Deep-Reinforcement-Learning-Based Offloading Scheduling for Vehicular Edge Computing

Wenhan Zhan, Chunbo Luo, *Member, IEEE*, Jin Wang, Chao Wang, *Member, IEEE*, Geyong Min, Hancong Duan, and Qingxin Zhu

***Abstract*—Vehicular edge computing (VEC) is a new computing paradigm that has great potential to enhance the capability of vehicle terminals (VTs) to support resource-hungry in-vehicle applications with low latency and high energy efficiency. In this article, we investigate an important computation offloading scheduling problem in a typical VEC scenario, where a VT traveling along an expressway intends to schedule its tasks waiting in the queue to minimize the long-term cost in terms of a tradeoff between task latency and energy consumption. Due to diverse task characteristics, dynamic wireless environment, and frequent handover events caused by vehicle movements, an optimal solution should take into account both *where* to schedule (i.e., local computation or offloading) and *when* to schedule (i.e., the order and time for execution) each task. To solve such a complicated stochastic optimization problem, we model it by a carefully designed Markov decision process (MDP) and resort to deep reinforcement learning (DRL) to deal with the enormous state space. Our DRL implementation is designed based on the state-of-the-art proximal policy optimization (PPO) algorithm. A parameter-shared network architecture combined with a convolutional neural network (CNN) is utilized to approximate both policy and value function, which can effectively extract representative features. A series of adjustments to the state and reward representations are taken to further improve the training efficiency. Extensive simulation experiments and comprehensive comparisons with six known baseline algorithms and their heuristic combinations clearly demonstrate the advantages of the proposed DRL-based offloading scheduling method.**

Manuscript received August 29, 2019; revised January 6, 2020 and February 19, 2020; accepted March 2, 2020. Date of publication March 6, 2020; date of current version June 12, 2020. This work was supported in part by the National Natural Science Foundation of China under Grant 61871096 and Grant 61972075, in part by the National Key Research and Development Program of China under Grant 2018YFB2101300, in part by the EU H2020 Research and Innovation Programme under the Marie Sklodowska-Curie under Grant 752979, and in part by the China Scholarship Council. *(Corresponding author: Chunbo Luo.)*

Wenhan Zhan is with the School of Computer Science and Engineering and the School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: zhanwenhan@uestc.edu.cn).

Chunbo Luo is with the School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China, and also with the College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter EX4 4QF, U.K. (e-mail: c.luo@uestc.edu.cn).

Jin Wang, Chao Wang, and Geyong Min are with the College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter EX4 4QF, U.K. (e-mail: jw855@exeter.ac.uk; chaowang@tongji.edu.cn; g.min@exeter.ac.uk).

Hancong Duan is with School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: duanhancong@uestc.edu.cn).

Qingxin Zhu is with School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: qxzhu@uestc.edu.cn).

Digital Object Identifier 10.1109/JIOT.2020.2978830

***Index Terms*—Computation offloading, deep reinforcement learning (DRL), mobile-edge computing (MEC), task scheduling, vehicular edge computing (VEC).**

I. Introduction

With the rapid development of the Internet-of-Things (IoT) technologies, smart vehicles have become increasingly prevalent, which has further boosted the proliferation of novel in-vehicle applications, such as autonomous driving, augmented reality, and virus scanning [1], [2]. Such advanced vehicular applications are in general computation intensive, bandwidth consuming, and/or latency sensitive, and it is difficult for onboard software and hardware systems to meet their demanding resource requirements. Vehicular edge computing (VEC) [1]–[3], which is the application of mobile-edge computing (MEC) [4], [5] in vehicular scenarios, has recently received significant attention as a promising solution to improve the situation. By offloading these resource-demanding tasks to the MEC servers attached in the roadside units (RSUs), the execution latency and energy consumption of in-vehicle applications can be significantly reduced. At the same time, the bandwidth to the core network can be saved, diminishing the risk of network congestion.

Task offloading is a key feature of MEC/VEC technologies. Due to the extra energy and time consumption induced by data transmission and remote task execution, offloading computation tasks to edge servers may not always bring benefits. A key technical challenge is to balance the overall costs of computation and communication when making offloading decisions. A number of efficient offloading algorithms have been reported in the recent literature (e.g., [3], [6], and [7]). To simplify the decision-making problem, most of the existing solutions are based on the assumption of a static or quasistatic environment, e.g., the state of the wireless channel is fixed during the complete offloading procedure. When user mobility and complex wireless signal propagation are taken into consideration, the assumption of a static environment no longer holds. The Markov decision process (MDP) is an effective mathematical tool to model the impact of user actions in a dynamic environment and allows seeking the optimal offloading decision for achieving a particular long-term goal [8]–[10]. To this end, a state transition probability matrix that describes the system dynamics (i.e., the probabilities of user actions leading to state transitions) should be constructed, based on which the optimal offloading policy can be derived by value iteration or policy iteration. However, in most real-world scenarios, the system dynamics is hard to measure or model. The transition probability matrix is normally intractable to obtain, especially when the state and action spaces are large.

Deep reinforcement learning (DRL) [11], [12] is envisioned as a promising solution to complex sequential decision-making problems and has attracted increasing research interests in a wide range of scientific disciplines. DRL is particularly suitable for solving the MEC/VEC offloading problems in dynamic environments due to a number of reasons. First, DRL can target the optimization of long-term offloading performance. This would outperform the "one-shot" and greedy application of the approaches proposed in static environments (e.g., [3], [6], and [7]), which may lead to strictly suboptimal results. Second, through DRL, the optimal offloading policy can be learned by interacting directly with the environment without any prior knowledge of the system dynamics (e.g., wireless channel or task arrival characteristics). This avoids the demand of the state transition matrix that is necessary when the conventional solutions are utilized to solve MDP (e.g., [8]–[10]). Third, DRL can take full advantage of the powerful representation capability of the deep neural network (DNN). The optimal offloading policy can be adequately approximated even in complicated problems with vast state and/or action spaces.

Recently, novel DRL-based offloading strategy designs have begun to emerge. For instance, Wang *et al.* [13] proposed an offloading method to schedule mutually dependent tasks. A deep Q-learning-based algorithm is developed in [14] to solve the offloading decision-making problem among multiple users. Both [13] and [14] consider static environments and resort to DRL to deal with the enormous state space. Offloading problems in dynamic scenarios are investigated in [15]–[17]. For example, Min *et al.* [15] utilized deep Q-learning to find the optimal offloading policy for an IoT device with energy harvesting (EH) capability and to select the optimal edge device and offloading data rate to maximize long-term system performance. They considered a fine-granularity application with a fixed computation-to-volume ratio (CVR). A computation task in this application model can be partitioned into multiple parts, and the number of CPU cycles required for computing each input bit is fixed. Chen *et al.* [16] designed a DRL-based offloading scheme for a mobile device with EH capability to select the best base station and the amount of energy for offloading. A queue of tasks with identical size and CVR is considered in this article. Our earlier work [17] proposes a DRL-based method to solve the offloading decision-making problem of a vehicle in a VEC environment, aiming to minimize its long-term offloading cost. The adopted application model is similar to [16]. The proliferation of new in-vehicle applications prompts the offloading methods in the VEC scenario to support more complex application models with different task sizes and CVRs. The diversity of offloading tasks coupled with the high environment dynamics makes the offloading problem more complicated, and all the methods introduced above are not applicable.

In this article, we investigate the computation offloading scheduling problem in a typical VEC scenario, where a vehicle terminal (VT) traveling along an expressway decides how to schedule the tasks waiting in its task queue, as shown in Fig. 1. The tasks are independently generated by different applications so that they have diverse characteristics (regarding data size and CVR). MEC servers equipped in RSUs can be utilized to conduct computation for the VT. The wireless vehicular communication environment is complex. Fading statistics may be unknown, and instantaneous channel knowledge is only causally available. Due to the VT's mobility, handover from one serving RSU to another occurs periodically. These issues lead to dynamically changing data transmission time/energy consumption and even transmission failures. Therefore, a good offloading scheduling strategy should decide not only *where* to schedule each task (i.e., executing the task locally in the VT or remotely on a MEC server) but also *when* to schedule (i.e., the scheduling order and time of each task). This problem is very involved in conventional solutions because of the sophisticated environment dynamics and vast state space. To tackle such challenges, we design a novel DRL-based offloading scheduling method (DRLOSM) that can minimize the long-term cost defined as a tradeoff between task execution latency and energy consumption. The main contributions of this article are summarized as follows.

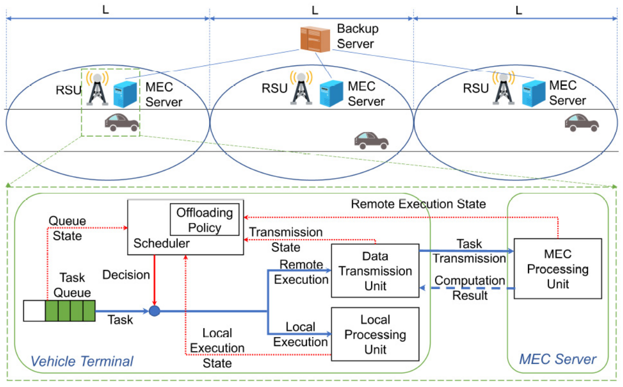


Fig. 1. Architecture of VEC offloading.

1) We model the offloading scheduling process by a carefully designed MDP, in which the impacts of task characteristics, wireless transmission and queue dynamics, and VT mobility are all taken into account. Considering that the MDP is by nature model-free and has an enormous state and action space, we propose applying DRL to find the optimal policy.
2) A series of approaches are applied to improve training efficiency and convergence performance. First, we design the training method based on the proximal policy optimization (PPO) algorithm, which is the state-of-the-art policy gradient method with excellent stability and reliability [18]. In addition, to better extract representative features of the task queue, a convolutional neural network (CNN) is embedded in the DNN architecture, which is used to approximate the offloading scheduling policy and value function. Finally, by deliberately adjusting the state and reward representations, a huge number of inefficient exploration attempts in the training process can be avoided to further improve the training efficiency.

3) Extensive simulation experiments are conducted to compare the proposed DRLOSM with six known baseline algorithms and their heuristic combinations. The results show that our approach can always achieve the lowest long-term cost. The potentials of applying DRL to solve complex decision-making problems in VEC are clearly exhibited.

The remainder of this article is organized as follows. The system model is presented in Section II. In Section III, we briefly introduce the background of DRL. Section IV formulates the computation offloading scheduling problem as an MDP. The implementation of DRL and the training method is elaborated in Section V. The simulation results and discussions are provided in Section VI. Finally, Section VII concludes this article.

II. System Model

In this section, we first introduce the system architecture. Then, the detailed mathematical models of task queue, communication and computation operations, and the overall designing objective are elaborated.

Notation: Throughout this article, we use $\mathbf{N}^+$ to denote the set of all positive integers. $\mathcal{CN}(0,\sigma^2)$ denotes a complex Gaussian distribution with zero mean and variance σ^2. $\lceil\cdot\rceil$ denotes the ceiling operator. $\mathbf{1}_{\{\Phi\}}$ is the indicator function that equals 1 if the condition Φ is satisfied and otherwise 0.

A. System Architecture

In this article, we consider a typical application scenario of VEC [2], as shown in Fig. 1. VTs are driven along an expressway, served by RSUs deployed along the roadside. The distance between adjacent RSUs is L meters, and the coverage regions of different RSUs do not overlap. Hence, the road is divided into segments based on the coverage range of RSU. A VT can only be served by one RSU through vehicle-to-infrastructure (V2I) communication. When it is driven across the boundary of two road segments, a handover occurs. Each RSU is equipped with a MEC server, which acts as a proximity cloud for VTs, i.e., a certain amount of computing resources can be reserved for each VT to conduct task computation. A backup server located at the network backend is connected to all the MEC servers through wired connections and can enhance the capability of each MEC server when its computing resource is not adequate. MEC servers can communicate with each other through the core network. Nevertheless, to avoid degrading the condition of the core network, raw in-vehicle application data (e.g., input data for task offloading), which usually have large size, are not allowed to be transmitted between RSUs [2].

