contributing to radically different exploration trajectories and larger explored space. A pioneering work was presented in [11], where researchers at OpenAI explored the application of evolutionary strategy [38] in parameter optimization and further proposed the deep neuroevolution as a competitive alternative to policy gradient. Instead of perturbing the action output of DNNs, the deep neuroevolution algorithm directly adds noise to network parameters and calculates the randomized finite differences, thereby addressing the convergence issues of gradient-based methods.

However, extensive experiments in [11] suggest that simply applying deep neuroevolution is computationally expensive. For instance, it takes 1,440 CPU cores for the deep neuroevolution to compress the training time to 10 minutes for a 3D Humanoid walking task,[1] while a state-of-the-art gradient-based DRL algorithm only requires one CPU core and approximate one day to achieve a strategy with slightly poor performance. This is because the deep neuroevolution is based on randomness, which overcomes the limitation of gradient descent-based algorithms but inevitably leads to inefficiencies. Based on this insight, we proposed to combine the advantages of deep neuroevolution and policy gradient in this work.

5.4.2 Combining Deep Neuroevolution With Policy Gradient

As shown in Fig. 2b, the DC-DRL algorithm maintains a population with of P distributed MEC environments and DRL agents. For convenience, we define $h=\{\theta,\theta',\phi,\phi'\}$, which denotes all the network parameters of a DRL agent. To apply deep neuroevolution, we divide the entire training process into multiple periods, which are called generations. Accordingly, we denote the parameters of DRL population and master agent at the gth generation as $\mathbb{H}=\{h_g^1,\ldots,h_g^p,\ldots,h_g^P\}$ and h_g^0, respectively, where the superscript p is the agent id, which is used to distinguish different individuals (namely DRL agents) in the population, and P is the size of population. $p=0$ means the master agent. $\mathbb{H}$ and h_g^0 are randomly initialized at the initialization process. For the sake of clarity, we assume the size of the population remains unchanged. Nevertheless, it should be noted that the DC-DRL algorithm can be directly applied to systems with a dynamic P.

At the beginning of generation g, the DRL population and master agent draw samples from a multivariate

1. A famous open-source library released by OpenAI, https://gym.openai.com/

Gaussian distribution, namely $h \sim \mathcal{N}(h, \sigma^2\mathbf{I})$, where σ is the mutation strength. In effect, we reformulate this sampling process by applying additive Gaussian noises to the original parameters

$$h_g^{p'} = h_g^p + \sigma\varepsilon^p, p \in \{0, 1, 2, \ldots, P\}, \tag{23}$$

where $h_g^{p'}$ indicates the temporary parameters after adding noise and ε^p follows a Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$. Next, the DRL population and master agent update their parameters based on the interactions with the local environments and policy gradient methods. We assume that after Υ rounds of training, the cumulative gradient of $h_g^{p'}$ is ψ^p. Thus, the new parameters can be represented as

$$\begin{aligned} h_g^{p''} &= h_g^{p'} + \alpha^p\psi^p \\ &= h_g^p + \sigma\varepsilon^p + \alpha^p\psi^p, \end{aligned} \tag{24}$$

where $p \in \{0, 1, 2, \ldots, P\}$ and α^p is the local learning rate of agent p.

Following the idea of deep neuroevolution, we evaluate the fitness values of the candidate solutions $\{h_g^{p''} | p \in \{0, 1, 2, \ldots, P\}\}$. As we focus on a DRL problem, the fitness value of $h_g^{p''}$ is set as the average long-term reward following policy $h_g^{p''}$, which can be calculated during the previous training process.

Then, the master agent can select the best k solutions to update the parameters of the next generation. Let $F_g = \{F(h_g^{p''}) | p \in \mathbb{K}\}$ denote the fitness values of the best k solutions, where F is the fitness function acting on parameters and $\mathbb{K}$ is a set containing indexes of the best k solutions. In this context, the master agent uses the following formula to update the distribution mean of the next generation parameters

$$h_{g+1} = \sum_{p\in\mathbb{K}} \eta_g^{p''} \cdot h_g^{p''}. \tag{25}$$

Here, $\eta_g^{p''}$ is the weighting parameter that satisfies: (1) $\sum_{p\in\mathbb{K}} \eta_g^{p''} = 1$; (2) $\eta_g^{p''} \propto F(h_g^{p''})$. In this way, the DRL population places more emphasis on parameters with higher performance. In the end, individuals with fitness values lower than f_g^{avg} in the population are discarded by replacing their parameters with h_{g+1}, where

$$f_g^{\text{avg}} = \sum_{p=0}^{P} \frac{F(h_g^{p''})}{P+1}. \tag{26}$$

Algorithm 2 presents the pseudocode for DC-DRL's training procedure. DC-DRL first applies additive Gaussian noises to network parameters with a certain possibility (line 4). Then, DC-DRL performs policy gradient to update network parameters and uploads the collected experiences (line 6-12). At the end of each generation, the master agent uses Equation (25) to update the distribution mean of the next generation parameters (line 16).

