Dependent Task Offloading for Edge Computing based on Deep Reinforcement Learning

Jin Wang, Jia Hu, Geyong Min, Wenhan Zhan, Albert Y. Zomaya, *Fellow, IEEE*, and Nektarios Georgalas

Abstract—Edge computing is an emerging promising computing paradigm that brings computation and storage resources to the network edge, hence significantly reducing the service latency and network traffic. In edge computing, many applications are composed of dependent tasks where the outputs of some are the inputs of others. How to offload these tasks to the network edge is a vital and challenging problem which aims to determine the placement of each running task in order to maximize the Quality-of-Service (QoS). Most of the existing studies either design heuristic algorithms that lack strong adaptivity or learning-based methods but without considering the intrinsic task dependency. Different from the existing work, we propose an intelligent task offloading scheme leveraging off-policy reinforcement learning empowered by a Sequence-to-Sequence (S2S) neural network, where the dependent tasks are represented by a Directed Acyclic Graph (DAG). To improve the training efficiency, we combine a specific off-policy policy gradient algorithm with a clipped surrogate objective. We then conduct extensive simulation experiments using heterogeneous applications modelled by synthetic DAGs. The results demonstrate that: 1) our method converges fast and steadily in training; 2) it outperforms the existing methods and approximates the optimal solution in latency and energy consumption under various scenarios.

Index Terms—Multi-access edge computing, task offloading, deep reinforcement learning, sequence to sequence neural networks.

1 Introduction

The rapid proliferation of smart user equipment (UE) like smart home appliances and wearable devices has spurred numerous innovative applications, such as augmented reality (AR), virtual reality (VR), face recognition, and digital healthcare. The increasing diversity and complexity of these applications demand intensive computing resources. Although new generations of UE possess intensive computing power, they still suffer from high energy and time costs when processing the resource-intensive applications. One solution is to run these applications on the cloud. However, it imposes huge traffic on the backhaul of mobile networks and hinders real-time applications due to the long distance between UE and cloud. To address these issues, Multi-access Edge Computing (MEC) [1] was proposed to shift certain computing and storage resources from the cloud to the MEC hosts at the network edge (e.g., base stations, access points, and edge routers) that is close to users. Hence, MEC can effectively alleviate network congestion and reduce the service latency for mobile users.

- *Jin Wang, Jia Hu, and Geyong Min are with the Department of Computer Science, University of Exeter, United Kingdom. E-mail: {jw855, j.hu, g.min}@exeter.ac.uk*
- *Wenhan Zhan is with the School of Computer Science and Engeeneering, University of Electronic Science and Technology of China, China. E-mail: zhanwenhan@uestc.edu.cn*
- *Albert Y. Zomaya is with the School of Information Technologies, The University of Sydney, Australia. E-mail: albert.zomaya@sydney.edu.au*
- *Nektarios Georgalas is with Applied Research Department, British Telecom, United Kingdom. E-mail: nektarios.georgalas@bt.com*
- *Code is available at https://github.com/linkpark/RLTaskOffloading (Corresponding authors: Jia Hu and Geyong Min.)*

One key technology in MEC is task offloading, which enables UE to offload computation-intensive tasks of user applications to MEC hosts, thus reducing latency and energy consumption at UE during the processing of the application. In task offloading, we must decide whether a task should be offloaded to an MEC host, depending on the task profile (i.e., the required CPU cycles and input/output data size) and the MEC environment. Many existing studies [2, 3, 4, 5, 6] developed heuristic or approximation algorithms, since the above offloading problem is NP-hard. However, they rely heavily on expert knowledge or accurate analytical models. As a consequence, considerable human efforts and expertise are required to tune these heuristics or analytical models to adapt to new scenarios, which is time-consuming and even unrealistic due to the increasing complexity of applications and system architecture of MEC.

Deep Reinforcement Learning (DRL), which combines Reinforcement Learning (RL) with Deep Neural Networks (DNN), is a promising approach to achieve flexible and adaptive task offloading without expert knowledge. DRL learns an effective policy (i.e., a mapping from environment states to actions) through interacting with the environment so as to maximize numerical rewards. With the help of the powerful representation ability of DNNs, DRL can effectively solve complex decision-making problems with large and high-dimensional state/action spaces. Recent breakthroughs in DRL have led to many successful applications in a wide range of areas including gaming [7], robotics [8], networking [9], etc. DRL has been adopted to handle task offloading problems in MEC [9, 10, 11, 12]. However, these methods assume the offloading tasks are independent without considering the inherent dependency among tasks of real-world applications. In practice, many applications are composed of dependent tasks where the outputs of some tasks are the inputs of others. Ignoring the task dependency

when making task offloading decisions will severely affect the QoS of applications and waste the edge resources.

To fill this gap, we propose a new DRL-based Task Offloading (DRLTO) scheme leveraging off-policy RL empowered by a Sequence-to-Sequence (S2S) neural network. The DRLTO is able to reduce the latency of running applications and the energy consumption at UE. In DRLTO, the task offloading problem is modelled as a Markov Decision Process (MDP). Applications are represented by Directed Acyclic Graphs (DAG), where vertices and edges denote tasks and their dependency, respectively. To effectively extract the key features of task dependency, an S2S neural network is applied to represent the policy and value function of the MDP. Specifically, the input of the S2S neural network is the DAG represented by a sequence of embedding vectors, while the output is the offloading plan for the DAG. To improve the training efficiency, we combine an off-policy DRL algorithm that includes a clipped surrogate objective and an entropy bonus to stabilize the training, provide better sample efficiency, and alleviate the issue of getting stuck in local optima. The DRLTO learns to make efficient offloading decisions through directly interacting with the environment and only requires minimal expert knowledge. The major contributions of this paper are summarized as follows:

- We develop an original DRL-based task offloading scheme, which leverages off-policy reinforcement learning with an S2S neural network to capture the intrinsic task dependency of applications. An MDP is designed to accurately model the dependent task offloading problem with well-designed state space, action space, and reward function.
- We design a new embedding method that encodes vertices of the DAG representing an application to a sequence of embeddings including both the task profiles and dependency information. This method can convert the raw DAG as the input of the S2S neural network without information loss.
- We combine an S2S neural network with an attention mechanism to capture the long-term dependency of the input tasks. This S2S neural network can effectively approximate the policy and value function of the MDP model for the dependent task offloading problem. To effectively train the S2S neural network, we apply an off-policy DRL algorithm with a clipped surrogate objective function and an entropy bonus. This algorithm has a strong exploration ability and thus can prevent the training from getting stuck in the local optima.
- Extensive simulation experiments were conducted using the synthetic DAGs, covering a wide range of topologies, task numbers, and data rates that correspond to the characteristics of real-world applications. The performance results show that our method outperforms advanced heuristic baselines and can obtain near-optimal results under dynamic MEC scenarios.

The rest of this paper is organized as follows. The problem formulation, energy model, and optimization target are presented in Section 2. The details of the DRL-based offloading scheme are described in Section 3. Simulation results are presented and discussed in Section 4. The related work is reviewed in Section 5. Finally, Section 6 concludes the paper.

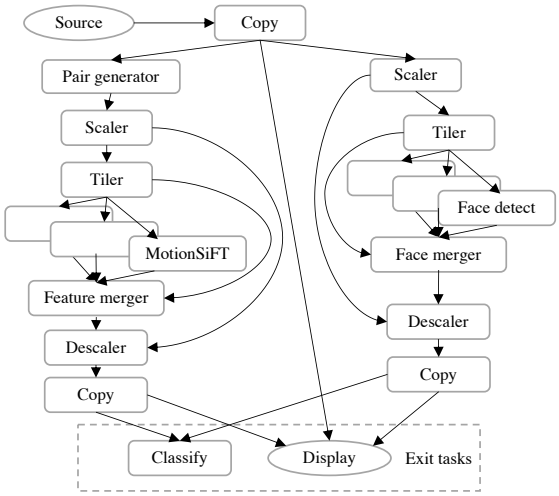

Fig. 1. An example of DAG for the gesture recognition application.

2 Problem Formulation: Task Offloading

The section presents the formulation of the task offloading problem. First, we give the details of the task offloading process in MEC. Next, the energy model and the optimization target used for task offloading are described.

In practice, many applications consist of multiple tasks with general dependency, which should be considered when making decisions on task offloading. For example, as shown in Fig. 1, a gesture recognition application consists of multiple dependent tasks [13]. Formally, let $G = (\mathcal{T}, \mathcal{E})$ denote a DAG, where a vertex $t_i \in \mathcal{T}$ and a directed edge $e(t_i, t_j) \in \mathcal{E}$ represents a task t_i and the dependency between t_i and t_j, respectively. A task can start to run only when all of its predecessors are finished. The exit tasks are those without subsequent tasks.

We consider a block fading channel, where the fading coefficient remains unchanged (i.e., the transmission rate is fixed) during the task offloading process. In general, the MEC host runs multiple virtual machines (VMs) to process the offloaded tasks. In this work, we consider that each UE is associated with a dedicated VM providing provide computing, communications and storage resources to the UE as specified in many existing studies [14, 15]. Any task in the DAG has two scheduling choices: offloaded to the MEC host or run locally on the UE. If the task t_i is offloaded, it has three phases of execution: 1) *sending phase:* the UE sends t_i to the MEC host through a wireless channel. 2) *executing phase:* the MEC host executes the received task t_i. 3) *receiving phase:* the MEC host returns the results to the UE. On the other hand, if t_i is locally executed, there is no data transmission between the UE and MEC host. The local processor of the UE directly processes the task when it is ready. For all tasks in a DAG, let $A_{1:n} = [a_1, a_2, \ldots, a_i, \ldots, a_n]$ denote an offloading plan, where n represents the total task number and a_i represents the offloading decision of t_i. Specially, $a_i = 1$ denotes that t_i is offloaded to the MEC host, otherwise, $a_i = 0$ means that t_i is scheduled to the local processor.

Let $\mathrm{T}_i^{\mathrm{ul}}$, $\mathrm{T}_i^{\mathrm{s}}$, and $\mathrm{T}_i^{\mathrm{dl}}$ denote the latency of sending, executing and receiving phase, respectively. Let $\mathrm{T}_i^{\mathrm{l}}$ denote the local execution latency for t_i. Besides, f_{l} and f_{s} represent the CPU clock speed of the UE and the VM in the MEC Host, respectively. R_{ul} denotes the uplink transmission rate while R_{dl} denotes the downlink transmission rate. Let $\mathrm{data}_i^{\mathrm{s}}$ and $\mathrm{data}_i^{\mathrm{r}}$ represent the data size of the task t_i that is offloaded to the MEC host and the result received, respectively. We denote the required CPU cycles for executing t_i as C_i. Therefore, all the above-defined latencies can be calculated by using the following equations:

$$\begin{aligned} \mathrm{T}_i^{\mathrm{ul}} &= \mathrm{data}_i^{\mathrm{s}}/R_{\mathrm{ul}}, \qquad \mathrm{T}_i^{\mathrm{s}} = C_i/f_{\mathrm{s}}, \\ \mathrm{T}_i^{\mathrm{dl}} &= \mathrm{data}_i^{\mathrm{r}}/R_{\mathrm{dl}}, \qquad \mathrm{T}_i^{\mathrm{l}} = C_i/f_{\mathrm{l}}. \end{aligned} \tag{1}$$

We also assume the available time of the uplink wireless channel, the MEC host, the downlink wireless channel, and the local processor, as $\mathcal{M}_i^{\mathrm{ul}}$, $\mathcal{M}_i^{\mathrm{s}}$, $\mathcal{M}_i^{\mathrm{dl}}$, $\mathcal{M}_i^{\mathrm{l}}$, respectively. Given a scheduling plan $A_{1:n}$, the available time of the resource depends on the finish time (FT) of the task scheduled immediately before t_i on that resource. If the task scheduled immediately before t_i does not utilize the resource, the FT on that resource is set as 0. We next analyse the local executing and remote offloading process in detail.