The focus of this article is to, from the perspective of VTs, identify an offloading scheduling strategy that can efficiently complete the execution of in-vehicle application computation tasks with a minimum cost. Our analysis hereinafter concentrates on one single representative VT.[1] The interaction between the VT and its serving MEC server regarding computation offloading is shown at the bottom of Fig. 1. We consider an abstract computing architecture in the VT, which consists of a task queue, a task scheduler, a local processing unit (LPU), and a data transmission unit (DTU).[2] Independent computation tasks, possibly generated by multiple types of applications, randomly arrive in the task queue (sorted by their generation time). The task *scheduler* synthesizes all available system information [including queue state, local execution (LE) state, transmission state, and remote execution (RE) state] and schedules the tasks based on an *offloading scheduling policy*. A task that is assigned for LE is processed on the LPU, according to the scheduled execution time. For RE, a task is first transmitted to the serving RSU via the DTU. Afterward, the MEC server at the RSU executes the task using the reserved computation resource and then sends the computation result back to the VT.

The whole system is considered to operate in a *slotted* fashion: the smallest time interval for clock counting, software/hardware operation, and message coding/transmission, is deemed as a unit *time slot*. At the beginning of each time slot, the scheduler monitors the system state and, if necessary, performs the offloading scheduling. Then, computation and/or communication operate during the entire time slot. The time instant that the considered VT starts conducting the scheduling process of the tasks in its queue is taken as the initial time slot 0. Let the speed of the VT be V (in meter/slot). At any time slot t ($t \in \mathbf{N}^+$), its location (coordinate) can be found by $x[t] = Vt + x[0]$, where $x[0]$ is the starting location.

B. Task Queue Model

We model the system's workload as a Poisson process with rate λ, indicating the expected number of computation tasks arriving in the VT's task queue in each time slot. The ith ($i \in \mathbf{N}^+$) arriving task J_i is described as a 3-tuple

$$J_i \triangleq \left(t_i^{\mathrm{g}}, d_i, k_i\right). \tag{1}$$

In (1), t_i^{g} is the time that J_i is generated, d_i (in bit) is the size of the task input data, and k_i (in CPU cycle/bit) is its CVR, which can be obtained by applying program profilers [19]. All tasks waiting in the queue are considered to be generated by computation-intensive in-vehicle applications (e.g., object recognition or virus scanning). In general, the computation results of such tasks (e.g., object tags or virus reports) have sufficiently smaller data sizes compared with those of the input (e.g., pictures or files) [3]. Therefore, the size and transmission time of the output data of all tasks are considered to be negligible throughout this article.

We consider tasks without stringent latency demands or execution priority. They are sorted in the queue by their generated time. In other words, when a new task arrives, it is appended to the rear end of the task queue. If any task in the queue is sent to the LPU or DTU by the scheduler, tasks after it are shifted forward to fill the empty position. We use Q to denote the maximum number of tasks the queue can hold, and use

[1]In this article, a fixed amount of the MEC computation resource and V2I transmission bandwidth is assumed to be reserved for each VT. Dynamically optimizing the allocation of limited computing and communication resources among multiple VTs is a more challenging issue, which is one of our future research topics.

[2]The focus of this article is to identify a proper offloading scheduling strategy to balance local and remote execution in a complicated vehicular communication and computation environment. A computing architecture with a single central unit is adopted.

$q[t]$ ($q[t] \leq Q$) to denote the actual number of tasks in the queue at time slot t. If $q[t] = Q$, new incoming tasks have to be discarded and an overflow event occurs. Now, the state of the task queue at any time slot t can be represented by a $Q \times 3$ matrix $\mathbf{Q}[t]$, in which the jth ($j \in \{1, 2, \ldots, q[t]\}$) row is formed by the three defining elements of the jth waiting task (denoted by $\mathbf{Q}[t]\langle j\rangle$), i.e., task generation time, input data size, and CVR, respectively. Each of the remaining $Q - q[t]$ rows in $\mathbf{Q}[t]$ is a 1×3 all-zero vector, indicating the absence of a waiting task at that queue position.

Note that we can use two notations to refer to a task. The first reflects the natural task generation process. For example, J_i in (1) represents the ith task generated by an in-vehicle application, and index i can be any positive integer. The second type of notation reflects the real-time status of the task queue. For instance, $\mathbf{Q}[t]\langle j\rangle$ is the jth (sorted) task waiting in the queue at a particular time slot t, and integer index j is upper bounded by Q. The former notation uniquely distinguishes different tasks. But the latter can better help describe the dynamic nature of the considered system. To facilitate presentation, with a little abuse of notation, we follow the similar way as (1) to describe task $\mathbf{Q}[t]\langle j\rangle$: when time t is known, we have

$$\mathbf{Q}[t]\langle j\rangle \triangleq \left(t_j^{\mathrm{g}}, d_j, k_j\right). \tag{2}$$

C. Communication Model

The wireless V2I communication (for task offloading) is conducted in a block-fading environment. The fading coefficient remains unchanged within each channel coherence time interval, which can be one or multiple time slots, but varies randomly (not necessarily independently) afterward. We assume that the channel fading coefficient between the VT and its serving RSU is *statistically* determined by the signal propagation environment between them. For instance, under Rayleigh fading, the complex fading coefficient h can be modeled by $h = \tilde{h}\hat{h}$, where $\hat{h} \sim \mathcal{CN}(0, 1)$ represents the small-scale fading and the large-scale fading coefficient $\tilde{h}$ reflects the impact of both path loss and shadowing phenomenon. In general, $\tilde{h}$ is a function of $\mathcal{E}$, a set of environmental factors, such as the position of the VT x, the distance to the serving RSU d, and possibly other factors. Since $\mathcal{E}$ is relatively simple to estimate, knowing $\mathcal{E}$ and thus $\tilde{h}$ statistically describes h as a random variable generated from $\mathcal{CN}(0, |\tilde{h}|^2)$.

Taking the capability of the employed channel code into consideration, the reliable transmission data rate r from the VT to its serving RSU is a determined function of the channel fading coefficient. This means the transmission rate is also statistically determined by the environmental factors, i.e., it obeys a conditional probability density function (PDF) $f(r|\mathcal{E})$. For a demonstration purpose, in this article, we limit the set $\mathcal{E} = \{x, d\}$: the environment is completely described by the location of the VT and the distance between the VT and the serving RSU. For fixed x and d, the PDF $f(r|\mathcal{E})$ is a fixed function but is unknown to either the VT or RSUs.

At any time slot t, the VT can obtain a certain level of knowledge regarding the instantaneous fading coefficient $h[t]$, based on which it can infer the data rate $r[t]$ (in bits/slot). If the VT intends to transmit data to the RSU at time slot t, then $r[t]$ bits data can be successfully delivered. Therefore, we denote $t^{\mathrm{tx}}(v, t)$ as the time consumed for transmitting data of size v (in bit) to the serving RSU, starting from time slot t. It satisfies

$$\sum_{s=t}^{t+t^{\mathrm{tx}}(v,t)-1} r[s] < v \leq \sum_{s=t}^{t+t^{\mathrm{tx}}(v,t)} r[s]. \tag{3}$$

In practice, channel knowledge can only be causally available. The movement of the VT also contributes to a dynamic channel environment where channel gains change rapidly. It is difficult to predict the exact achievable transmission rate at future time slots. Hence, the value of $t^{\mathrm{tx}}(v, t)$ can only be known when the data transmission actually completes. As a result, a good offloading scheduling strategy should make the offloading decisions sequentially, instead of conducting one-shot solution.

Another issue caused by the movement of the VT and the dynamic wireless channel environment is the transmission failure due to handover. As we mentioned earlier, to avoid degrading the condition of the core network, raw in-vehicle application data are not allowed to be transmitted between RSUs. When the VT leaves the coverage area of an RSU, if the input data transmission of a task has not yet finished, the previously transmitted data cannot be passed to the new serving RSU and have to be discarded. The task has to be retransmitted by the VT with the cost of wasted energy and large delay. In contrast, since the size of computation output data is negligible, the computation result can be delivered from the previous serving RSU to the new one if the task input data transmission has been completed before handover.

D. Computation Model

The tasks in the queue are expected to be executed timely and efficiently, i.e., with small latency and energy usage. Large delay reduces the usefulness of the in-vehicle application and user experience. High energy consumption quickly drains the VT's battery. They can both be considered as the cost of task execution. The demanded time and energy for conducting the calculation of each task, either locally on the LPU or remotely on the MEC server, are presented as follows.

1) Local Execution Model: Assume that the scheduler determines to schedule task J_{i} to LE at time slot t_i^{a}. Then starting from t_i^{a}, all of J_i's CPU cycles, i.e., $d_{\mathrm{i}}k_i$, will be executed on the LPU. The required number of time slots to complete J_i on the LPU is

$$t_i^{\mathrm{l}} = \left\lceil \frac{d_i k_i}{f^{\mathrm{l}}} \right\rceil \tag{4}$$

where f^{l} (in cycle/slot) is the CPU frequency of the LPU.

Following [20] and [21], the average power consumption, p^{l} (in Joule/slot), can be modeled by a super-linear function of CPU frequency as $p^{\mathrm{l}} = \xi \cdot (f^{\mathrm{l}})^{\nu}$, where ξ and ν are both constants. Hence, the energy consumed for executing J_i locally can be obtained by

$$e_i^{\mathrm{l}} = p^{\mathrm{l}} t_i^{\mathrm{l}}. \tag{5}$$

2) Remote Execution Model: If task J_i is scheduled for offloading at time slot t_i^{a}, the DTU starts to deliver the task to the serving RSU at time slot t_i^{a}. Upon successfully receiving all the input data, the MEC server executes the task remotely for the VT. The total time consumption consists of two parts: 1) time for wireless data transmission and 2) time for task computation on the MEC server. The former is related to the scheduled starting time t_i^{a} since the transmission data rate at each time slot depends on the location of the VT, as presented in Section II-C. The number of time slots for completing the delivery of the d_i bits data through V2I communication can be derived according to (3) as

$$t_i^{\mathrm{tx}}(t_i^{\mathrm{a}}) = t^{\mathrm{tx}}\big(d_i, t_i^{\mathrm{a}}\big). \tag{6}$$

In addition, the number of time slots required for executing J_i on the MEC server can be calculated by

$$t_i^{\mathrm{exe}} = \left\lceil \frac{d_i k_i}{f^{\mathrm{s}}} \right\rceil \tag{7}$$

where f^{s} is the CPU frequency reserved for the VT by the service provider. Hence, the total time slots consumed for offloading J_i can be obtained by

$$t_i^{\mathrm{o}}\big(t_i^{\mathrm{a}}\big) = t_i^{\mathrm{tx}}\big(t_i^{\mathrm{a}}\big) + t_i^{\mathrm{exe}}. \tag{8}$$

From VT's perspective, RE of tasks does not consume its energy. The energy usage for offloading J_i is only for data transmission and is given by

$$e_i^{\mathrm{o}}\big(t_i^{\mathrm{a}}\big) = p^{\mathrm{tx}} t_i^{\mathrm{tx}}\big(t_i^{\mathrm{a}}\big) \tag{9}$$

where p^{tx} (in Joule/slot) is the transmission power of the VT.

E. Objective

As the VT drives along the expressway, the scheduler continuously determines "where" and "when" to execute the tasks waiting in its queue. For each task J_i, the former refers to whether it should be computed locally on the LPU (denoted by a binary indicator $a_i = 0$) or offloaded to the MEC server for RE (denoted by $a_i = 1$). The latter refers to the starting time of the local calculation or offloading operation (denoted by integer t_i^{a}).

We define the *latency* experienced by task J_i as the total time duration (time slots) between the time instant that J_i is generated [i.e., t_i^{g} in (1)] and the time instant that the execution of J_i is completed. Therefore, the latency can be obtained as a function of the scheduling decisions a_i and t_i^{a}

$$\ell_i\big(a_i, t_i^{\mathrm{a}}\big) = t_i^{\mathrm{a}} + t_i^{\mathrm{e}}\big(a_i, t_i^{\mathrm{a}}\big) - t_i^{\mathrm{g}} \tag{10}$$

where $t_i^{\mathrm{a}} - t_i^{\mathrm{g}}$ represents the time duration that J_i spends to wait in the queue, and

$$t_i^{\mathrm{e}}(a_i, t_i^{\mathrm{a}}) = (1 - a_i)t_i^{\mathrm{l}} + a_i t_i^{\mathrm{o}}\big(t_i^{\mathrm{a}}\big) \tag{11}$$

is a unified expression of completing the execution of J_i using (4) and (8).