5.4.3 *Analysis on Parameter Update*

Theorem 2. *The parameter update of the DC-DRL algorithm can be regarded as a special case of the deep neuroevolution.*

Proof. For comparison, we review a pioneering work in [11], where the deep neuroevolution is proposed as a competitive alternative to policy gradient. In [11], the parameters are updated based on the Gaussian perturbation. To be specific, the deep neuroevolution takes gradient steps with the following function estimator:

$$\nabla_{h_g}\mathbb{E}[F(h)] \approx \sum_{p\in\mathbb{K}} \frac{F(h_g^{p''})}{\sigma^2 \cdot k}(h_g^{p''} - h_g), \tag{27}$$

where $h_g^{p''}$ is the Gaussian-perturbed parameters.

On the other hand, because $\sum_{p\in\mathbb{K}} \eta_g^{p''} = 1$, Equation (25) is equal to

$$h_{g+1} - h_g = \sum_{p\in\mathbb{K}} \eta_g^{p''}(h_g^{p''} - h_g). \tag{28}$$

Note that σ and k are constants, and $\eta_g^{p''}$ is proportional to $\propto F(h_g^{p''})$. By comparing Equations (27) and (28), we see that the $\frac{F(h_g^{p''})}{\sigma^2 \cdot k}$ can be considered as the weighting parameters in Equation (25). Therefore, the parameter update of the DC-DRL algorithm can be regarded as a special case of the deep neuroevolution. □

In fact, the deep neuroevolution is interpreted as calculating a finite difference derivative estimate in a random direction [11]. Based on this insight and Theorem 2, the parameter update of the DC-DRL algorithm can be interpreted as a finite difference derivative estimate in a direction where Gaussian perturbation and policy gradient act together. In practice, the combination of deep neuroevolution and policy gradient uses the parameter-sharing and experience-sharing schemes at the same time, which empowers the DC-DRL algorithm with stronger exploration ability and higher convergence rate. This will be proved by the evaluations in Section 6.5.

6 Numerical Results

6.1 Evaluation Setup

In this section, we use both simulation environments and a real testbed to evaluate the performance of our proposed algorithm. For simulations, we set the environmental parameters according to the history log of a face recognition application[2] and [27], [39], [40]. The default parameters are as follows, if not specified. We design the parameters to simulate computation-intensive and deadline-sensitive situations. The simulations are run under scenarios where the coordinates of base stations and ES are fixed. There are 3 base stations connecting MUs with ES, each with a coverage range of 80 m. In addition, 30 MUs are randomly distributed within the communication coverage of the MEC system. For data transmission, the channel number M is 5, the transmit power p_t is 0.1 W, the channel bandwidth ω is 4 MHz, the noise power $\varpi_{i,e}$ is –100 dBm, the path loss exponent p_l is 4, and the price per energy unit is 0.1.

For the computation tasks, we set the task arrival rate constant at an average of 20 tasks per second to simulate the

2. Source: https://github.com/ageitgey/face_recognition

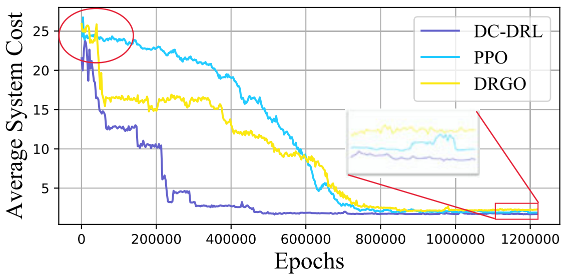
user requests. The size of input data and the number of required CPU cycles are sampled from the history log of a face recognition application, which ranges from [471, 6583] KB and [49, 1123] Megacycles, respectively. The unit price of CPU cycle in ES is 0.05 unit/Gcycle. For the computation capacities, we assume that ES is equipped with a 32-core CPU, where the CPU frequency of each core is 3.0 GHz. In addition, we assume each MU is equipped with a 4-core CPU, whose frequency is 1.2 GHz. Lastly, we set d_i and Θ as 2 and 1, respectively.

For the design of DC-DRL agent, we set the number of distributed DRL agents P as 20, the upload interval j_{upload} as 100, and the local training times Υ as 100. Each distributed DRL agent has the same DNN structure. In particular, both the actor-network and critic-network have two hidden layers, with $40 \times U$ neurons and $30 \times U$ neurons, respectively. We use the rectified linear unit as the activation function in each DNN and tanh as the output function of each actor-network. The maximum capacities of replay memory in each distributed agent and master agent are 5,000 and 100,000, respectively. The batch size for DRL training is 64. The initial learning rates of the critic-network and actor-network are 0.001 and 0.0001, respectively. The discount rate for future rewards is 0.9. For the soft-update of target networks, we set the τ as 0.001. Lastly, the Adam optimizer is used to optimize the neural networks. Our code is available at: github.com/qiuxy23/DC-DRL.