Local Executing: In this case, t_i is scheduled to the local processor. Note that t_i can only start execution until all its immediate predecessors are finished and the local processor is available to run. Besides, the immediate predecessors of t_i can be either run on the MEC host or the local processor. Therefore, the FT of task t_i on the local processor, $\mathrm{FT}_i^{\mathrm{l}}$, can be defined as

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{l}} &= \max\left\{\mathcal{M}_i^{\mathrm{l}}, \max_{j\in\mathrm{pre}(t_i)}\left\{\mathrm{FT}_j^{\mathrm{l}}, \mathrm{FT}_j^{\mathrm{dl}}\right\}\right\} + \mathrm{T}_i^{\mathrm{l}}, \\ \mathcal{M}_i^{\mathrm{l}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{l}}, \mathrm{FT}_{i-1}^{\mathrm{l}}\right\}. \end{aligned} \tag{2}$$

where $\mathrm{pre}(t_i)$ denotes the set of immediate predecessors of t_i. $\mathrm{FT}_j^{\mathrm{dl}}$ denotes the FT of transmitting a task t_j over the wireless downlink channel.

Remote Offloading: In this case, t_i is offloaded to the MEC host. As mentioned above, the offloading process for t_i includes three phases. In the sending phase, let $\mathrm{FT}_j^{\mathrm{ul}}$ denote the FT of transmitting the task t_j over the wireless uplink channel, where t_j is an immediate predecessor of t_i. If t_j is locally scheduled, t_i can only start its sending phase after t_j has finished local execution. Otherwise, if t_j is offloaded, t_i can only start sending after t_j has completed its sending phase. Therefore, the FT of task t_i on the wireless uplink channel can be calculated as:

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{ul}} &= \max\left\{\mathcal{M}_i^{\mathrm{ul}}, \max_{j\in\mathrm{pred}(t_i)}\left\{\mathrm{FT}_j^{\mathrm{l}}, \mathrm{FT}_j^{\mathrm{dl}}\right\}\right\} + \mathrm{T}_i^{\mathrm{ul}}, \\ \mathcal{M}_i^{\mathrm{ul}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{ul}}, \mathrm{FT}_{i-1}^{\mathrm{ul}}\right\}. \end{aligned} \tag{3}$$

In the executing phase, three conditions must be met before t_i can start running on the MEC host. First, t_i should finish its sending phase. Second, all predecessors of t_i should finish executing. Third, the MEC host is available to run the task. Therefore, the FT of t_i on the MEC host, $\mathrm{FT}_i^{\mathrm{s}}$, can be calculated as

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{s}} &= \max\left\{\mathcal{M}_i^{\mathrm{s}}, \max\left\{\mathrm{FT}_i^{\mathrm{ul}}, \max_{j\in\mathrm{pred}(t_i)}\mathrm{FT}_j^{\mathrm{s}}\right\}\right\} + \mathrm{T}_i^{\mathrm{s}}, \\ \mathcal{M}_i^{\mathrm{s}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{s}}, \mathrm{FT}_{i-1}^{\mathrm{s}}\right\}. \end{aligned} \tag{4}$$

Similarly, we obtain the FT of task t_i on the downlink wireless channel, $\mathrm{FT}_i^{\mathrm{dl}}$ as

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{dl}} &= \max\left\{\mathcal{M}_i^{\mathrm{dl}}, \mathrm{FT}_i^{\mathrm{s}}\right\} + \mathrm{T}_i^{\mathrm{dl}}, \\ \mathcal{M}_i^{\mathrm{dl}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{dl}}, \mathrm{FT}_{i-1}^{\mathrm{dl}}\right\}. \end{aligned} \tag{5}$$

Energy consumption of the UE is another important factor that we should consider in MEC. In general, the energy consumption of an UE mainly consists of computation and transmission cost. Executing task locally or offloading task to the MEC host result in different energy costs.

When the task is locally executed, no transmission is needed, thus the energy consumption is mainly contributed by the computation process, which is defined as

$$\mathrm{E}_i^{\mathrm{l}} = \rho f_{\mathrm{l}}^{\zeta}\mathrm{T}_i^{\mathrm{l}}, \tag{6}$$

where $\rho f_{\mathrm{l}}^{\zeta}$ represents the computational power of the mobile device. ρ is a power coefficient while ζ is a constant (usually close to 3) [3].

When the task is offloaded, there is no computation process on the local device, thus the energy consumption is mainly contributed by wireless transmission, which is defined as

$$\mathrm{E}_i^{\mathrm{s}} = P^{\mathrm{Tx}}\mathrm{T}_i^{\mathrm{ul}} + P^{\mathrm{Rx}}\mathrm{T}_i^{\mathrm{dl}}, \tag{7}$$

where P^{Tx} and P^{Rx} are the sending and receiving power, respectively.

Based on the above definitions, giving the offloading plan, $A_{1:n}$, of a DAG, the latency and energy consumption can be calculated as

$$\mathrm{T}_{A_{1:n}}^{c} = \max_{t_i\in\mathcal{K}}\left\{\max\{\mathrm{FT}_i^{\mathrm{l}}, \mathrm{FT}_i^{\mathrm{dl}}\}\right\}, \tag{8}$$

$$\mathrm{E}_{A_{1:n}}^{c} = \sum_{t_i\in\mathcal{T}, a_i=1}\mathrm{E}_i^{\mathrm{s}} + \sum_{t_j\in\mathcal{T}, a_j=0}\mathrm{E}_j^{\mathrm{l}}, \tag{9}$$

where $\mathcal{K}$ denotes the set of exit tasks.