Similarly, combining (5) and (9), the *energy consumption* for executing J_i can be defined as a function of the scheduling decisions a_i and t_i^{a}

$$\varepsilon_i\big(a_i, t_i^{\mathrm{a}}\big) = (1 - a_i)e_i^{\mathrm{l}} + a_i e_i^{\mathrm{o}}\big(t_i^{\mathrm{a}}\big). \tag{12}$$

Therefore, the *cost* of making the scheduling decisions for task J_i is defined as a tradeoff between the resulting task latency and energy consumption [20]–[22]

$$c_i\big(a_i, t_i^{\mathrm{a}}\big) = \alpha \ell_i\big(a_i, t_i^{\mathrm{a}}\big) + \beta \varepsilon_i\big(a_i, t_i^{\mathrm{a}}\big) \tag{13}$$

where weighting coefficients α and β can be chosen to reflect designing preference toward smaller time or energy usage.

The overall objective of our system design is to find the optimal computation offloading scheduling strategy for the scheduler, in order to minimize the *long-term cost*, i.e., the average cost of making scheduling decisions for all the tasks generated by the VT

$$\underset{\boldsymbol{a}, \boldsymbol{t}^{\mathrm{a}}}{\text{minimize}} \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} c_i\big(a_i, t_i^{\mathrm{a}}\big) \tag{14}$$

where $\boldsymbol{a} = [a_1, a_2, \dots, a_n]$ and $\boldsymbol{t}^{\mathrm{a}} = [t_1^{\mathrm{a}}, t_2^{\mathrm{a}}, \cdots, t_n^{\mathrm{a}}]$. The term $\sum_{i=1}^{n} c_i(a_i, t_i^{\mathrm{a}})$ is the weighted sum of time and energy consumption of all the tasks.

Solving the problem (14) is very challenging due to the impacts of the random task generation process and dynamic wireless transmission environment. Conventional optimization techniques are hence hard to apply. In this article, we propose to apply DRL to identify the optimal offloading scheduling policy. We will show that our method can efficiently and effectively estimate the unknown system dynamics, and then properly make the scheduling decisions to achieve the expected long-term objective.

III. Deep Reinforcement Learning Background

DRL is the enhancement of reinforcement learning (RL), with DNN for state representation or function approximation [11], [12]. In an RL problem, an agent interacts with the environment over time. At each time step n, the agent observes an environment state s_n in the state space $\mathcal{S}$, and selects an action a_n from the action space $\mathcal{A}$, following the policy $\pi(a_n|s_n)$, which is the probability of taking action a_n when observing state s_n. Then, the environment transits to the next state $s_{n+1} \in \mathcal{S}$ and emits the reward signal r_n to the agent, according to the environment dynamics $P(s_{n+1}|s_n, a_n)$ and the reward function $R(s_n, a_n, s_{n+1})$, respectively. This process continues indefinitely unless the agent observes a terminal state (in an episodic problem). The accumulated reward of the agent from state s_m is defined as

$$G_m = \sum_{l=0}^{\infty} \gamma^l r_{m+l} \tag{15}$$

where $\gamma \in (0, 1]$ is called the discount factor. Such a problem is usually formulated as an MDP, defined as a 5-tuple, i.e., $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$.

The value function $v_\pi(s) = \mathbb{E}_\pi[G_m|s_m = s]$ is the expectation of accumulated reward following policy π from state s. The action value function $q_\pi(s, a) = \mathbb{E}_\pi[G_m|s_m = s, a_m = a]$ is the expectation of accumulated reward for selecting action a at state s and then following policy π. They, respectively, indicate how good a state and a state–action pair is, and are connected via $v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s, a)$. The objective of RL is to find the optimal policy π^*, to maximize the expectation of accumulated reward from any state in the state space, i.e.,

$$v_{\pi^*}(s) = \max_{\pi} v_\pi(s) = \max_{a} q_{\pi^*}(s, a), \quad s \in \mathcal{S}. \tag{16}$$

DRL uses DNNs to approximate the policy and/or value function. With the powerful representation capability of DNN, large state space can be supported. Current DRL methods can be classified into two categories: 1) value-based methods and 2) policy-based methods.

Value-based DRL methods adopt DNNs to approximate the value function (called value network), e.g., deep Q-network (DQN) [23] and double DQN [24]. Generally, the core idea of value-based DRL methods is to minimize the difference between the value network and the real value function. A natural objective function can be written as

$$L^{\mathrm{V}}(\theta) = \mathbb{E}_n[v_{\pi^*}(s_n) - v(s_n; \theta)]^2 \tag{17}$$

where $v(\cdot; \theta)$ is the value network, and θ is the set of its parameters. $v_{\pi^*}(\cdot)$ represents the real value function, which is unknown but is estimated by different value-based RL methods. The expectation $\mathbb{E}_n[\cdot]$ indicates the empirical average over a finite batch of samples in an algorithm that alternates between sampling and optimization.

Policy-based DRL methods use DNNs to approximate the parameterized policy (called policy network), e.g., REINFORCE [25] and actor–critic [26]. Comparing with value-based DRL methods, these methods have better convergence properties and can learn stochastic policies. Policy-based DRL methods work by computing an estimator of the policy gradient, and the most commonly used gradient estimator has form

$$\nabla L^{\mathrm{PG}}(\theta) = \mathbb{E}_n\Big[\nabla_\theta \log \pi(a_n|s_n; \theta)\hat{A}_n\Big] \tag{18}$$

where π is a stochastic policy and $\hat{A}_n$ is an estimator function at time step n.

Traditional policy-based DRL methods, such as REINFORCE, have two main defects: 1) the Monte Carlo sampling (to obtain G_n) brings high variance, leading to slow learning and 2) the on-policy (training and sampling using the same policy) update can easily converge to a local optimum.

Recently, the generalized advantage estimation (GAE) is proposed to make a compromise between variance and bias [27]. The GAE estimator is written as

$$\hat{A}_n^{\mathrm{GAE}(\gamma,\phi)} = \sum_{l=0}^{\infty}(\gamma\phi)^l \eta_{n+l}^v \tag{19}$$

where ϕ is used to adjust the bias–variance tradeoff, and

$$\eta_n^v = r_n + \gamma v(s_{n+1}; \omega) - v(s_n; \omega). \tag{20}$$

To alleviate the local optimum problem, off-policy learning is introduced to increase the exploration ability of policy gradient methods. The PPO algorithm proposed by OpenAI is recently the state-of-the-art [18]. Its objective function is

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}_n\Big[\min\Big(r_n(\theta)\hat{A}_n, \mathrm{clip}(r_n(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_n\Big)\Big] \tag{21}$$

where $r_n(\theta)$ is the policy probability ratio defined as

$$r_n(\theta) = \frac{\pi(a_n|s_n; \theta)}{\pi(a_n|s_n; \theta_{\mathrm{old}})}. \tag{22}$$

The clip function $\mathrm{clip}(r_n(\theta), 1-\epsilon, 1+\epsilon)$ constrains the value of r_n, which removes the incentive for moving r_n outside the interval $[1-\epsilon, 1+\epsilon]$, with ϵ being a hyperparameter to control the clip range. By taking the minimum of the clipped and unclipped objective, the final objective is restricted as a lower bound to the unclipped objective. Because of these advantages, in this article, we design our DRLOSM based on PPO.

IV. MDP Formulation

We propose to apply DRL to solve the considered offloading scheduling problem. To this end, we first formulate an MDP that can adequately describe the offloading scheduling process and then use DRL to find the optimal policy for the MDP. In what follows, each element of the MDP is presented.

A. State Space

At the beginning of each time slot, the scheduler monitors the system state, based on which the offloading scheduling decision is made. We defined the state space of our MDP as

$$\mathcal{S} \triangleq \Big\{s | s = \Big(\mathbf{Q}, s^{\mathrm{lpu}}, s^{\mathrm{dtu}}, s^{\mathrm{mec}}, x, d\Big)\Big\} \tag{23}$$

where each state s uses $3Q+5$ parameters (dimensions) to represent the status of VT's task queue ($\mathbf{Q}$), the LPU (s^{lpu}), the DTU (s^{dtu}), the MEC server (s^{mec}), and finally the wireless environment ($\{x, d\}$). As we stated in Section II, at each time slot t, $\mathbf{Q}$ is a $Q \times 3$ matrix, the jth row of which clarifies the task generation time, input data size, and CVR of the task $\mathbf{Q}[t]\langle j\rangle$. x and d are the current location of the VT and its distance to the serving RSU, which determines the statistics of the current V2I transmission data rate. The remaining state parameters s^{lpu}, s^{dtu}, and s^{mec} are elaborated as follows.

1) LPU State s^{lpu}: We use s^{lpu} to describe the number of *remaining CPU cycles* that the LPU requires to complete the task currently running on it. Specifically, assume that the scheduler decides to start operating the task J_i on the LPU from a certain time slot t_1. Then at the beginning of the t_1th time slot, $s^{\mathrm{lpu}}[t_1]$ is initialized to $d_i k_i$, which is the total number of CPU cycles demanded by J_i. Afterward, during each time slot $t \geq t_1$, the LPU provides f^{l} CPU cycles (and consumes p^{l} Joules of energy) to execute the task. The value of s^{lpu} is thus reduced by f^{l}, until the computation is finished and s^{lpu} is set to 0. Such an LPU state updating process (for a single task) can be described by

$$s^{\mathrm{lpu}}[t] = \max\Big\{s^{\mathrm{lpu}}[t-1] - f^{\mathrm{l}}, 0\Big\}, \quad t > t_1. \tag{24}$$

When s^{lpu} reaches 0, the LPU is said to be *idle* and is able to host a new task.

2) DTU State s^{dtu}: Parameter s^{dtu} describes the amount of *remaining data volume* of the task that needs to be transmitted to the MEC server by the DTU. Now, suppose that the scheduler decides to offload task J_i, starting from time slot t_2. This leads to $s^{\mathrm{dtu}}[t_2] = d_i$. Then in the tth time slot ($t \geq t_2$), up to $r[t]$ bits data can be delivered from the VT to the RSU (consuming p^{tx} Joules of energy), so that the value s^{dtu} is decreased by $r[t]$. Consequently, the DTU state updating process (for a single task) is given by

$$s^{\mathrm{dtu}}[t] = \max\left\{s^{\mathrm{dtu}}[t-1] - r[t-1], 0\right\}, \quad t > t_2 \tag{25}$$

until s^{dtu} reaches 0.

It is worth noting that the above process can be interrupted by a handover event. For instance, at a certain time slot t_3 ($t_3 \geq t_2$), the VT enters the coverage region of a new RSU, but the transmission of J_i has not yet finished, i.e., $s^{\mathrm{dtu}}[t_3] \neq 0$. As we mentioned earlier, in this case, the retransmission of the whole task will be automatically conducted. Therefore, $s^{\mathrm{dtu}}[t_3]$ is reinitialized to d_i immediately, and afterward the data transmission continues until $s^{\mathrm{dtu}} = 0$. Since the retransmission causes waste of both time and energy, a good offloading scheduling policy should try to avoid it in its decision-making process.

3) MEC Server State s^{mec}: Similar to the LPU state, the value of s^{mec} indicates the number of remaining CPU cycles that the MEC server requires to execute for the current offloaded task. Assume that at a specific time slot t_4, the transmission of J_i is found completed (i.e., $s^{\mathrm{dtu}}[t_4]$ reaches zero). Then, the MEC server state is initialized to $s^{\mathrm{mec}}[t_4] = d_i k_i$ immediately. In each time slot, the MEC server provides f^{s} CPU cycles (no energy consumption from the VT's perspective) for task computation and thus reduces the value of s^{mec} by f^{s}. This leads to the MEC server state satisfying

$$s^{\mathrm{mec}}[t] = \max\left\{s^{\mathrm{mec}}[t-1] - f^{\mathrm{s}}, 0\right\}, \quad t > t_4. \tag{26}$$

At any time slot t, only when both $s^{\mathrm{dtu}}[t] = 0$ and $s^{\mathrm{mec}}[t] = 0$, we say that the DTU is idle so that a V2I transmission process can be activated.[3] Otherwise, if either $s^{\mathrm{dtu}}[t] \neq 0$ or $s^{\mathrm{mec}}[t] \neq 0$, tasks cannot be scheduled for offloading.