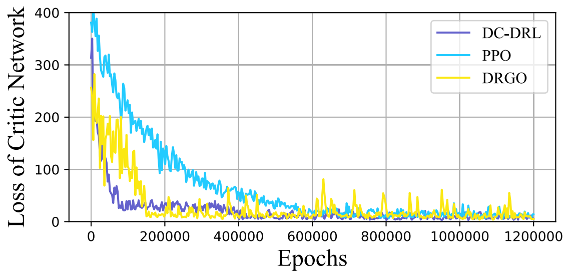
6.2 Convergence Analysis

In this work, we propose a DRL-based algorithm integrated with several improvements to achieve higher performance. To demonstrate this, we compare the convergence properties of the DC-DRL algorithm with the PPO [32] and our previous work [40]. PPO is a state-of-the-art DRL algorithm that applies a KL penalty coefficient [41] to adjust the stepsize of parameter update adaptively. In [40], we proposed a DRL-based algorithm called DRGO, which applies the adaptive genetic algorithm (AGA) to take advantage of the evaluation ability of the critic-network during the exploration process. Fig. 3 shows the average system cost and the loss of critic-network under scenarios where the number of MUs is 60. We use a measurement interval of 800s and the average system cost is defined as the average cost of the measurement interval. The loss of critic-network is the MSE of N samples randomly selected from the replay memory, which can be calculated with Equation (16). Because the parameters of the actor-network are updated based on the back propagation of the critic-network, we only plot the loss curve of the critic-network. Note that the epochs of the DC-DRL algorithm in Fig. 3 are the sum of the training epochs of all distributed DRL agents.

As highlighted in the left circle of Fig. 3a, the average system cost of the DC-DRL algorithm is the lowest at the beginning. This is because the surrogate gradients for the gradient descent of DRL are inaccurate during the early stages of training, while the DC-DRL algorithm can quickly obtain better parameter settings by combing deep neuroevolution with policy gradient. In addition, we observe that the DC-DRL converges after about 5×10^5 epochs, while DRGO and PPO converge after about 7.5×10^5 epochs. This illustrates that the improvements we introduce can



(a) Average system cost. vs. Epoch



(b) Loss of critic-network. vs. Epoch

Fig. 3. The convergence analysis of the DC-DRL algorithm.

accelerate the convergence. This is because the DC-DRL algorithm adopts multiple environments to increase the diversity of collected experiences. In particular, as illustrated in Fig. 3b, the loss value of the DC-DRL algorithm drops rapidly to about 30 at the beginning and then converges with a stable trend. This demonstrates that greater diversity contributes to a much faster and smoother learning curve. In addition, as highlighted in the box of Fig. 3a, the DC-DRL algorithm converges to the lowest cost by comparison. This difference in costs is due to the greater exploration ability of the DC-DRL algorithm, which contributes to reducing the unexplored action space and getting rid of the local optima. Consequently, it allows the DC-DRL algorithm to make decisions more comprehensively.

6.3 Performance Evaluation Under Various Environment Settings

6.3.1 Performance Under Different Numbers of MUs

In the following, we evaluate the performance of the DC-DRL algorithm under scenarios where the numbers of MUs range from 10 to 60. Since the DC-DRL is a long-term performance optimization algorithm, it is first compared to two baselines, i.e., (a) Random Strategy: each MU randomly decides whether to process the task locally or offload it to ES, (b) Greedy Strategy: we randomly generate 10,000 action sequences as the candidate set and select the action sequence with the lowest cost. Table 3 presents the comparative results. It is easy to find that the DC-DRL algorithm can achieve the lowest system cost compared with two baselines. It also shows that the performance gap between our proposed algorithm and the baselines grows as the number of MUs increases. This is because the long-term implications of current decisions are not taken into account in the random and greedy strategies. In this case, the current task may be completed at a lower cost, but resource shortage or even task failure may occur later, resulting in a higher system cost in the long run.

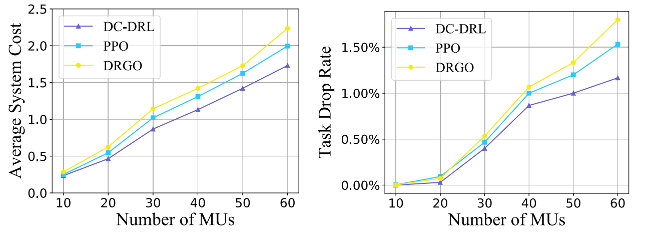
TABLE 3
Comparative Performance Under Different Number of MUs

Avg. Cost / Algorithm / User	DC-DRL	Greedy	Random
10 MUs	0.23	0.32	0.35
20 MUs	0.46	0.67	0.70
30 MUs	0.86	1.01	1.08
40 MUs	1.13	3.22	8.77
50 MUs	1.42	11.90	16.80
60 MUs	1.73	20.18	24.70

Also, we compare the DC-DRL algorithm with two advanced DRL algorithms that focus on optimizing long-term performance, that is, PPO and DRGO. The results are demonstrated in Fig. 4. From Fig. 4a and Table 3, it can be observed that by considering the long-term performance, all three algorithms (i.e., DC-DRL, PPO and DRGO) have achieved advantages over the above baselines that consider one-shot optimization (i.e., random and greedy). In particular, the DC-DRL algorithm outperforms the PPO and DRGO in terms of system cost, especially when the number of MUs exceeds 30. Fig. 4b reveals the same trend with respect to task drop rates. The DC-DRL algorithm can better reduce task latency to meet the deadline constraints. This is because the state and action spaces increase when the number of MUs increases, making PPO and DRGO more liable to fall into local optima. Moreover, the DC-DRL algorithm shows no sign of deterioration as the number of MUs increases, proving that it has certain scalability. This makes further applications feasible if the number of MUs increases by several orders of magnitude. We can divide MUs into several groups and reuse the DC-DRL algorithm to make offloading decisions.