QoS can be used to measure how good an offloading plan is, considering both latency and energy consumption. As in [16, 17], we use a similar definition of QoS as the optimization objective, which is a weighted sum of the normalized differences in latency and energy consumption between the offloading plan and local execution:

$$J_{A_{1:n}} = \lambda_t\frac{\mathrm{T}_l^c - \mathrm{T}_{A_{1:n}}^c}{\mathrm{T}_l^c} + \lambda_e\frac{\mathrm{E}_l^c - \mathrm{E}_{A_{1:n}}^c}{\mathrm{E}_l^c}, \tag{10}$$

where T_l^c and E_l^c are the latency and energy consumption of executing all tasks locally on the UE. λ_t and $\lambda_e \in [0, 1]$ are scalar weights. Eq. (10) represents a weighted sum approach of a multi-objective optimization problem. The weighted-sum approach is extensively used since it is generally effective and easy to implement. Besides, the weights reflect the relative importance of energy and latency, which can be set based on the user's preference. For general DAGs, the scheduling problem is NP-hard [18]. Therefore, it is extremely hard to find the optimal offloading plan with reasonable time complexity.

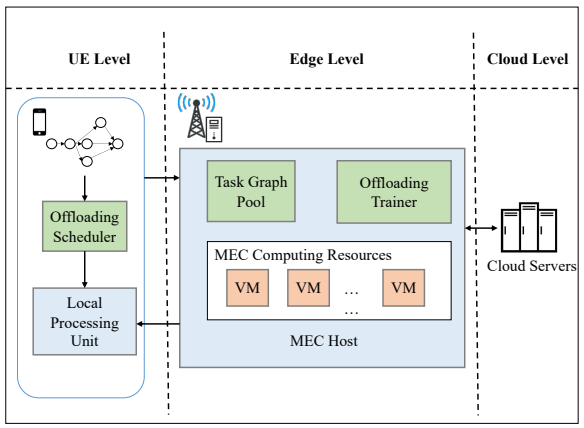

Fig. 2. The proposed DRL-based task offloading scheme.

3 The DRLTO Scheme

This section presents the proposed DRLTO in detail. We first present the necessary background of DRL and S2S neural network. Next, the overall design of the DRLTO architecture is described. Finally, the offloading model, the S2S neural network, and the training algorithm are presented in detail.

3.1 Background

Deep Reinforcement Learning: In general, the targeted problem of RL is formulated as an MDP that is defined by a 6-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{P}_0, \mathcal{R}, \gamma)$, where $\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, and $\mathcal{P}_0$ denote the state space, action space, state transition probability matrix, and initial state distribution, respectively. $\mathcal{R}$ means the reward function and $\gamma \in (0, 1)$ represents the discount factor. Denote $\pi(a|s)$ as the policy that specifies the probability of taking action a given state s. The trajectories sampled under the policy π are denoted as $\tau^{\pi} = (s_0, a_0, r_0, s_1, a_1, r_1, ...)$, where $s_0 \sim \mathcal{P}_0$, $a_t \sim \pi(\cdot|s_{t-1})$ and r_t is the immediate reward at time step t. The state value function of a state s_t under a policy π, denoted as $v_\pi(s_t)$, is the expected return when starting in s_t and following π thereafter, which is defined as $v_\pi(s_t) = \mathbb{E}_\pi\left[\sum_{k=0} \gamma^k r_{t+k} \middle| s_t\right]$. Similarly, the state-action value function, denoted as $q_\pi(s_t, a_t)$, is the expected return starting from s_t, taking the action a_t, and thereafter following policy π, which can be formally denoted as $q_\pi(s_t, a_t) = \mathbb{E}_\pi\left[\sum_{k=0} \gamma^k r_{t+k} \middle| s_t, a_t\right]$. The objective of RL is to find a policy that maximizes the total return $G_0 = \sum_{s_0} \mathcal{P}_0(s_0) v_\pi(s_0)$.

In general, the current methods for DRL include value-based and policy-based methods. Value-based methods aim to get the optimal value function of every state in the environment: $v^*_\pi(s) = \max_a q_{\pi^*}(s, a)$, where π^* denotes the optimal policy. A typical valued-based method is deep Q-learning [19], which approximates the optimal value function by DNN (i.e., deep Q-network). However, deep Q-learning indirectly obtains a deterministic policy through updating the deep Q-network toward the one-step reward. A drawback of this one-step updating rule is that obtaining a reward only directly affects the value of the state-action pair that led to the reward while the values of other state-action pairs are indirectly affected by the updated Q-network. This can result in a slow learning process [20]. In contrast to value-based methods, policy-based DRL directly parameterizes the policy with a DNN and updates the policy network by performing gradient ascent on the total return [21] rather than the one-step return, thus it can achieve a faster convergence rate. However, when facing a combinatorial optimization problem such as task offloading in MEC, it has two main defects: 1) using Monte-Carlo estimation can bring high variance leading to slow learning. 2) conventional policy gradient methods are data inefficient and can get stuck in local optima [22]. To address the above issues, we apply proximal policy optimization (PPO) [23] that includes generalized advantage estimation (GAE) [24], off-policy clipped surrogate objective, and an entropy bonus to train the S2S neural network in this work. In particular, GAE can substantially reduce the variance by introducing an exponentially weighted estimator of the advantage function into the objective. The off-policy clipped surrogate objective can stabilize the training and improve the sample efficiency. Besides, adding the entropy bonus can further enhance the exploration [23], thus alleviating the problem of getting stuck in local minima.