The ($3Q+5$)-dimensional state space $\mathcal{S}$ defined in (23) completely describes the characteristics of the environment that our agent (i.e., the scheduler) interacts with. Due to the fact that each dimension parameter has a large value range (i.e., $x \geq 0$, $0 \leq d \leq L/2$, $s^{\mathrm{lpu}} \geq 0$, $s^{\mathrm{dtu}} \geq 0$, and $s^{\mathrm{mec}} \geq 0$), our MDP actually has an enormous state space.

B. Action Space

The action space of our MDP includes three *types* of scheduling actions: 1) LE; 2) RE; and 3) holding on (HO). LE and RE are used to conduct the "where to schedule" operation, and HO is specially designed to carry out the "when to schedule" operation. Each type of action includes multiple individual actions. They are elaborated as follows.

[3]In this article, for simplicity, the situation that a MEC server can still receive data while executing another offloaded task is not considered. Hence, only when both $s^{\mathrm{dtu}}[t] = 0$ and $s^{\mathrm{mec}}[t] = 0$, offloading can be performed.

1) LE Action Type: This type of actions schedule tasks waiting in the VT's queue to the LPU. The complete set of such actions is defined as

$$\mathcal{LE} \triangleq \left\{\mathrm{LE}_1, \mathrm{LE}_2, \ldots, \mathrm{LE}_Q\right\} \tag{27}$$

where LE_i ($i \in \{1, 2, \ldots, Q\}$) is dedicated to the ith task in the queue. For instance, at the beginning of a certain time slot t_1, the scheduler decides to take action LE_i (this is possible only when the LPU is currently idle and the ith task in the queue exists, i.e., $s^{\mathrm{lpu}}[t_1] = 0$ and $i \leq q[t_1]$). By taking LE_i, task $\mathbf{Q}[t_1]\langle i\rangle$ is sent to the LPU, which changes the state of the LPU ($s^{\mathrm{lpu}}[t_1]$) and the task queue ($\mathbf{Q}[t_1]$) immediately. Specifically, $s^{\mathrm{lpu}}[t_1]$ is set to $d_i k_i$, and in VT's queue, the tasks after $\mathbf{Q}[t_1]\langle i\rangle$ are shifted forward to fill the empty position. Afterward, the LPU state updates slot by slot as mentioned in Section IV-A1.

2) RE Action Type: When the DTU is idle and the queue is not empty, an RE action can be taken to offload the specified task. We define all the actions of this type as a set, i.e.,

$$\mathcal{RE} \triangleq \left\{\mathrm{RE}_1, \mathrm{RE}_2, \ldots, \mathrm{RE}_Q\right\}. \tag{28}$$

By taking RE_j at time slot t_2, task $\mathbf{Q}[t_2]\langle j\rangle$ is sent to the DTU for uploading. The DTU state is set as $s^{\mathrm{dtu}}[t_2] = d_j$, and the task queue state $\mathbf{Q}[t_2]$ is updated similar to the LE action type as introduced before. Then, the states of the DTU and MEC server change with time according to Sections IV-A2 and IV-A3, respectively, without taking any action.

3) HO Action Type: The HO actions are responsible for postponing task scheduling. The set of actions that perform such a function is defined as

$$\mathcal{HO} \triangleq \{\mathrm{HO}_1, \mathrm{HO}_2, \ldots\} \tag{29}$$

where HO_w ($w \in \mathbf{N}^+$) means that the scheduler determines to keep all waiting tasks to stay in the queue for w time slots, even though the LPU and/or DTU are capable of accepting the computation or transmission of jobs. For example, at time slot t_5, an action HO_w is taken. Then in the next w time slots (from time slot t_5 to time slot $t_5 + w - 1$), the scheduler does not carry out any offloading decisions (the tasks scheduled for local or remote operations before time slot t_5 are not affected). Certainly conducting an HO action increases the latency of all tasks waiting in the queue. However, if the wireless condition is poor and offloading causes unnecessarily large delay and/or energy consumption, properly postponing the task scheduling process to wait for better transmission opportunities would be worthwhile.

4) Action Space and Action Legitimacy: With the three types of actions defined above, the complete action space of our MDP, $\mathcal{A}$, is defined as the union of the three sets $\mathcal{LE}$, $\mathcal{RE}$, and $\mathcal{HO}$, i.e.,

$$\mathcal{A} \triangleq \mathcal{LE} \cup \mathcal{RE} \cup \mathcal{HO}. \tag{30}$$

However, for each system state $s \in \mathcal{S}$, the set of actions that the scheduler can take, $\mathcal{A}(s)$, may be only a subset of $\mathcal{S}$. Throughout this article, we term the actions included in $\mathcal{A}(s)$ as *legal actions* for state s, and the remaining in $\mathcal{A}$ [i.e., those included in the complementary set of $\mathcal{A}(s)$] as *illegal actions*.

TABLE I
LEGITIMACY OF EACH ACTION TYPE

Situation	Queue	LPU	DTU	$\mathcal{HO}$	$\mathcal{LE}$	$\mathcal{RE}$
(1)	Not Empty	Idle	Idle	✓	✓	✓
(2)	Not Empty	Idle	Busy	✓	✓	×
(3)	Not Empty	Busy	Idle	✓	×	✓
(4)	Not Empty	Busy	Busy	×	×	×
(5)	Empty			×	×	×

✓: Legal; ×: Illegal.

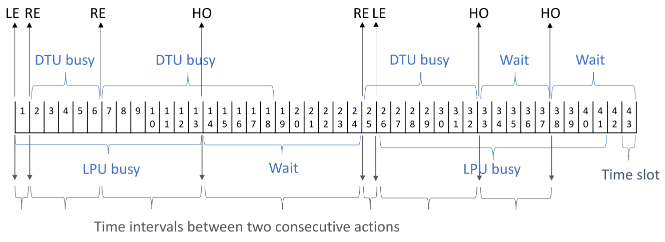


Fig. 2. Example task scheduling timeline.

Specifically, at the beginning of each time slot t, the scheduler monitors the system state s and determines whether a scheduling action should be taken. Consider the case that a total of $q[t]$ tasks are waiting in the queue to be scheduled. As mentioned earlier, if the current LPU is idle ($s^{\text{lpu}}[t] = 0$), LE actions in the set $\{\text{LE}_1, \text{LE}_2, \cdots, \text{LE}_{q[t]}\}$ and all HO actions are legal actions. But the LPU being busy ($s^{\text{lpu}}[t] > 0$) causes all actions in $\mathcal{LE}$ to be illegal. Similarly, if the current DTU is idle ($s^{\text{dtu}}[t] = 0$ and $s^{\text{mec}}[t] = 0$), RE actions in the set $\{\text{RE}_1, \text{RE}_2, \cdots, \text{RE}_{q[t]}\}$ and all HO actions are legal. When the DTU becomes busy ($s^{\text{dtu}}[t] > 0$ or $s^{\text{mec}}[t] > 0$), all actions in $\mathcal{RE}$ would be illegal. Finally, when there is no task in the queue (i.e., $\mathbf{Q}[t]$ is an all-zero matrix) or no executing resource (i.e., both the LPU and DTU are busy), no action can be taken (i.e., $\mathcal{A}(s) = \varnothing$). Table I provides a summary of the legitimacy regarding each type of actions.

It is worth noting that, due to the dynamic wireless environment, the diversity of offloading tasks, and the potential of task parallel executing (both locally and remotely), the time interval between two consecutive actions in fact varies. This is different from most conventional MDP modeling for MEC, which assumes a fixed interdecision time duration [15], [16].

Fig. 2 gives an example of a task scheduling timeline in our MDP. Consider the case that the VT's task queue is not empty, and the LPU, DTU, and MEC states are all initialized to be zero. At time slot 1, both the LPU and the DTU are idle, which is situation (1) in Table I. The scheduler chooses an LE action from the legal action space $\mathcal{A}(s)$, and the associated task is sent to the LPU. At time slot 2, since the LPU is busy, actions in $\mathcal{LE}$ are excluded from the legal action space (situation (3) in Table I). The scheduler takes an RE action to offload a task for RE, and the DTU becomes busy, too. The time slots 3–6 represent situation (4) in Table I. Due to the lack of computation and communication resources, no action can be taken. At time slot 7, the scheduler observes that the DTU is idle and immediately takes another RE action. At time slot 13, the LPU is found to be idle [situation (2) in Table I], but the scheduler considers it to be improper to locally compute any of the tasks in the queue (e.g., because all tasks require a large number of CPU cycles). An HO action for postponing the task scheduling for 11 time slots is adopted. As a result, before the 25th time slot, no action can be taken even when the DTU becomes idle from time slot 19. After the HO operation completes, the decision-making process continues in the similar fashion.

Although the system has a different state in each time slot, the scheduler's actions would actively affect the system state only on those time slots that the action is taken (e.g., the time slots 1, 2, 7, and 14 in Fig. 2). The change of system states on other time slots (termed intermediate states) is caused by environmental factors, such as new arriving tasks (changing queue state $\mathbf{Q}$), computation with local CPU (changing LPU state s^{lpu}), successful or failing (due to handover) V2I transmission (changing DTU state s^{dtu}), computation with remote CPU (changing MEC sever state s^{mec}), and mobility of VT (changing x and d). Therefore, the intermediate states are treated to be part of the environment and are not explicitly considered in our MDP. In other words, the actual state space $\mathcal{S}$ in our MDP includes only the states when actions can be taken. It is not difficult to see that, even though the intermediate states are excluded, the state space $\mathcal{S}$ is still extremely large.

C. Rewards

Suppose that at time slot t_n^{a}, the scheduler takes the nth action $a_n \in \mathcal{A}$ on state s_n ($s_n \in \mathcal{S}$). Then, after t_n^{b} time slots, it comes to a new state s_{n+1} ($s_{n+1} \in \mathcal{S}$), based on which a new action needs to be taken (at time slot t_{n+1}^{a}). We define the reward function $R(s_n, a_n, s_{n+1})$ based on the time and energy consumption of all the tasks in the VT within the time interval of this state transition. Note that the time interval of state transition, i.e., t_n^{b}, varies based on different states and actions, but is easy to obtain ($t_n^{\text{b}} = t_{n+1}^{\text{a}} - t_n^{\text{a}}$).

At each time slot t, if an arriving task is not finished, its latency would increase by one time slot. Hence, the total time delay (in slot) of all the tasks in the VT at time slot t is denoted as

$$\Delta_\ell^{\text{s}}(t) = q[t] + \mathbf{1}_{\{s^{\text{lpu}}[t]>0\}} + \mathbf{1}_{\{s^{\text{dtu}}[t]>0\}} + \mathbf{1}_{\{s^{\text{mec}}[t]>0\}}. \tag{31}$$

In (31), if a task is running on the LPU, then $\mathbf{1}_{\{s^{\text{lpu}}[t]>0\}} = 1$, and if a task is in the process of offloading, either $\mathbf{1}_{\{s^{\text{dtu}}[t]>0\}}$ or $\mathbf{1}_{\{s^{\text{mec}}[t]>0\}}$ is 1. Therefore, the *total* time delay of all the tasks in the VT from state s_n to s_{n+1} after taking action a_n can be calculated by

$$\Delta_\ell(s_n, a_n, s_{n+1}) = \sum_{t=t_n^{\text{a}}}^{t_{n+1}^{\text{a}}-1} \Delta_\ell^{\text{s}}(t). \tag{32}$$

The energy consumption is only associated with the task local computation and input data transmission. Hence, the total energy consumption of all the tasks in the VT at time slot t can be calculated by

$$\Delta_e^{\text{s}}(t) = p^{\text{l}} \cdot \mathbf{1}_{\{s^{\text{lpu}}[t]>0\}} + p^{\text{tx}} \cdot \mathbf{1}_{\{s^{\text{dtu}}[t]>0\}}. \tag{33}$$

Therefore, the *total* energy consumption of all the tasks in the VT from state s_n to s_{n+1} after taking action a_n is given by

$$\Delta_e(s_n, a_n, s_{n+1}) = \sum_{t=t_n^{\mathrm{a}}}^{t_{n+1}^{\mathrm{a}}-1} \Delta_e^{\mathrm{s}}(t). \tag{34}$$

Finally, considering the fact that our system has a dynamic workload, if the task arriving rate λ is relatively large compared with scheduling speed, overflow events may possibly occur and new incoming tasks have to be discarded. This issue brings bad user experience on vehicular applications. Hence, we consider a cost induced by task queue overflow in our MDP, $\Delta_o(s_n, a_n, s_{n+1})$, which denotes the number of overflowed tasks from state s_n to s_{n+1} after taking action a_n.