6.3.2 Performance Under Different Sizes of Required Data

Next, to demonstrate the robustness of the DC-DRL algorithm to different applications, we consider the impact of the size of the required data, including program code and input data. The simulation results are shown in Fig. 5a, where the sizes of the required data range from 1,000 KB to 10,000 KB. The results show that the average system costs of



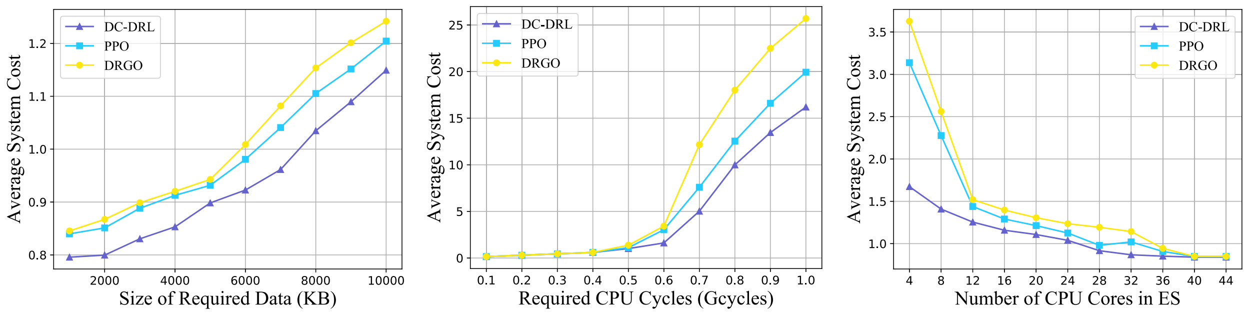
(a) Average system cost. vs. Number of MUs (b) Task drop rate. vs. Number of MUs

Fig. 4. Simulation results: performance under different numbers of MUs.

all applied algorithms are increasing. This is because increasing data size inevitably leads to an increase in data transmission time and queuing time. Even so, the DC-DRL algorithm still has the lowest average cost compared with the other baseline solutions. This is because the DC-DRL algorithm can better manage the wireless network resources, thus avoiding data congestion. In this case, some tasks with low computational overhead can be performed locally to alleviate network congestion, allowing subsequent tasks that are computationally expensive to be offloaded to ES. Therefore, in the long run, the DC-DRL algorithm is able to achieve a lower system cost.

6.3.3 Performance Under Different Required CPU Cycles

Next, we run simulations under scenarios with a greater range of task execution time. The required CPU cycles range from 0.1 Gcycles to 1.0 Gcycles. For a greater number of required CPU cycles, more energy is required to accomplish the task. On the other hand, more time is required given the same computation capacity. Fig. 5b demonstrates the variation of average system costs as the number of required CPU cycles changes. It can be observed that the performances of all three algorithms deteriorate as the number of required CPU cycles increases. In particular, the deterioration grows more rapidly when the required CPU cycles are larger than 0.6 Gcycles. This suggests that the increase in the initial average system cost is due to the increase in energy consumption, while the rapid deterioration in the case that CPU cycles are larger than 0.6 Gcycles is due to the lack of computational resources. In this case, the incoming tasks must wait for a long time in the task queue until the computational resources are available, which may result in task



(a) Average system cost. vs. Size of required data (KB) (b) Average system cost. vs. Required CPU cycles (Gcylces) (c) Average system cost. vs. Number of CPU Cores in ES

Fig. 5. Average system cost of the proposed DC-DRL algorithm and baselines under various settings.

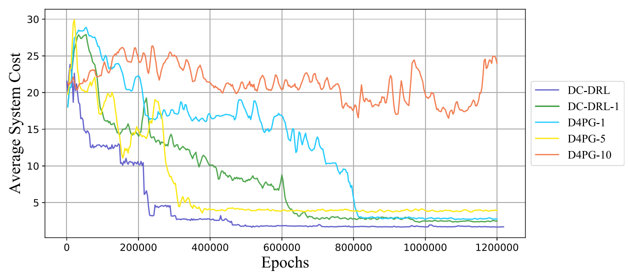


Fig. 6. Impact of adaptive n-step learning.

failure. Nevertheless, the DC-DRL algorithm still maintains a lower cost than that of the PPO and DRGO algorithms, proving its advantage in easing the strain on resources.