Sequence-to-Sequence Neural Network: A typical S2S neural network consists of an encoder and a decoder, where the encoder and decoder are implemented by multi-layer RNNs. The input sequence is fed into the encoder and the final hidden state is output through forward propagation. The decoder initializes its hidden state via the encoder's output state and outputs a target sequence. The original S2S neural network [25] uses a fixed vector to encode the whole input sequence, which can lead to serious information loss for input sequences with long-term dependency. To cope with this problem, the attention mechanism [26] was proposed. In the attention mechanism, at each decoding step, a decoder output depends on its previous output, previous hidden state, and a context vector which is a weighted sum of the encoder output. Introducing the context vector gives the decoder a chance to "attend" important information from the source input sequence. Some work [27, 28] extends the S2S neural network to handle combinatorial optimisation problems including Travelling Salesman Problem (TSP) and Convex Hull. Inspired by the above work, we introduce the S2S neural network in our method, since task offloading in MEC is a typical combinatorial optimisation problem. Specifically, the S2S neural network is used to approximate both policy and value function of DRLTO, which can effectively extract representative features from DAGs.

3.2 The DRLTO Design

The DRLTO can be integrated into an emerging MEC platform defined by ETSI [1]. As shown in Fig. 2, the MEC system consists of three levels: UE level, edge level, and cloud level. UE level includes various user devices (such as smartphones, tablets, vehicles) and software applications. In edge level, each MEC host contains the computing, storage and network resources for processing applications on its virtual machines (VMs). In cloud level, cloud servers are hired to process computation-intensive and resource hungry jobs. The DRLTO scheme contains three major components: offloading scheduler, task graph pool, and offloading trainer. Each UE has an offloading scheduler, which makes

offloading decisions for user applications. Specifically, the trained S2S neural network is included in the offloading scheduler. The MEC host contains a task graph pool and an offloading trainer, which are used to gather DAGs from UE and conduct periodical training process, respectively.

The training for the S2S neural network is based on periodically gathered DAGs. At the off-peak time (e.g. midnight), the offloading trainer runs the training procedure. After training, the MEC host sends the parameters of the trained S2S neural network back to UE. The UE can then make the offloading decision via forward-propagation of the trained S2S neural network. The MEC host executes the offloaded tasks and returns the results to the UE. For those locally executed tasks, the local processing unit of the UE executes them when ready.

3.3 The Task Offloading Model

In order to adapt DRL to solve the task offloading problem, we model the problem as an MDP, where the state space, action space, and reward function of the MDP are defined as follows.

- **State Space:** When scheduling the task t_i, the state of the MEC system depends on the scheduling results of the previous tasks of t_i (i.e., the partial offloading plan). Hence, we define the state space as a combination of the DAG information (including DAG topology and task profiles) and the partial offloading plan. Formally, let G denote the encoded DAG and $A_{1:i}$ denote the offloading plan for the sequence of tasks from t_1 to t_i. The state space is thus denoted as

$$\mathcal{S} := \{s | s = (G, A_{1:i})\}. \tag{11}$$

  Specifically, G is comprised of a sequence of task embeddings. Each embedding consists of three elements: 1) a vector that includes an index of the task and the estimated task costs $\mathrm{T}_i^{\mathrm{l}}$, $\mathrm{T}_i^{\mathrm{ul}}$, $\mathrm{T}_i^{\mathrm{s}}$, and $\mathrm{T}_i^{\mathrm{dl}}$; 2) a vector of indices of immediate previous tasks; 3) a vector of indices of immediate subsequent tasks. The length of previous/subsequent task indices vector is limited to p ($p = 12$ in our experiments). We pad the vector with -1, in case the number of previous/subsequent tasks is less than p. The embedding vectors are then fed into the S2S neural network to obtain offloading plans.
- **Action Space:** A task could be either offloaded to an MEC host or run locally on the UE. Let $a_i = 1$ represent offloading to an MEC host and $a_i = 0$ represent local execution for task t_i. Therefore, the action space can be defined as $\mathcal{A} := \{1, 0\}$.
- **Reward Function:** The objective of our method is to maximize the QoS. In order to reach this objective, we define the reward function at each step as the estimated increment of the QoS:

$$\begin{aligned} R(s_i, a_i) &= \lambda_t \frac{\overline{\mathrm{T}}_l^c - \Delta \mathrm{T}_i}{\mathrm{T}_l^c} + \lambda_e \frac{\overline{\mathrm{E}}_l^c - \Delta \mathrm{E}_i}{\mathrm{E}_l^c}, \\ \Delta \mathrm{T}_i &= \mathrm{T}_{A_{1:i}}^c - \mathrm{T}_{A_{1:i-1}}^c, \\ \Delta \mathrm{E}_i &= \mathrm{E}_{A_{1:i}}^c - \mathrm{E}_{A_{1:i-1}}^c, \end{aligned} \tag{12}$$

  where $\overline{\mathrm{T}}_l^c$ and $\overline{\mathrm{E}}_l^c$ are the average latency and energy consumption for a task in the DAG, which are given by $\overline{\mathrm{T}}_l^c = \mathrm{T}_l^c / n$ and $\overline{\mathrm{E}}_l^c = \mathrm{E}_l^c / n$, respectively.

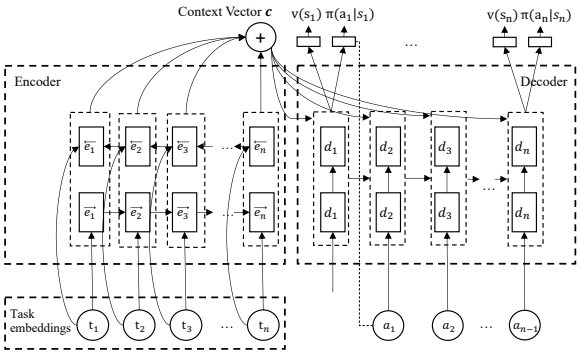

Fig. 3. Structure of the S2S neural network for DRLTO.