Therefore, the overall cost brought by action a_n can be considered as a weighted sum of $\Delta_\ell(s_n, a_n, s_{n+1})$, $\Delta_e(s_n, a_n, s_{n+1})$, and $\Delta_o(s_n, a_n, s_{n+1})$

$$\mathrm{cost}(s_n, a_n, s_{n+1}) \triangleq \alpha\Delta_\ell(s_n, a_n, s_{n+1}) + \beta\Delta_e(s_n, a_n, s_{n+1}) + \zeta\Delta_o(s_n, a_n, s_{n+1}). \tag{35}$$

The weighting parameters α and β can be properly chosen to reflect the user preference toward smaller delay or lower energy usage in the offloading scheduling policy design, and ζ is the penalty parameter of task overflow.

The reward function of our MDP, $R(s_n, a_n, s_{n+1})$, is defined as the negative of the cost function, indicating how good this transition is, i.e.,

$$R(s_n, a_n, s_{n+1}) \triangleq -k_s \cdot \mathrm{cost}(s_n, a_n, s_{n+1}) \tag{36}$$

where the constant parameter k_s is chosen to scales the value range of the reward.

Starting from an initial state $s_m \in \mathcal{S}$, the scheduler can interact with the environment following a specific stochastic offloading scheduling policy, $\pi(a_n|s_n)$. Then, a Markov chain, i.e., $s_m, a_m, s_{m+1}, a_{m+1}, \ldots,$ is obtained. The accumulated reward can be written as

$$G_m = \sum_{n=0}^{\infty} \gamma^n R(s_{m+n}, a_{m+n}, s_{m+n+1}), \quad s_m \in S. \tag{37}$$

We can find that G_m is the weighted sum of time and energy consumption of all the tasks if the discount factor γ is set 1 and no task overflow happens. Therefore, the goal of finding the optimal offloading scheduling policy π^* is consistent with the original objective introduced in Section II-E.

The enormous state and action space coupled with the highly dynamic environment make it difficult to obtain the environment dynamics, i.e., $P(s_{n+1}|s_n, a_n)$. In the next section, we resort to the model-free DRL to search for the optimal offloading scheduling policy.

V. DRL-Based Offloading Scheduling

Our DRLOSM is described in this section. We first introduce the architecture of the DNN used to approximate the offloading scheduling policy, and then design our training method based on PPO to train the policy network. Afterward, several methods to improve training efficiency are presented.

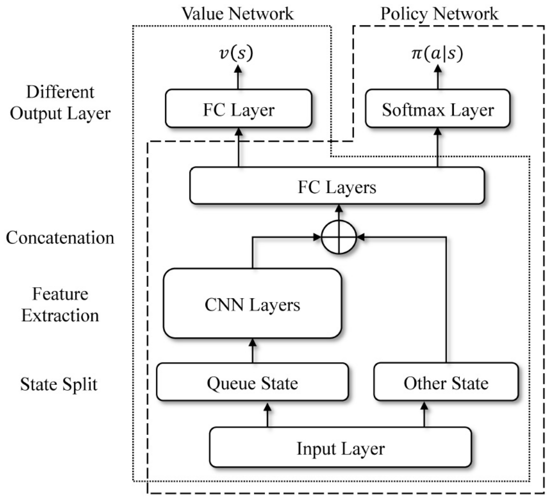


Fig. 3. DNN architecture design.

A. Network Architecture

Our DNN architecture design is illustrated in Fig. 3. Two functions are needed in the training process: 1) the offloading scheduling policy $\pi(a_n|s_n;\theta)$, which is the target of learning and 2) the value function $v(s_n;\omega)$, which is used for advantage estimation. Both functions take the environment state $s \in \mathcal{S}$ as input but are with different outputs. Considering the fact that features useful for estimating the value function could also help to select actions and *vice versa*, we utilize a parameter-shared DNN architecture to simultaneously approximate both the policy and value function. Specifically, the policy network and the value network share most of the network structure (network layers and corresponding parameters). The difference lies in the output layers after shared fully connected (FC) layers. For the policy network, a softmax layer outputs the probability distribution of all actions. For the value network, an FC layer outputs the state value.

As we explained in Section IV-A, the state space $\mathcal{S}$ of the considered MDP is very large. Using a single FC network structure for feature extraction can lead to a considerably inefficient training process. We observe that for each state $s = (\mathbf{Q}, s^{\mathrm{lpu}}, s^{\mathrm{dtu}}, s^{\mathrm{mec}}, x, d)$, most parameters are used to describe the state of the task queue, i.e., $\mathbf{Q}$. Since $\mathbf{Q}$ has a fixed matrix form and the data stored in it are structured, we propose to embed a CNN in the DNN architecture for efficient extraction of representative features of the task queue. To this end, as shown in Fig. 3, the queue state is first separated from the input state and is sent to CNN layers. The outputs of the CNN layers (representative features of the task queue) are concatenated with the remaining state information before being input to the FC layers for facilitating function approximations. In Section VI, we will show through experiments that such a network structure can significantly improve training efficiency over using only FC layers.

In our experiments, we consider the case $Q = 20$. Table II shows the shared structure in our CNN-embedded DNN architecture. Four convolutional layers are used to extract features in the queue. Note that the input data size of the first

TABLE II
ARCHITECTURE OF THE SHARED DNN

Layer	Kernels	Stride	Output	Remark
Input			All states	
Split			$20 \times 2 \times 1$, other states	Cache other states
Conv2D	$2 \times 2 \times 8$	1	$19 \times 1 \times 8$	Resize, No padding
Conv1D	4×16	1	16×16	No padding
Conv1D	4×16	2	7×16	No padding
Conv1D	4×20	1	4×20	No padding
Concat			$4 \times 20 \oplus$ other states	Concat other states
FC layer			512	Layer Norm. [28]
FC layer			512	Layer Norm.

Leaky-ReLU is adopted as the activation function for all neurons.

Algorithm 1 Training Algorithm Based on PPO

```
1: Initialize the DNN parameter $\theta$ randomly to obtain $\pi_\theta$
2: Initialize the sampling policy $\pi_{\theta_{\text{old}}}$ with $\theta_{\text{old}} \leftarrow \theta$
3: for $iteration = 1, 2, \ldots$ do
4:     /* Sampling (exploring) with $\pi_{\theta_{\text{old}}}$ */
5:     for $i = 1, 2, \ldots, N$ do
6:         Sample a whole episode in the environment with $\pi_{\theta_{\text{old}}}$, and store the trajectory in $T_i$
7:         Compute advantage estimates $\hat{A}_{ni}^{\text{GAE}(\gamma,\phi)}$ according to (19) for each time step $n$ in the $T_i$
8:     end for
9:     Cache all sampled data in two sets: T and A
10:    /* Optimizing $\pi_\theta$ (exploiting) */
11:    for $epoch = 1, 2, \ldots, K$ do
12:        Update the policy based on the objective function (38), i.e., $\theta \leftarrow \arg\max_\theta L^{\text{PPO}}(\theta)$, by Adam using the sampled data from T and A for one epoch
13:    end for
14:    Synchronise the sampling policy with $\theta_{\text{old}} \leftarrow \theta$
15:    Drop T and A
16: end for
```

convolutional layer is $20 \times 2 \times 1$ but not $20 \times 3 \times 1$ because we drop all the task generated time in **Q**. Since the time consumed for task waiting in the queue is considered in the reward signal of each time step, including this information is not helpful for our policy optimization.

B. Training Algorithm

Since the parameter-shared DNN architecture is adopted, the overall objective is a combination of the error terms of both the policy network and value network. We utilize GAE as the estimator function. The objective function of the policy network can be derived by substituting (19) and (22) into (21), and that of the value network can be obtained by (17). Therefore, the overall maximization objective function can be written as

$$L^{\text{PPO}}(\theta) = \mathbb{E}_n\Big[L_n^{\text{CLIP}}(\theta) - cL_n^{\text{V}}(\theta)\Big] \tag{38}$$

where c is the loss coefficient. The subscript n of L_n^{CLIP} and L_n^{V} means that we obtain their value without taking the expectation in (21) and (17) (we take the empirical average from a finite batch of samples after their combination).

The details of the proposed training algorithm based on PPO are presented in Algorithm 1. Two DNNs are initialized with the same parameter ($\theta_{\text{old}} \leftarrow \theta$), one for sampling ($\pi_{\theta_{\text{old}}}$), and the other for optimizing (π_θ). The algorithm alternates between sampling (lines 4–8) and optimization (lines 10–13). In the sampling stage, N trajectories [e.g., $(s_0, a_0, r_0, s_1, \ldots, s_{\text{terminal}})$] are sampled following the old policy $\pi_{\theta_{\text{old}}}$. For training efficiency, the GAEs for each time step n in each trajectory T_i, i.e., $\hat{A}_{ni}^{\text{GAE}(\gamma,\phi)}$, are computed in advance in this stage. The sampled data are cached for optimization (line 9). In the optimization stage, the parameter θ of the policy π_θ is updated for K epochs. In each epoch, we improve the policy π_θ by conducting stochastic gradient ascend on the cached sampled data based on the objective function, i.e., (38). After the optimization stage, we update the sampling policy $\pi_{\theta_{\text{old}}}$ with the current π_θ and drop the cached data (lines 14 and 15). Then, the next iteration begins.

It is worth noting that the random policy in the exploring stage cannot ensure the legitimacy of the chosen action according to the current state. Two types of illegal actions, as described in Section IV-B, can both be selected when sampling (line 6). We deal with this problem as follows. First, if an illegal action type is chosen (e.g., selecting LE action type when the LPU is busy), the system will neglect the illegal action, leaving the system state unchanged. Since an action is needed at this time, this will prompt the offloading scheduling policy to reschedule according to the same state. Thanks to the stochastic policy supported by the policy-based DRL, there are always opportunities for other action types to be selected. Then, the offloading process continues. To force our learning algorithm to effectively reduce the attempts to explore illegal actions, we add a penalty term, k_i, to the reward when an illegal action type is selected during the training process. Second, when the task specified by an LE$_j$ or RE$_j$ action does not exist, i.e., $j > q[t]$, for simplicity, we let the scheduler select the first task in the queue automatically.

C. Training Efficiency

So far, we have carefully designed our DNN architecture to improve the feature extraction process and applied the state-of-the-art PPO algorithm to enhance the training performance. However, the state space and action space of this MDP are both very large, leading to a huge exploration space. As a result, the training process is difficult to converge. To handle this issue, we adopt a series of methods to improve training efficiency.

First, we restrict the selection of HO actions. Two constant parameters are defined, i.e., p^{hmax} and p^{g}. p^{hmax} limits the maximum waiting time of an HO action, and p^{g} controls the granularity of HO actions. This changes the set $\mathcal{HO}$ defined in (29) as

$$\mathcal{HO} \triangleq \big\{\text{HO}_{p^{\text{g}}}, \text{HO}_{2p^{\text{g}}}, \ldots, \text{HO}_{np^{\text{g}}}\big\}, \quad np^{\text{g}} \le p^{\text{hmax}}. \tag{39}$$

Each time the scheduler takes an HO action, maximally the scheduling can be delayed by np^{g} time slots. If necessary, more HO actions can be taken to further postpone the decision-making procedure. To limit the maximum waiting time of an HO action does not change the original MDP, but significantly reduce the action space (the unlimited and discrete action space raises great challenges in implementing the DRL algorithm [29]). Furthermore, the length of each time slot can be very tiny, which makes nearly no difference between consecutive HO actions. Larger granularity can help the algorithm learn the difference between HO actions. By setting the granularity $p^{\mathrm{g}} = 1$, we derive the original MDP for modeling the desired offloading scheduling process described in Section II. Now, the total number of HO actions in $\mathcal{HO}$ becomes $\lceil p^{\mathrm{hmax}}/p^{\mathrm{g}} \rceil$.

To avoid the training process to explore the HO actions that hold the tasks for too long, which is clearly of little use, the continuous waiting time of the system is recorded. A penalty term k_h is applied when the waiting time exceeds a certain threshold value. In our experiments, this threshold is simply set to L/V, meaning that we do not encourage the VT to pass the complete coverage area of an RSU without executing any task. In practice, such a value can certainly be much smaller.