6.3.4 *Performance Under Different Number of CPU Cores in ES*

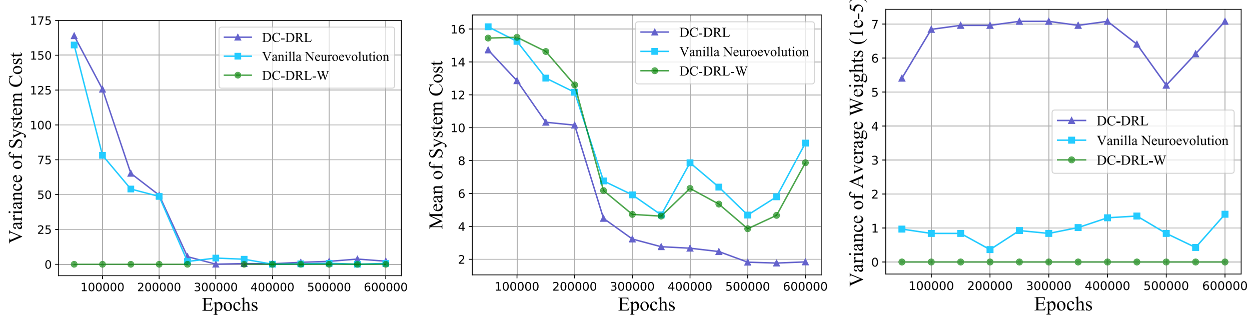
We also analyze the impact of the computation capacity of ES on the average system cost. As illustrated in Fig. 5c, the simulations are run under scenarios where the numbers of CPU cores in ES range from 4 to 44. As expected, the average system cost decreases as the number of CPU cores increases. This relationship is because high computation capacities can ease the conflict between reducing energy consumption and saving execution time. We observe that the DC-DRL algorithm achieves the lowest cost compared with the PPO and DRGO algorithms through the simulations. This suggests that our proposed algorithm can improve resource utilization to cope with the shortage of computational resources. Moreover, as the number of CPU cores of ES increases, the average system cost of the DC-DRL algorithm decreases at the beginning but remains stable when the number of CPU cores in ES is larger than 32. This is because the DC-DRL algorithm already reaches a near-optimal (even optimal) strategy, redundant computational resources do not lead to significant performance improvements. Based on this insight, we can deploy computational resources on the server in a more resource-efficient manner.

6.4 Impact of Adaptive N-Step Learning

In this section, we present the performance improvement of integrating adaptive n-step learning in the design of the DC-DRL algorithm. For comparison, we adopt the D4PG algorithm [25] as the baseline, which is a state-of-the-art distributed DRL algorithm. In particular, D4PG uses a preset n to calculate the n-step return for parameter updates. Without loss of generality, we compare the DC-DRL with the D4PG algorithm when n equals 1, 5, and 10, which are denoted as D4PG-1, D4PG-5, D4PG-10, respectively. In addition, we compare it with the DC-DRL without adaptive n-step learning by setting $n_{\max} = 1$, which is denoted as DC-DRL-1. The simulations are run under scenarios where $U = 60$. Fig. 6 shows the variation in average costs during the entire training process. It is obvious that the cost of the D4PG-10 fluctuates acutely and shows no sign of convergence. This is because a large n inevitably leads to high variance in distributed training settings. Besides, with $n = 1$, the D4PG-1 and DC-DRL-1 converge at a much slower rate compared with the DC-DRL. The reason is that the long-term reward estimates of D4PG-1 and DC-DRL-1 in the early training phase are highly biased, resulting in a high deviation from the true direction of convergence. On the other hand, the DC-DRL can leverage the information of multiple steps to obtain a better estimate and improve efficiency. Moreover, compared with the D4PG-5 algorithm, the curve of DC-DRL is significantly more stable and has less fluctuation despite their convergence trends are similar, which attributes to the policy correction of adaptive n-step learning. We draw the conclusion that the DC-DRL algorithm can achieve a balance between reducing variance and improving training efficiency.

6.5 Impact of Combining Deep Neuroevolution With Policy Gradient

Next, we investigate the performance improvement of combining deep neuroevolution with policy gradient. On the one hand, as shown in Fig. 6, we can note that although the D4PG-1 is not troubled by the high variance issue, its final performance is still inferior to the DC-DRL algorithm. It implies that the introduction of deep neuroevolution can enhance exploration ability and avoid premature convergence. On the other hand, we compare our algorithm with vanilla (standard) deep neuroevolution [11] and DC-DRL without deep neuroevolution (denoted as DC-DRL-W) to investigate the performance improvement of introducing policy gradient. DC-DRL-W is implemented by setting $P = 1$. As we mentioned, deep neuroevolution maintains a population with P distributed DRL agents. In this case, it is important to achieve fairness in terms of individual efforts in training DRL agents. Fig. 7a shows the variance of system



(a) Variance of System Cost. vs. Epochs (b) Mean of System Cost. vs. Epochs (c) Variance of Average Weights. vs. Epochs

Fig. 7. Impact of combining deep neuroevolution with policy gradient.

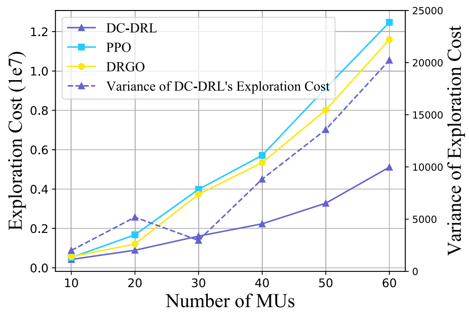


Fig. 8. Exploration cost under different number of MUs.

costs across the population. The variances of DC-DRL-W are 0 because $P = 1$. We find that the DC-DRL and vanilla deep neuroevolution share similar trends, and the variance soon decreases to around 0 after training. This suggests that the performances of different DRL agents vary within a small range, ensuring the fairness of training.