3.4 The S2S Neural Network for DRLTO

According to the definition of our MDP model, the offloading problem can be converted to an S2S prediction problem where the input is the sequence of task embeddings and the output is a scheduling plan for those tasks. The policy $\pi(a_i | G, A_{1:i-1})$ is denoted as the probability of selecting action a_i for task t_i under the state $s = (G, A_{1:i-1})$. Moreover, $\pi(A_{1:n} | G)$ is the probability of having the offloading plan $A_{1:n}$ given the graph G with n tasks. Applying the chain rules of probability, we have

$$\pi(A_{1:n} | G) = \prod_{i=1}^{n} \pi(a_i | G, A_{1:i-1}). \tag{13}$$

To effectively approximate the policy defined in Eq. (13), an S2S neural network is a good option. As shown in Fig. 3, we combine an S2S neural network with the attention mechanism to approximate both the policy and the value function of the DRLTO. Formally, denote the sequence of task embeddings as $\mathbf{t} = [t_1, t_2, ...t_n]$ and the function of encoder network as f_{enc}, then the hidden state of the encoder network can be obtained by:

$$e_i = f_{\mathrm{enc}}(e_{i-1}, t_i; \theta_{\mathrm{enc}}), \tag{14}$$

where e_i is the hidden state for encoding step i and θ_{enc} are the parameters of the encoder network. Let f_{dec} be the function of the decoder network. The hidden state of the decoder network d_j for decoding step j is calculated by:

$$d_j = f_{\mathrm{dec}}(d_{i-1}, a_{j-1}, c_j; \theta_{\mathrm{dec}}), \tag{15}$$

where θ_{dec} are the parameters of the decoder network. c_j is the context vector of the attention mechanism, which is defined by a weighted sum of the hidden states of the encoder network:

$$c_j = \sum_{i=1}^{n} \alpha_{ji} e_i. \tag{16}$$

The weight α_{ji} of each hidden state of the encoder network, e_i, is computed by

$$\alpha_{ji} = \frac{e^{f_{\mathrm{score}}(d_{j-1}, e_i)}}{\sum_{k=1}^{n} e^{f_{\mathrm{score}}(d_{j-1}, e_k)}} \tag{17}$$

where the score function, $f_{\mathrm{score}}(d_{j-1}, e_i)$, is used to measure how well the input of the encoder network at position i matches the output of the decoder network at position j. In

this paper, we define the score function as a trainable neural network as in the work [26].

The policy and value networks share most of the parameters (encoder and decoder) except for the top layer of the decoder. For the policy neural network, we add a fully connected layer on the output of the decoder, d_j , and use softmax function to convert the output into the distribution over actions, $\pi(a_j|s_j)$. For the value neural network, we add another fully connected layer on d_j and use the output to represent the state value, $v(s_j)$. The shared parameters in the S2S architecture are used to extract common features in DAGs, therefore training the policy neural network that can accelerate the training of the value neural network and vice versa.

The offloading decision for each task is made by the trained S2S neural network. Overall the steps for offloading process are follows:

Step 1: A topological sorting is conducted to sort tasks of the DAG according to the rank values of tasks by decreasing order, which is defined as

$$rank(t_i) = \begin{cases} \mathrm{T_i^o} & \text{if } t_i \in \mathcal{K}, \\ \max_{t_j \in \mathrm{succ(t_i)}}(rank(t_j)) + \mathrm{T_i^o} & \text{if } t_i \notin \mathcal{K}, \end{cases} \tag{18}$$

where $\mathrm{succ(t_i)}$ represents the set of immediate successors of t_i and $\mathrm{T_i^o} = \mathrm{T}_i^{\mathrm{ul}} + \mathrm{T}_i^{\mathrm{s}} + \mathrm{T}_i^{\mathrm{dl}}$.

Step 2: Tasks are then embedded into a sequence of vectors as the input of the encoder (the details of the embedded vectors are presented in Section 3.3). Next, the output sequence of the encoder will be used to calculate the context vector. At the jth decoding step, the offloading decision can be generated through $a_j = \arg\max_{a_j} \pi(a_j|s_j)$.

Step 3: The UE and MEC host cooperatively finish executing all tasks according to the offloading decisions.

3.5 The Training Process of the DRLTO

The training goal is to find an optimal policy so that the accumulated reward is maximal, which is expressed as

$$\max_{\theta} L(\theta) = \mathbb{E}\left[\max_{\theta} \pi_\theta(A_{1:n}|G) \sum_{t}^{n} R(s_t, a_t)\right], \tag{19}$$

where θ are the parameters of the S2S neural network, n is the task number of the DAG, $R(s_t, a_t)$ is the reward function, s_t and a_t is the observed state and offloading decision for the tth task, respectively.

To improve the performance and training efficiency, the training target function is modified based on PPO [23], which generates trajectories using the old policy $\pi_{\theta_{\mathrm{old}}}$ and updates the current policy π_θ whose initial value equals $\pi_{\theta_{\mathrm{old}}}$ for several epochs. In order to avoid a large update of the policy, PPO penalizes changes to the policy via a clip function. The clipped target function for the S2S neural network is given by

$$L^{\mathrm{C}}(\theta) = \mathbb{E}\left[\sum_{t=1}^{n} \min\left(\mathrm{pr}_t(\theta)\hat{A}_t, \mathrm{clip}(\mathrm{pr}_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right], \tag{20}$$

Algorithm 1: Off-policy Training for the S2S Neural Network

```
Create two S2S neural networks with θ_old and θ for the old policy π_{θ_old} and target policy π_θ with randomly generated initial values.
θ_old ← θ
for i = 1, 2, ..., l do
    /** Exploration stage **/
    Collect set of trajectories D_i on policy π_{θ_old}.
    for each trajectory τ ∈ D_i do
        Compute advantage estimates Â_1, ..., Â_n according to Eq. (24).
        Compute the target state values v̂_π(s_1), ..., v̂_π(s_n) following the equation: v̂_π(s_t) = Σ_{k=0}^{n-t+1} γ^k r_{t+k}.
        Store advantage estimates and target state values in D_i.
    /** Exploitation stage **/
    for k = 1, 2, ..., m do
        Optimize the target function L^PPO w.r.t. θ_k by taking minibatch SGD (via Adam) using the sampled minibatches from D_i.
    Synchronise old and new parameters: θ_old ← θ.
```

where $\hat{A}_t$ is the estimator of the advantage function at time step t and ϵ is a hyperparameter to control the clip range. $\mathrm{pr}_t(\theta)$ is the policy probability ratio that is given by

$$\mathrm{pr}_t(\theta) = \frac{\pi_\theta(a_t|G, A_{1:t})}{\pi_{\theta_{\mathrm{old}}}(a_t|G, A_{1:t})}. \tag{21}$$

The clip function $\mathrm{clip}(\mathrm{pr}_t(\theta), 1-\epsilon, 1+\epsilon)$ aims to limit the value of $\mathrm{pr}_t(\theta)$, which removes the incentive for moving $\mathrm{pr}_t(\theta)$ outside of the interval $[1-\epsilon, 1+\epsilon]$. Finally, taking the minimum of the clipped and unclipped objective restricts the final objective as a lower bound on the unclipped objective.