Second, we can further reduce the size of the action space by setting a constant parameter $p^{\mathrm{smax}} \leq Q$ and limit the number of possible LE and RE actions be at most p^{smax}. In other words, the sets defined in (27) and (28) are changed to

$$\mathcal{LE} \triangleq \{\mathrm{LE}_1, \mathrm{LE}_2, \ldots, \mathrm{LE}_{p^{\mathrm{smax}}}\} \quad (40)$$

$$\mathcal{RE} \triangleq \{\mathrm{RE}_1, \mathrm{RE}_2, \ldots, \mathrm{RE}_{p^{\mathrm{smax}}}\}. \quad (41)$$

Setting $p^{\mathrm{smax}} = Q$ leads to the original MDP. However, if the system workload is relatively light (i.e., λ is relatively small), for most of the time, the number of tasks actually waiting in the queue $q[t]$ would be much smaller than Q. A large number of actions would be illegal (i.e., $\{\mathrm{LE}_{q[t]+1}, \mathrm{LE}_{q[t]+2}, \ldots, \mathrm{LE}_Q\}$). Large number of illegal actions also leads to slow learning. In addition, since tasks are sorted by their generated time in the queue, tasks in the front of the queue should have a higher priority when scheduling. By choosing the value of p^{smax} to be smaller, the fairness of task scheduling is guaranteed, although with a certain sacrifice of achievable performance. As a result, the total number of actions in $\mathcal{A}$ becomes $2p^{\mathrm{smax}} + \lceil p^{\mathrm{smax}}/p^{\mathrm{g}} \rceil$.

Finally, having a large number of tasks waiting in the queue is considered to be an unwanted event, since it may lead to a higher probability of task overflow and inefficient exploration. To avoid such an event to occur, we add a penalty term to the reward according to the current queue length, as $k_q q[t]^u$, in which the k_q and u reflect our desire of the queue length. When $k_q = 0$, the learning objective is consistent with the original MDP problem. Choosing larger values of k_q and u results in a smaller number of waiting tasks and a more efficient training process.

VI. Performance Evaluation

In this section, extensive simulation experiments are conducted to evaluate the proposed DRLOSM. Our algorithm and network architecture are implemented using TensorFlow [30]. The main simulation parameters and training hyperparameters are listed in Tables III and IV, respectively.

TABLE III
Simulation Parameters

Parameter	Value
Length of time slot	0.01 sec
LPU's CPU frequency f^{l}	400 MHz
LPU power linear parameter ξ	1.25×10^{-26}
LPU power exponential parameter ν	3
Wireless transmission power p^{tx}	1.258 W
Size of task input data d_i	$[0.5, 3]$ MB
Computation-to-volume ratio k_i	$[100, 3200]$ cycles/byte
Size of VT's task queue Q	20
Selection range of each action p^{smax}	4
Specific waiting period for HO action	$\{0.2, 0.4, 0.6, 0.8\}$ sec
MEC server' CPU frequency f^{s}	2000 MHz
RSU's coverage region L	160 meter
Speed of VT V	20 meter/sec
Expected V2I data rates	$4, 8, 16, 32, 16, 8, 4$ Mbps
Standard deviation of V2I data rates	$\sqrt{3}$ Mbps

TABLE IV
Training Hyperparameters

Parameter	Value	Parameter	Value
Overflow coefficient ζ	0.5	Learning rate	10^{-4}
Optimization method	Adam [31]	Discount factor γ	0.99
Loss coefficient c	0.5	Clipping range ϵ	0.2
Adv. discount factor ϕ	0.95	Scaling parameter k_s	0.5
Penalty k_i	4	Penalty k_h	2
Penalty k_q	0.0025	Queue coefficient u	2

We set the length of each time slot as 0.01 s, and, for ease of understanding, use second (instead of time slot) in the unit of time-related parameters. The LPU's CPU frequency (f^{l}) and its power parameters (ξ and ν) are set according to [20]. Hence, the power consumption for local computing p^{l} can be obtained by $p^{\mathrm{l}} = \xi \cdot (f^{\mathrm{l}})^{\nu}$. The V2I transmission power is set to $p^{\mathrm{tx}} = 1.258$ W [20]. The data size d_i and CVR k_i for each task are both sampled from uniform distributions within the respective regions shown in Table III. The three parameters that affect the action space, i.e., the maximum waiting time slots of an HO action p^{hmax}, the granularity of an HO action p^{g}, and the selecting range of each action type p^{smax} are set as 80, 20, and 4, respectively. The specific waiting period for each HO action is shown in Table III.

The V2I transmission data rate at each time slot is considered to be a Gaussian random variable, whose expectation is related to the distance between the VT and its serving RSU. To simplify the experiment environment, we evenly divide the coverage region of each RSU into seven road segments. The expected data rate in each segment is considered to be the same. For the seven segments, they are set to be, from the left end to the right end, 4, 8, 16, 32, 16, 8, and 4 Mb/s, respectively, to reflect the fact that a higher data rate can be achieved when the VT is closer to the RSU.

Note that all the experiment environment setups, such as the channel statistics, CPU frequency of LPU and MEC server, system workload, and inter-RSU distance are unknown to the agent (i.e., the scheduler). But as long as the environment is fixed, our DRLOSM is capable of learning the optimal offloading scheduling policy by directly interacting with the environment. Hence, extending the experiment environment to more general cases is straightforward, e.g., more complicated

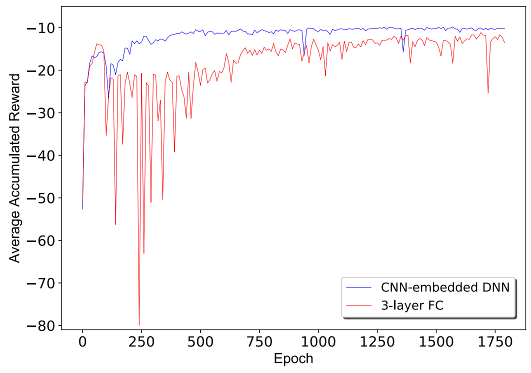


Fig. 4. Learning curves of different DNN architectures.

fading characteristics and different but fixed inter-RSU distances.

A. Convergence Performance

We first validate the convergence performance of the proposed DRLOSM by training it in the experiment environment. To balance the impacts of task latency and energy consumption on the cost function (13), the weighting coefficients α and β are set as 0.07 and 1, respectively (the quantity of task latency is much larger than the quantity of energy consumption).

Two different DNN architectures, acting as the offloading scheduling policies, are adopted in this experiment for comparison: 1) the proposed CNN-embedded DNN architecture (CNN-embedded DNN) and 2) a 3-layer FC DNN architecture (3-layer FC). FC DNN is one of the most classical and commonly used DNN architectures, which has been widely used in many studies, including our previous work [17]. In the 3-layer FC, each FC layer has 512 neurons with leaky-ReLU activation functions, and layer normalization is adopted. The two policy networks are trained for 1800 epochs. After every five training epochs, they are tested in the same environment, and the corresponding accumulated reward is recorded.

Fig. 4 shows the learning curves (average accumulated reward versus the number of training epochs) of the two DNN architectures. The proposed CNN-embedded DNN performs much better than the 3-layer FC. It obtains a higher average accumulated reward and converges faster (after around 500 training epochs) and more stably. At the same time, the size of the CNN-embedded DNN (319 848 training parameters) is also smaller than that of the 3-layer FC (560 140 training parameters). From the experiment, the advantage of embedding a CNN in the DNN architecture is exhibited.

This experiment also shows the considerable time and computational overhead of the proposed DRLOSM during training. Even for the CNN-embedded DNN, it takes about 5 hours to converge to a good solution on an NVIDIA GeForce GTX 1080 Ti. However, in our application scenario, the training process can be performed offline in the remote cloud. The VT only needs to execute the inference process to make offloading decisions. Considering the simple structure and small size of the CNN-embedded DNN, the inference process of DRLOSM is very efficient (only 1.5 ms on the NVIDIA GeForce GTX 1080 Ti), which can be readily supported by many practical embedded AI accelerators.

B. Performance in the Static Queue Scenario

Now, we evaluate the performance of our DRLOSM through comparisons with a number of baseline algorithms. We start from a relatively simple static queue scenario (SQS), in which no new task can be generated after the system initialization (i.e., the task arrival rate $\lambda = 0$). The following six baseline algorithms are considered.

1) *All Local Execution (AL):* All tasks are executed locally.
2) *All Offloading (AO):* All tasks are scheduled to offload to the MEC servers, regardless of the wireless condition.
3) *Random Offloading (RD):* All the actions in the action space are chosen randomly with the same probability.
4) *Time Greedy (TG):* When the LPU becomes idle, the task with the lowest CVR is immediately scheduled for LE. When the DTU becomes idle, the task with the highest CVR is immediately scheduled for offloading. The HO actions are avoided in this algorithm since they always increase the task latency.
5) *Energy Greedy (EG):* We assume that EG knows the expected V2I data rate on each road segment on the expressway. It offloads tasks only on the road sections with the best wireless condition, which brings the lowest energy consumption. If the energy consumption can be further reduced, EG can also schedule a task for LE (rather than offloading).

The above five baseline algorithms adopt predefined action rules (i.e., policy) to make offloading scheduling decisions. Different from the DRLOSM's policy obtained through learning, these predefined action rules are relatively naive and intuitive, making the decision process of the five algorithms even more efficient, but with less flexibility (only suitable for some special situations).

In addition to the above five intuitive methods, the genetic algorithm (GA) is adopted as another baseline algorithm. As a metaheuristic algorithm, GA is a practical solution for combinatorial optimization problems.

1) *GA*: The GA framework in DEAP is adopted to implement this baseline [32]. The scheduling plan is encoded into the chromosome of each individual, which is an action sequence for the scheduler to schedule the tasks in the queue. Each gene in the chromosome is an integer, indicating one of the scheduling actions. The length of each chromosome is based on the length of EG's action sequence, which is long enough to find the optimal solution. We assume that GA knows the wireless condition of each road segment. Hence, it can evaluate each individual by applying the action sequence in a simulated environment. As the algorithm progresses, individuals that obtain lower costs gain more opportunity for reproduction. When GA terminates, the most adaptable individual is selected as the final scheduling plan. The tuning parameters of GA are summarized in Table V.

TABLE V
PARAMETERS OF THE GA BASELINE

Parameter	Setting
Elite strategy	enable
Elite count	1
Population size	100
Maximum generations	100
Selection strategy	Tournament
Tournament size	3
Crossover probability	0.6
Crossover strategy	Two point crossover
Creation distribution	Uniform
Mutation probability	0.2
Independent mutation distribution	Uniform
Independent mutation probability	0.1

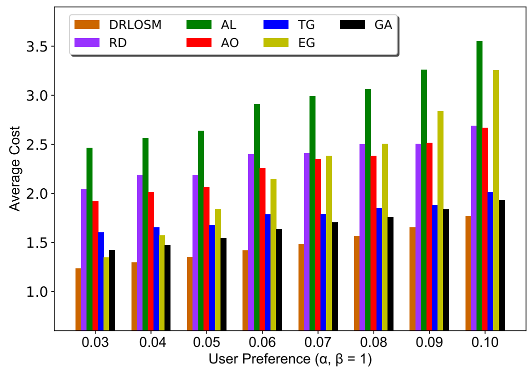


Fig. 5. Comparison of average cost in SQS.

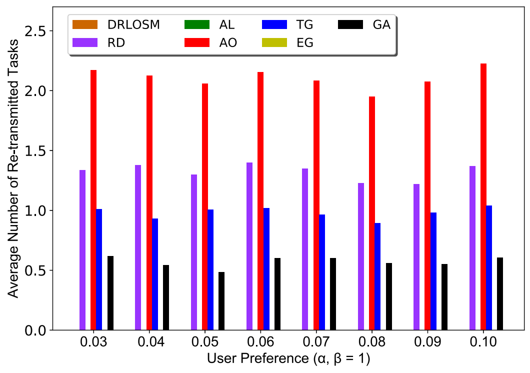


Fig. 6. Comparison of average number of retransmitted tasks in SQS.