In addition, Fig. 7b shows the mean of system costs across the population. Because the DC-DRL can leverage the DRL population, its costs decrease faster than the DC-DRL-W. Noteworthy, although the variances of DC-DRL and vanilla deep neuroevolution are similar, the rate of decline in the costs varies greatly. To explain this, we calculate the average weight of the first layer in the actor-network and the corresponding variances of the population. Fig. 7c presents the result. Compared to vanilla deep neuroevolution, the DC-DRL algorithm has higher variance across the entire training process. Intuitively, higher variance means a larger explored area. As a result, the DC-DRL algorithm is easier to escape from local optima and find an excellent solution.

6.6 Performance With Respect to Exploration Cost

To deploy DRL applications in MEC, a major obstacle lies in the huge exploration costs during the training process. Fig. 8 exhibits the required exploration costs for three DRL algorithms to converge. The simulations are run under scenarios where the numbers of MUs range from 10 to 60. Note that the exploration costs for the DC-DRL algorithm are the sum of the costs for all distributed agents. We see that the DC-DRL algorithm achieves the lowest exploration costs even in the case that the DRGO attempts to reduce the exploration costs by leveraging AGA and the evaluation ability of the critic-network. The reasons are stated in the following. On one hand, the DC-DRL algorithm uses adaptive n-step learning to improve the sample efficiency and accelerate the training process. On the other hand, the introduction of deep neuroevolution enables the DC-DRL to quickly obtain better parameter settings in the early stages of training. Besides, we also plot the variance of DC-DRL's exploration costs to study the fairness across the population. It can be observed that the variance is smaller by three orders of magnitude compared with the total exploration costs. Therefore, we can conclude that the costs are evenly spread across multiple environments.

6.7 Experiments in Real-World Scenarios

Furthermore, to validate the practicality and applicability of the DC-DRL algorithm, we build a real multi-user computation

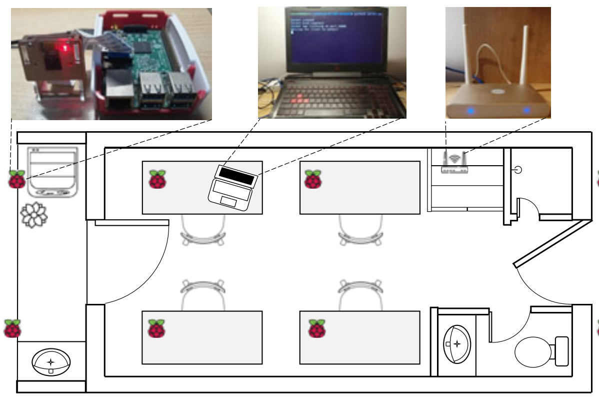


Fig. 9. The implemented multi-user computation offloading testbed.

offloading testbed as in [42]. The testbed structure is shown in Fig. 9. An OMEN Gaming PC of HP is used as ES, equipped with an Intel Core i7-7700HQ CPU (4 physical cores, each running at 2.8-3.8 GHz) and 16 GB RAM. A set of Raspberry Pi3 B are used as MUs, equipped with an ARMv7 Processor (4 physical cores, each running at 1.2 GHz) and 1 GB RAM. For connectivity, a WiFi router acts as the base station to connect MUs and ES. The Raspberry Pi3 B is integrated 802.11n wireless LAN, allowing it to offload tasks to ES through WiFi. The transmit power of Raspberry Pi3 B is 31 dBm. Similar to [42], we employ a face recognition application in the experiments, which is widely used in many scenarios, (i.e., smart home security). Due to the CPU and memory limits, Raspberry Pi can offload the incoming face recognition tasks to PC. We deploy our computation offloading prototype on both MUs and ES to support edge computing.

The experiments are conducted under different number of Raspberry Pis. To calculate the system cost, we record the CPU occupation time, data transmission time, and application response time. In addition, the energy consumption is estimated according to the approach in [28]. The experimental results are illustrated in Fig. 10, where the edge-only means all tasks are performed on ES and the local-only means all tasks are performed locally. When the number of Raspberry Pis is less than 5, the edge-only algorithm outperforms the random algorithm. However, with the increase of the Raspberry Pis' number, the performance of the edge-only algorithm deteriorates at a much faster rate than the random algorithm. This is because in the edge-only algorithm, all Raspberry Pis are competing for the limited resources of ES, and no one can enjoy the expected performance. It reveals that the experimental testbed is competitive and poses high requirements for offloading decisions. In addition, we can observe that when the number of

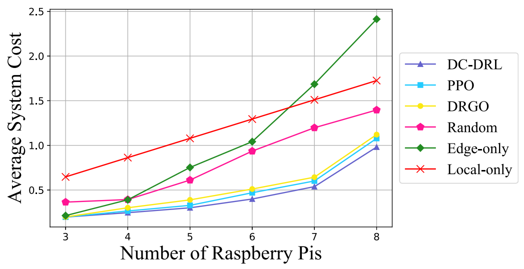


Fig. 10. Average system cost versus the number of Raspberry Pis.