As discussed in the previous subsection, a shared parameter S2S neural network is crafted for both policy and value function approximation. To train this neural network, the clipped objective function and the value function loss are integrated into the target function. Besides, adding an entropy bonus can ensure efficient exploration. Following the suggestion in work [23], we combine these terms and obtain the following function as the final training target:

$$L^{\mathrm{PPO}}(\theta) = \mathbb{E}\left[L^{\mathrm{C}}(\theta) - c_1 L^{\mathrm{VF}}(\theta) + c_2 S[\pi_\theta](s_t)\right], \tag{22}$$

where c_1 and c_2 are coefficients, S denotes an entropy bonus, and L^{VF} is a squared-error loss between predicted state values $v_\pi(s)$ and target state values $\hat{v}_\pi(s)$:

$$L^{\mathrm{VF}}(\theta) = \mathbb{E}\left[\sum_{t=1}^{n} \left(v_\pi(s_t) - \hat{v}_\pi(s_t)\right)^2\right], \tag{23}$$

where $\hat{v}_\pi(s_t) = \sum_{k=0}^{n-t+1} \gamma^k r_{t+k}$.

Training the S2S neural network with RL is quite different from training DNN with RL, which can use transition segments (s_t, a_t, r_t, s_{t+1}) for back propagation. When training the S2S neural network, the entire trajectory should

TABLE 1
The Parameters of Simulation Environment

Notation	Parameter	Value
$R_{\mathrm{ul}}, R_{\mathrm{dl}}$	UL/DL Transmission Rate	$\{3, 7, 11, 15, 19\}$ Mbps
f_{l}	CPU Clock Speed of UE	1 GHz
f_{s}	CPU Clock Speed of a VM in MEC Host	4×2.5 GHz
ζ, ρ	Constants in Energy Model	$\zeta = 3$, $\rho = 1.25 \times 10^{-26}$
P^{Tx}	Avg. Wireless Sending Power	1.258 W
P^{Rx}	Avg. Wireless Receiving Power	1.181 W
fat	Width of a DAG	{0.3, 0.4, 0.5, 0.6, 0.7, 0.8}
density	Density of Dependency of a DAG	{0.3, 0.4, 0.5, 0.6, 0.7, 0.8}
ccr	Comm. to Comp. Ratio of a DAG	0.3 to 0.5
data_i^s, data_i^r	Sending/Receiving Data Size for a Task	5 KB to 50 KB
C	Required CPU Cycles for a Task	10^7 to 10^8 cycles/sec
p	Length of Task Indices Vector	12

TABLE 2
The Neural Network and Training Hyperparameters

Hyperparameter	Value	Hyperparameter	Value
Encoder Layers	2	Encoder Layer Type	Bi-LSTM
Encoder Hidden Units	256	Encoder Layer Norm.	On
Decoder Layers	2	Decoder Layer Type	LSTM
Decoder Hidden Units	256	Decoder Layer Norm.	On
Learning Rate	10^{-4}	Activation function	Tanh
Optimization Method	Adam	Loss Coefficient c_1	0.5
Discount Factor γ	0.99	Entropy Coefficient c_2	0.01
Adv. Discount Factor λ	0.95	Clipping Constant ϵ	0.2
Batch Size	500		

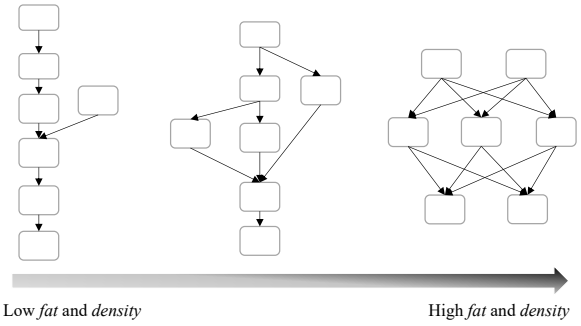

Fig. 4. The examples of synthetic DAGs from low *fat* and *density* to high *fat* and *density*.

be divided into sequences which are then fed into the network. For example, two components of trajectories are firstly sampled from the environment, which are defined by a scheduling plan $A_{1:n}$ and a sequence of state values $[v_\pi(s_1), v_\pi(s_2), \ldots, v_\pi(s_n)]$ obtained from conducting a forward propagation of the S2S neural network with the tasks embedding sequence. Next, the reward sequences $[r_1, r_2, \ldots, r_n]$ can be obtained by applying the scheduling plan to the environment. Furthermore, the TD-error term δ at time step t can be calculated by $\delta_t = r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)$. Finally, the generalized advantage estimator (GAE) [24] is used to obtain the advantage function at time step t as

$$\hat{A}_t = \sum_{k=0}^{n-t+1} (\gamma\lambda)^k (\delta_{t+k}), \tag{24}$$

where λ $(0 < \lambda < 1)$ is used to control the trade-off between bias and variance.