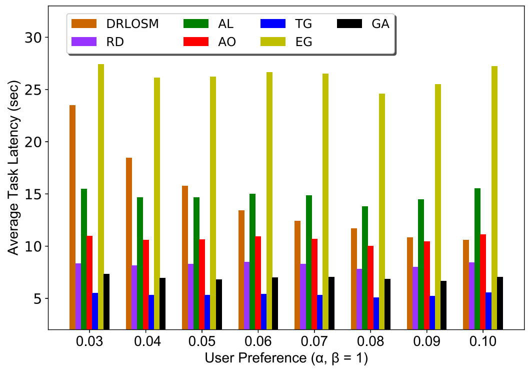


Fig. 7. Comparison of average task latency in SQS.

GA is a typical one-shot planning algorithm, which tries to compute the best offloading scheduling plan according to the current system state. However, once the system state, based on which the scheduling plan is made, changes (e.g., new tasks are generated), it should be re-executed. Hence, if new tasks keep arriving dynamically, GA should be kept re-executed in an online fashion. Considering that the running cost of GA is much expensive (more than 2 min each time on a Xeon E5-2650 v4 server in our experiment while the running time of other algorithms is negligible), it is only suitable for SQS.

Simulations are performed via changing the weighting factor α and fixing $\beta = 1$. Under each setting (one pair of α and β), the simulation is run for 500 times. In each simulation, the initial state of task queue $\mathbf{Q}$ and the VT's initial position $x[0]$ are randomly chosen, but are kept identical for all the algorithms. Since there is no risk of task overflow in SQS, we set the penalty parameters ζ and k_q as 0 in training.

Fig. 5 shows the average cost for executing each task under different user preferences in SQS. We can see that AL, AO, and RD always have a high cost, because of their inflexible and naive behaviors. When α is small, which means that the scheduling objective focuses more on energy consumption, EG performs well. As α grows to be large, TG starts to outperform EG. In most cases, GA can be much better than the above methods. However, when α is very small, a great number of HO actions are needed. This may cause the searching space of GA to be extremely large and thus reduce the achievable performance. For instance, in our experiments, when $\alpha = 0.03$ the average cost of GA is larger than that of EG. Clearly, our DRLOSM can always attain the lowest cost, no matter if α is very small or very large. It outperforms GA, even when the agent does not have any prior knowledge regarding the environment dynamics.

In Fig. 6, we display the average number of retransmitted tasks caused by transmission failures due to handover events. Since AL and EG do not have transmission failures, they do not require retransmission. Their results in Fig. 6 are always 0. (But their overall costs are still high.) AO causes the most number of retransmissions since all tasks have to be offloaded. Randomly placing some tasks for LE (i.e., RD) does not fundamentally solve the problem. Even the GA algorithm faces a significant amount of task retransmissions and hence a waste of time and energy resources. Again, by learning the knowledge of the dynamic environment, our DRLOSM can properly choose its scheduling actions to avoid the probable transmission failures and hence seldom causes task retransmissions.

If we separate the overall cost and consider individually the task latency and energy consumption, the comparisons among different schemes are displayed in Figs. 7 and 8, respectively. As expected, EG always consumes the lowest level of energy, with the cost of the largest delay. TG uses the highest energy consumption to guarantee the fastest execution of tasks. DRLOSM and GA provide both relatively small delays and low energy usage compared with other methods. Even without the knowledge of environment dynamics, DRLOSM can be more adapt to user preference, and obtain a better overall performance than GA as shown earlier in Fig. 5. The

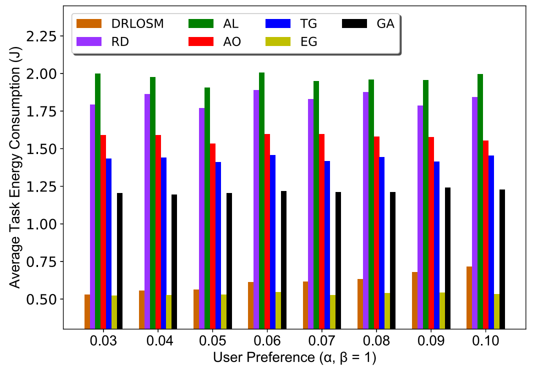


Fig. 8. Comparison of average task energy consumption in SQS.

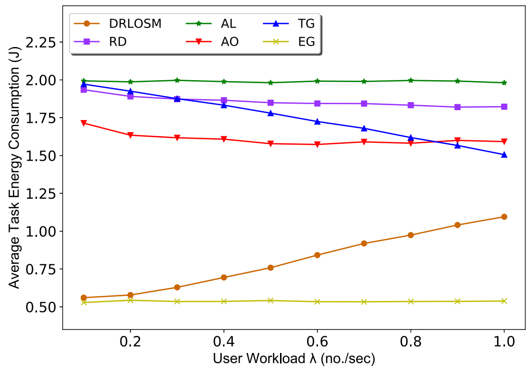


Fig. 10. Average task energy consumption versus workload in DQS.

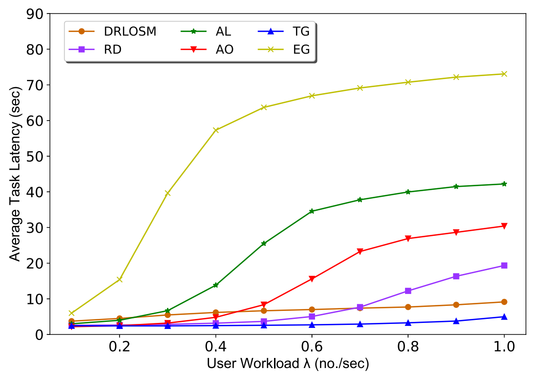


Fig. 9. Average task latency versus workload in DQS.

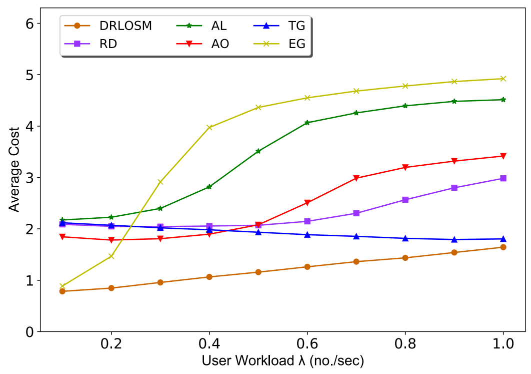


Fig. 11. Average cost versus workload in DQS.

vast searching space of the considered problem often makes it difficult for GA to find sufficiently good solutions.

C. Performance in the Dynamic Queue Scenario

In this section, we consider the more general situation in which the system workload $\lambda > 0$. In such a dynamic queue scenario (DQS), new tasks keep arriving in the VT's task queue and a good offloading scheduling solution should also take them into consideration. As mentioned in the previous section, applying the one-shot GA would require extremely high computation complexity. Hence, we compare our DRLOSM with only the remaining five baseline algorithms. We fix $\alpha = 0.06$ and $\beta = 1$. For each algorithm, the simulations are conducted for 200 rounds, each of which lasts 150 s. Within each simulation round, the initial state of different algorithms are set to be identical. We adjust the value λ to change from 0.1 (nearly no workload) to 1.0 (huge workload under which overflow occurs in all six algorithms) and investigate the impact of workload on system performance.

Fig. 9 shows the average task latency comparison in DQS. As λ increases, the average task latency of all the algorithms grows. For EG, AL, AO, and RD, there is a sudden rise of the curve slope at certain values of λ. These methods do not adapt their scheduling behaviors according to the workload. When λ is sufficiently large, the speed of task execution starts to lag behind the task arrival rate, which causes tasks to stack in the queue. When λ further increases, the task latency curves become flat because the task queues become fully occupied, and new tasks have to be discarded due to overflow. As expected, among the algorithms, TG has the lowest average task latency, and EG has the highest. DRLOSM always achieves a relatively small latency. The performance curve is smooth, which implies that it can adjust its execution strategy according to the workload.

The average task energy consumption comparison is shown in Fig. 10. The performance curves of EG, AL, AO, and RD are almost irrelevant to λ, which reflects the fact they do not adjust their behaviors to handle different workloads. The energy consumption of TG decreases as λ raises. This is because the proportion of retransmitted tasks decreases when more tasks are executed. The algorithm is efficient for very high-workload regimes. DRLOSM has more energy consumption as the workload is larger, because it is able to change its offloading scheduling policy, by scheduling more tasks with higher energy usage for avoiding a rapidly increased queue, to maintain a relatively small overall cost. This is the reason behind its good task latency performance in Fig. 9.

Fig. 11 illustrates the comparison of overall costs. When the workload is very light, EG achieves good performance. On the other hand, for a very heavy workload, TG outperforms other baseline algorithms. Our DRLOSM is strictly better than both EG and TG, for all possible values of λ. Finally, Fig. 12 displays the average number of retransmitted tasks due to communication failures. We can observe that DRLOSM properly handle the handover issue, even it does not have knowledge regarding the dynamics of the wireless environment. Only under high workload, the chances of DRLOSM's transmission failure rise. This is because it tries

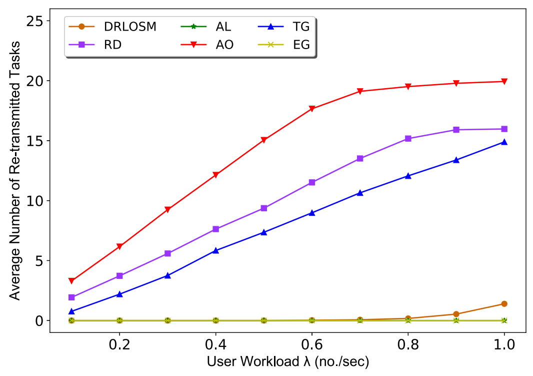


Fig. 12. Average number of retransmitted tasks versus workload in DQS.

TABLE VI
OFFLOADING STRATEGY OF CDS IN DIFFERENT SITUATIONS

Queue Length	Distance to RSU	Description	Strategy
$q[t] > \mathrm{LQ}$		too many tasks in queue	TG
$q[t] < \mathrm{SQ}$		system workload is low	EG
$\mathrm{SQ} < q[t] < \mathrm{LQ}$	$d > \mathrm{LD}$	offloading is not economical	AL
$\mathrm{SQ} < q[t] < \mathrm{LQ}$	$d < \mathrm{SD}$	offloading is economical	AO
$\mathrm{SQ} < q[t] < \mathrm{LQ}$	$\mathrm{LD} < d < \mathrm{SD}$	no dominant strategy	RD

to offload more tasks to adapt to the system workload even in bad wireless conditions. The advantages of the proposed solution are thus clearly exhibited.

D. Comparison With the Combinative Strategy

In this section, we consider applying a combined decision strategy (CDS) that can dynamically switch among the five intuitive algorithms, i.e., TG, EG, AL, AO, and RD, to achieve higher performance than each of them. The switching rule is determined based on the length of the task queue $q[t]$ and the distance between the VT and the serving RSU d, and four heuristic thresholds are adopted, respectively, termed long queue (LQ), short queue (SQ), long distance (LD), and short distance (SD). Table VI summarizes the offloading strategy of the CDS. Intuitively, if $q[t]$ is very large, it is better to adopt TG to empty the long task queue. When $q[t]$ is small, EG can be applied to minimize energy consumption without the necessity of concerning the impact of the system workload. If SQ $< q[t] <$ LQ, the CDS further considers the value of d. If the VT is sufficiently close to the serving RSU, AO is, in general, the best solution. But if d is sufficiently large, offloading may face failure and thus AL is chosen. Finally, when SQ $< q[t] <$ LQ and LD $< d <$ SD both occur, none of the above four strategies would dominate others, and hence RD is selected.

It is difficult to determine the optimal values of the four thresholds. We consider 24 different combinations of them (as shown in Table VII), which results in 24 ways to employ CDS. They are all evaluated in both SQS and DQS. The experiment results are shown in Figs. 13 and 14.

From the figures, we can see that the CDS algorithms have diverse performance, in different environmental conditions. Some of them can perform close to the DRLOSM (e.g., CDS 3 and CDS 6 when $\alpha = 0.03$ in SQS). In extreme situations (e.g., with a huge workload, or very imbalanced time and energy preference), they may even be slightly better (e.g., CDS 19 and CDS 24 in DQS when $\lambda = 1.0$). The main reason behind such observations is that in these situations, carrying out a simple greedy algorithm is already sufficiently good, but DRL may not be able to converge to the exact global optima.