Raspberry Pis is less than 7, the average cost of DC-DRL increases with a stable trend, while when the number is larger than 7, the system cost begins to rise at a faster rate. The reasons are two folds. First, the initial increase in system cost is mainly attributed to the increase in energy consumption. And when the number of Raspberry Pis is larger than 7, the rapid rise in cost is due to the lack of resources. Second, due to the exponential growth of search space, the increase in the Raspberry Pis' number makes learning more challenging. Even so, through trial-and-error, DC-DRL is more likely to find a high quality solution. Overall, the average performance gains over PPO, DRGO, random, edge-only, and local-only are approximately 9.4, 15.8, 45.6, 59.0, and 62.5 percent, respectively.

7 Conclusion

In this paper, we have studied the computation offloading problem in multi-user MEC. With the objectives of minimizing the overall system cost and adapting to the dynamic environment, we formulate the optimization problem as an MDP and leverage the recent advances in DRL to solve the problem. Due to the fact that it is impractical to directly apply existing DRL algorithms to MEC, we propose a distributed and collective DRL algorithm called DC-DRL and empower it with a better practical perspective. To provide a greater number of experiences and increase the experience diversity, the DC-DRL algorithm assimilates experiences and knowledge from multiple environments and improves the performance of distributed DRL agents collectively. Furthermore, we propose adaptive n-step learning and combine deep neuroevolution with policy gradient to maximize the benefits of distributed collective training. Evaluation results have shown that: 1) the DC-DRL algorithm can obtain knowledge beyond a single agent for higher performance and evenly spread the costs across multiple environments, 2) the adaptive n-step learning can improve the training efficiency without affecting the convergence stability, 3) the combination of deep neuroevolution and policy gradient can greatly improve the utility of the distributed training system, thus reducing exploration costs and improving convergence effects. In the future, we shall consider using DRL to schedule sub-tasks with directed acyclic graph (DAG) constraints.

Acknowledgments

The work described in this article was supported in part by the National Key Research and Development Plan (2018YFB1 003803), in part by National Natural Science Foundation of China (61802450, 61722214), in part by Natural Science Foundation of Guangdong(2018A030313005), in part by Program for Guangdong Introducing Innovative and Entrepreneurial Teams (2017ZT07X355), and in part by Pearl River Talent Recruitment Program (No. 2019QN01X130).

References

[1] Y. Li, A.-C. Orgerie, I. Rodero, B. L. Amersho, M. Parashar, and J.-M. Menaud, "End-to-end energy models for edge cloud-based IoT platforms: Application to data stream analysis in IoT," *Future Gener. Comput. Syst.*, vol. 87, pp. 667–678, 2018.

[2] A. R. Zamani, I. Petri, J. Diaz-Montes, O. Rana, and M. Parashar, "Edge-supported approximate analysis for long running computations," in *Proc. IEEE 5th Int. Conf. Future Internet Things Cloud*, 2017, pp. 321–328.

[3] C. Xu *et al.*, "Making big data open in edges: A resource-efficient blockchain-based approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 870–882, Apr. 2019.

[4] S. Mu, Z. Zhong, D. Zhao, and M. Ni, "Joint job partitioning and collaborative computation offloading for Internet of Things," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 1046–1059, Feb. 2019.

[5] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] O. Vinyals *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[7] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," 2018, *arXiv: 1812.00332*.

[8] M. Tan *et al.*, "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 2820–2828.

[9] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, "Bridging the gap between value and policy based reinforcement learning," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 2775–2785.

[10] L. Espeholt *et al.*, "IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1802.01561

[11] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017, *arXiv: 1703.03864*.

[12] W. Chen, Y. Yaguchi, K. Naruse, Y. Watanobe, and K. Nakamura, "QoS-aware robotic streaming workflow allocation in cloud robotics systems," *IEEE Trans. Services Comput.*, to be published, doi: 10.1109/TSC.2018.2803826.

[13] Z. Hong, W. Chen, H. Huang, S. Guo, and Z. Zheng, "Multi-hop cooperative computation offloading for industrial IoT–edge–cloud computing environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, pp. 2759–2774, Dec. 2019.

[14] W. Li, X. You, Y. Jiang, J. Yang, and L. Hu, "Opportunistic computing offloading in edge clouds," *J. Parallel Distrib. Comput.*, vol. 123, pp. 69–76, 2019.

[15] A. Alelaiwi, "An efficient method of computation offloading in an edge cloud platform," *J. Parallel Distrib. Comput.*, vol. 127, pp. 58–64, 2019.

[16] J. Zhao, Q. Li, Y. Gong, and K. Zhang, "Computation offloading and resource allocation for cloud assisted mobile edge computing in vehicular networks," *IEEE Trans. Veh. Technol.*, vol. 68, no. 8, pp. 7944–7956, Aug. 2019.

[17] Z. Zhang, Z. Hong, W. Chen, Z. Zheng, and X. Chen, "Joint computation offloading and coin loaning for blockchain-empowered mobile-edge computing," *IEEE Internet Things J.*, vol. 6, no. 6, pp. 9934–9950, Dec. 2019.

[18] Y. Wang, K. Wang, H. Huang, T. Miyazaki, and S. Guo, "Traffic and computation co-offloading with reinforcement learning in fog computing for industrial applications," *IEEE Trans. Ind. Informat.*, vol. 15, no. 2, pp. 976–986, Feb. 2019.

[19] M. G. R. Alam, M. M. Hassan, M. Z. Uddin, A. Almogren, and G. Fortino, "Autonomic computation offloading in mobile edge for IoT applications," *Future Gener. Comput. Syst.*, vol. 90, pp. 149–157, 2019.

[20] Q. Qi *et al.*, "Knowledge-driven service offloading decision for vehicular edge computing: A deep reinforcement learning approach," *IEEE Trans. Veh. Technol.*, vol. 68, no. 5, pp. 4192–4203, May 2019.

[21] J.-Y. Baek, G. Kaddoum, S. Garg, K. Kaur, and V. Gravel, "Managing fog networks using reinforcement learning based load balancing algorithm," 2019, *arXiv: 1901.10023*.

[22] W. Jiang, G. Feng, S. Qin, T. S. P. Yum, and G. Cao, "Multi-agent reinforcement learning for efficient content caching in mobile D2D networks," *IEEE Trans. Wireless Commun.*, vol. 18, no. 3, pp. 1610–1622, Mar 2019.

[23] H. Y. Ong, K. Chavez, and A. Hong, "Distributed deep Q-learning," 2015, *arXiv:1508.04186*.

[24] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.

[25] G. Barth-Maron *et al.*, "Distributed distributional deterministic policy gradients," 2018, *arXiv: 1804.08617*.

[26] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019.

[27] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[28] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2012, pp. 2716–2720.

[29] G. Gao, M. Xiao, J. Wu, K. Han, L. Huang, and Z. Zhao, "Opportunistic mobile data offloading with deadline constraints," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3584–3599, Dec. 2017.

[30] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2000, pp. 1008–1014.

[31] T. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," in *Proc. Int. Conf. Learn. Representations*, 2016. [Online]. Available: http://arxiv.org/abs/1509.02971

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[33] H. B. McMahan *et al.*, "Communication-efficient learning of deep networks from decentralized data," 2016, *arXiv:1602.05629*.

[34] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," 2019, *arXiv: 1904.12901*.

[35] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," 2016, *arXiv:1604.00981*.

[36] S. Wang *et al.*, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 63–71.

[37] J. C. Duchi, M. I. Jordan, M. J. Wainwright, and A. Wibisono, "Optimal rates for zero-order convex optimization: The power of two function evaluations," *IEEE Trans. Inf. Theory*, vol. 61, no. 5, pp. 2788–2806, May 2015.

[38] H. Beyer, "Evolution strategies," *Scholarpedia*, vol. 2, no. 8, 2007, Art. no. 1965.

[39] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, Fourth-quarter 2017.

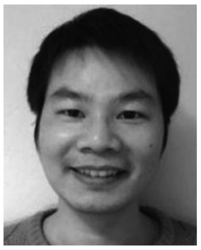
[40] X. Qiu, L. Liu, W. Chen, Z. Hong, and Z. Zheng, "Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing," *IEEE Trans. Veh. Technol.*, vol. 68, no. 8, pp. 8050–8062, Aug. 2019.

[41] S. Kakade and J. Langford, "Approximately optimal approximate reinforcement learning," in *Proc. 19th Int. Conf. Mach. Learn.*, 2002, vol. 2, pp. 267–274.

[42] M. Hu, L. Zhuang, D. Wu, Y. Zhou, X. Chen, and L. Xiao, "Learning driven computation offloading for asymmetrically informed edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1802–1815, Aug. 2019.

Xiaoyu Qiu received the BS degree from Sun Yat-Sen University, Guangzhou, China, in 2020. He is currently working toward the MS degree with the School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China. He is proactively working on edge computing, cloud computing, cloud robotics, and computation offloading, with emphasis on artificial intelligence in edge/cloud computing.

Weikun Zhang is currently working toward the BEng degree in software engineering with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China. His current research interests include the blockchain enabled market, the resource allocation for edge computing, network economics, and IoT.



Wuhui Chen (Member, IEEE) received the bachelor's degree from Northeast University, Shenyang, China, in 2008 and the master's and PhD degrees from the University of Aizu, Aizu-Wakamatsu, Japan, in 2011 and 2014, respectively. From 2014 to 2016, he was a research fellow with the Japan Society for the Promotion of Science, Japan. From 2016 to 2017, he was a researcher with the University of Aizu, Japan. He is currently an associate professor with Sun Yat-Sen University, Guangzhou, China. His research interests include edge/cloud computing, cloud robotics, and blockchain.

Zibin Zheng (Senior Member, IEEE) is currently a professor with the School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China. His research interests include service computing and cloud computing. He received the Outstanding PhD Thesis Award of the Chinese University of Hong Kong, Hong Kong, in 2012, the Association for Computing Machinery's Special Interest Group on Software Engineering Distinguished Paper Award at the International Conference on Science and Engineering, 2010, the Best Student Paper Award at the International Conference on Web Services, 2010, and the IBM PhD Fellowship Award at 2010. He served as program committee (PC) member of the IEEE International Conference on Cloud Computing, International Conference on Web Services, International Conference on Service Computing (SCC), International Conference on Service-Oriented Computing, International Symposium on Service-Oriented System Engineering (SOSE), etc.