TABLE VII
24 CDS ALGORITHMS WITH DIFFERENT THRESHOLDS

CDS	LQ	SQ	LD	SD	CDS	LQ	SQ	LD	SD
CDS 1	10	4	60	40	CDS 2	8	4	60	40
CDS 3	6	4	60	40	CDS 4	10	4	60	20
CDS 5	8	4	60	20	CDS 6	6	4	60	20
CDS 7	10	2	60	40	CDS 8	8	2	60	40
CDS 9	6	2	60	40	CDS 10	4	2	60	40
CDS 11	10	2	60	20	CDS 12	8	2	60	20
CDS 13	6	2	60	20	CDS 14	4	2	60	20
CDS 15	10	0	60	40	CDS 16	8	0	60	40
CDS 17	6	0	60	40	CDS 18	4	0	60	40
CDS 19	2	0	60	40	CDS 20	10	0	60	20
CDS 21	8	0	60	20	CDS 22	6	0	60	20
CDS 23	4	0	60	20	CDS 24	2	0	60	20

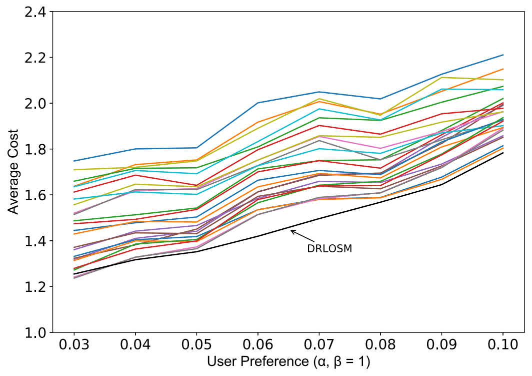


Fig. 13. Comparison of average cost in SQS.

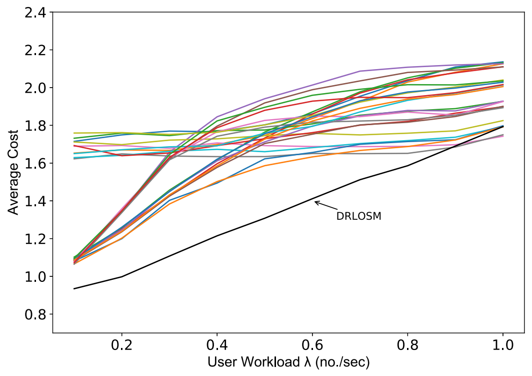


Fig. 14. Comparison of average cost in DQS.

However, to determine when and which the intuitive strategy should be adopted in different environment relies on expert knowledge (e.g., to determine suitable thresholds and the operating strategy in each condition) and demands a lot of effort to fine-tune the decision rule (e.g., finding the best threshold values in a different environment). In practice, the operation environment and expectation toward performance can be affected by a number of factors, such as user preference, system workload, fading characteristics, CPU frequency of

LPU and MEC server, and the distance between RSUs. Any change in such factors would require the combinative strategy to be redesigned and reoptimized. In such a complicated and dynamic environment, the proposed DRLOSM can address these issues by interacting directly with the environment and learning the optimal offloading scheduling policy. The advantages of model-free DRL-based algorithms are obvious.

VII. Conclusion

We have investigated the computation offloading scheduling problem in a typical VEC scenario, which is hard to solve using conventional methods because of the highly dynamic environment and the enormous state space. A novel DRL-based method has been proposed in this article to address these issues. It is designed based on the state-of-the-art PPO algorithm. A parameter-shared DNN architecture, which is enhanced by a CNN, is utilized to approximate both the policy and value function. A series of methods have been considered to deal with the large state and action spaces and improve training efficiency. Extensive simulation experiments have been conducted to demonstrate that our proposed method can efficiently learn the optimal offloading scheduling policy without requiring any prior knowledge of the environment dynamics, and it significantly outperforms a number of known baseline algorithms in terms of the long-term cost.

In this article, a fixed amount of the MEC computation resource and V2I transmission bandwidth is assumed to be reserved for each VT. In more general conditions, the resource limitation on each RSU should be considered. VTs would compete or cooperate with each other to share the limited resources. The offloading scheduling problem, in this case, may be formulated using a multiagent partially observable MDP (multiagent POMDP) [11]. In addition, one may also consider the scenarios in which each task has an individual priority demand (regarding, e.g., latency) and the VT's computing architecture is formed by multiple heterogeneous subsystems with diverse characteristics. Applying DRL to solve the offloading scheduling problem in these systems may demand new approaches to formulate the optimization problem, define the equivalent MDP, and design the DNN architecture and RL training method. They are treated as meaningful future research directions.

References

[1] J. Feng, Z. Liu, C. Wu, and Y. Ji, "Mobile edge computing for the Internet of Vehicles: Offloading framework and job scheduling," *IEEE Veh. Technol. Mag.*, vol. 14, no. 1, pp. 28–36, Mar. 2019.

[2] K. Zhang, Y. Mao, S. Leng, Y. He, and Y. Zhang, "Mobile-edge computing for vehicular networks: A promising network paradigm with predictive off-loading," *IEEE Veh. Technol. Mag.*, vol. 12, no. 2, pp. 36–44, Jun. 2017.

[3] Y. Dai, D. Xu, S. Maharjan, and Y. Zhang, "Joint load balancing and offloading in vehicular edge computing and networks," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4377–4387, Jun. 2019.

[4] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[6] X. Lin, Y. Wang, Q. Xie, and M. Pedram, "Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment," *IEEE Trans. Services Comput.*, vol. 8, no. 2, pp. 175–186, Mar./Apr. 2015.

[7] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Trans. Commun.*, vol. 64, no. 10, pp. 4268–4282, Oct. 2016.

[8] Y. Zhang, D. Niyato, and P. Wang, "Offloading in mobile cloudlet systems with intermittent connectivity," *IEEE Trans. Mobile Comput.*, vol. 14, no. 12, pp. 2516–2529, Dec. 2015.

[9] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, 2016, pp. 1451–1455.

[10] H. Ko, J. Lee, and S. Pack, "Spatial and temporal computation offloading decision algorithm in edge cloud-enabled heterogeneous networks," *IEEE Access*, vol. 6, pp. 18920–18932, 2018.

[11] Y. Li, "Deep reinforcement learning: An overview," Jan. 2017. [Online]. Available: arXiv:1701.07274.

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2011.

[13] J. Wang, J. Hu, G. Min, W. Zhan, Q. Ni, and N. Georgalas, "Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning," *IEEE Commun. Mag.*, vol. 57, no. 5, pp. 64–69, May 2019.

[14] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for MEC," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, 2018, pp. 1–6.

[15] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, "Learning-based computation offloading for IoT devices with energy harvesting," *IEEE Trans. Veh. Technol.*, vol. 68, no. 2, pp. 1930–1941, Feb. 2019.

[16] 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.

[17] W. Zhan, C. Luo, J. Wang, G. Min, and H. Duan, "Deep reinforcement learning-based computation offloading in vehicular edge computing," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, 2019, pp. 1–6.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," Jul. 2017. [Online]. Available: arXiv:1707.06347.

[19] A. R. Khan, M. Othman, S. A. Madani, and S. U. Khan, "A survey of mobile cloud computing application models," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 393–413, 1st Quart., 2013.

[20] T. Q. Dinh, J. Tang, Q. D. La, and T. Q. S. Quek, "Offloading in mobile edge computing: Task allocation and computational frequency scaling," *IEEE Trans. Commun.*, vol. 65, no. 8, pp. 3571–3584, Aug. 2017.

[21] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 4, pp. 974–983, Apr. 2015.

[22] X. Lyu, H. Tian, C. Sengul, and P. Zhang, "Multiuser joint task offloading and resource optimization in proximate clouds," *IEEE Trans. Veh. Technol.*, vol. 66, no. 4, pp. 3435–3447, Apr. 2017.

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

[24] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double *Q*-learning," in *Proc. AAAI 30th Conf. Artif. Intell. (AAAI)*, 2016, pp. 2094–2100.

[25] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3, pp. 229–256, 1992.

[26] T. Degris, M. White, and R. S. Sutton, "Off-policy actor–critic," in *Proc. 29th Int. Conf. Mach. Learn. (ICML)*, 2012, pp. 179–186.

[27] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–14.

[28] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," Jul. 2016. [Online]. Available: arXiv:1607.06450.

[29] G. Dulac-Arnold *et al.*, "Deep reinforcement learning in large discrete action spaces," Dec. 2015. [Online]. Available: arXiv:1512.07679.

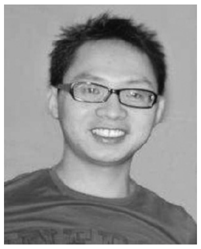
[30] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX 12th Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.

[32] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Mach. Learn. Res.*, vol. 13, no. 70, pp. 2171–2175, Jul. 2012.

Wenhan Zhan received the B.E. and M.Sc. degrees from the University of Electronic Science and Technology of China (UESTC), Chengdu, China, in 2010 and 2013, respectively, where he is currently pursuing the Ph.D. degree.

Since 2013, he has been an Experimentalist with UESTC. From 2018 to 2019, he worked as a Visiting Scholar with the Department of Computer Science, University of Exeter, Exeter, U.K. His research interests mainly lie in distributed system, cloud computing, edge computing, and artificial intelligence.



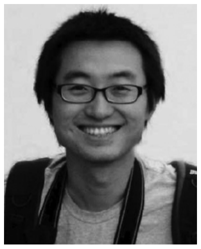
Chunbo Luo (Member, IEEE) received the Ph.D. degree in high-performance cooperative wireless networks from the University of Reading, Reading, U.K.

His research has been supported by NSFC, Royal Society, EU H2020, and industries. His research interest focuses on developing model-based and machine learning algorithms to solve networking and engineering problems, such as wireless networks, with a particular focus on unmanned aerial vehicles.

Dr. Luo is a Fellow of the Higher Education Academy.

Jin Wang received the B.Eng. and M.Eng. degrees in computer system architecture from the University of Electronic Science and Technology of China, Chengdu, China, in 2014 and 2017, respectively. He is currently pursuing the Ph.D. degree in computer science with the University of Exeter, Exeter, U.K.

His research interests include deep reinforcement learning, applied machine learning, cloud and edge computing, and computer system optimization.



Chao Wang (Member, IEEE) received the B.E. degree from the University of Science and Technology of China, Hefei, China, in 2003, and the M.Sc. and Ph.D. degrees from the University of Edinburgh, Edinburgh, U.K., in 2005 and 2009, respectively.

From 2009 to 2012, he was a Postdoctoral Research Associate with the KTH-Royal Institute of Technology, Stockholm, Sweden. Since 2013, he has been with Tongji University, Shanghai, China, where he is an Associate Professor. He is currently a Marie Sklodowska-Curie Individual Fellowship with the University of Exeter, Exeter, U.K. His research interests include information theory and signal processing for wireless communication networks, as well as data-driven research and applications for smart city and intelligent transportation systems.

Geyong Min received the B.Sc. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 1995, and the Ph.D. degree in computing science from the University of Glasgow, Glasgow, U.K., in 2003.

He is a Professor of high-performance computing and networking with the Department of Computer Science, College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter, U.K. His research interests include computer networks, wireless communications, parallel and distributed computing, ubiquitous computing, multimedia systems, and modeling and performance engineering.

Hancong Duan received the B.Sc. degree from Southwest Jiaotong University, Chengdu, China, in 1995, and the M.E. and Ph.D. degrees from the University of Electronic Science and Technology of China, Chengdu, in 2005 and 2007, respectively.

He is currently a Professor of computer science with the University of Electronic Science and Technology of China. His research interests include large-scale P2P content delivery network, distributed storage, cloud computing, and artificial intelligence.

Qingxin Zhu received the Ph.D. degree in cybernetics from the University of Ottawa, Ottawa, ON, Canada, in 1993.

From 1993 to 1996, he was a Postdoctoral Researcher with the Department of Electronic Engineering, University of Ottawa and the School of Computer Science, Carleton University, Ottawa. He was a Senior Researcher with Nortel, Ottawa, and Omnimark, Ottawa, from 1996 to 1997. Since 1998, he has been with the University of Electronic Science and Technology of China, Chengdu, China, as a Professor and the Director of Operations Research Office. From 2002 to 2003, he worked as a Senior Visiting Scholar with the Computer Department, Concordia University, Montreal, QC, Canada. His research interests include bioinformatics, operational research, and optimization.