The training algorithm is formally presented in Algorithm 1. First, the sampling and updating neural networks are initialized with the same parameters. In each loop, the sampling neural network is used to sample a set of trajectories which are stored in D_i. Next, the sequence of advantage estimates $\hat{A}$ and the estimated state values $\hat{v}_\pi$ are calculated and stored in D_i. Since then, D_i consists of the tasks embedding sequences $[t_1, t_2, \ldots, t_n]$, sampled scheduling plans $A_{1:n}$, reward sequences $[r_1, r_2, \ldots, r_n]$, sampled state values sequences $[v_\pi(s_1), v_\pi(s_2), \ldots, v_\pi(s_n)]$, advantage estimates sequences $[\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n]$, and estimated state values sequences $[\hat{v}_\pi(s_1), \hat{v}_\pi(s_2), \ldots, \hat{v}_\pi(s_n)]$. At the exploitation stage, the updating neural network is improved by conducting minibatch Stochastic Gradient Descent (SGD) on D_i with the target function defined in Eq. (22) for m epochs. *Adam* is adopted as the optimization method for its efficiency and stability.

4 Results and Discussion

This section presents the experimental results and performance evaluation of the DRLTO. We first present how to set the simulation environment and hyperparameters. Next, the training results of the DRLTO are presented, including the average reward, value loss, and policy loss. Finally, we analyze the performance results of the DRLTO through comparing it with eight existing algorithms.

4.1 Simulation Environment and Hyperparameters

In our simulation experiments, we consider that the UE is in a small cell network with different transmission rates dependent on the distance between the UE and MEC host. The set of transmission rate is {3 Mbps, 7 Mbps, 11 Mbps, 15 Mbps, 19 Mbps}, which covers the most area of a cell from a distal end to a proximal end. We set the constants in the energy model $\rho = 1.25 \times 10^{-26}$ and $\zeta = 3$ according to [3]. The CPU clock speed of the UE f_{l} is set to be 1 GHz. We set the total computation capacity of the edge server as 10 GHz. Moreover, P^{Tx} and P^{Rx} are 1.258 W and 1.181 W, respectively [3].

In MEC, user applications can be modelled as DAGs with various topologies. For example, a data compression application is composed of multiple tasks with linear dependency, while face recognition and gesture capture applications involve more complex inner dependency among tasks (as shown in Fig. 1). Our scheme aims to obtain an effective offloading policy for general purpose, which can adapt to any DAG topologies. To effectively train our DRL-based algorithm, we need the information of task profiles and dependency for many different mobile applications. The current real datasets for mobile applications only contain information of a very limited number of applications [13]. Therefore, we use a synthetic DAG generator [29] to generate various DAGs representing heterogeneous applications. The properties of DAGs are controlled by several parame-

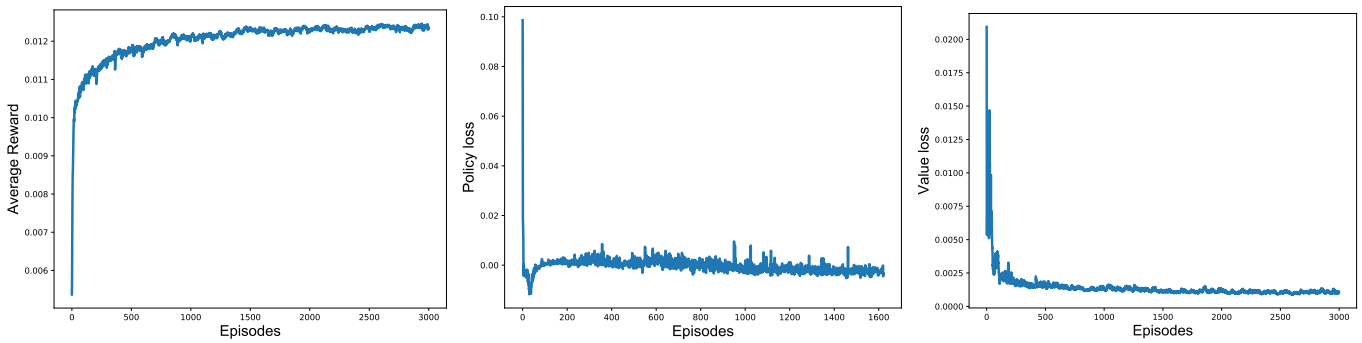

Fig. 5. The values of average reward, policy loss, and value loss in the training process of the DRLTO.

ters including *fat*, *density*, and *ccr*. Here, *fat* is used to control the width and the height of a DAG. *density* determines the number of edges between two levels of a DAG. *ccr* denotes the communication-to-computation ratio, which is the ratio between the communication cost and the computation cost. Using a simulator to generate synthetic DAGs has many benefits. For example, it can fast generate a large number of DAGs representing various mobile applications. In addition, it is easy to control the features of generated DAGs by tuning simulator parameters.

For the DAG generation, we randomly pick *fat* from {0.3, 0.4, 0.5, 0.6, 0.7, 0.8}, *density* from {0.3, 0.4, 0.5, 0.6, 0.7, 0.8} and *ccr* from {0.3, 0.4, 0.5}. It is reasonable to set *ccr* less than 0.5, since many emerging applications are computation-intensive. Fig. 4 shows an example of DAG topologies from low *fat* and *density* to high *fat* and *density*. For the settings of task profile, the transmission data size of a task is set between 5 KB and 50 KB. The required CPU cycles for a task is set between 10^7 and 10^8 cycles/sec. The task number n of the generated DAG ranges from 10 to 50 with a step size of 5. According to the above setting, we randomly generate 500 graphs for each task number as the training dataset and additional 100 graphs for each task number as testing dataset. Specifically, the parameters of our simulation environment are listed in Table 1.

The S2S neural network is implemented via Tensorflow. Specifically, the encoder is set as a two-layer bi-directed Long Short-Term Memory (LSTM) with 256 hidden units while the decoder is set as a two-layer dynamic LSTM also with 256 hidden units. Besides, the layer normalization [30] is added for both encoder and decoder. During the training process, the learning rate is set as 10^{-4}, the coefficients is set as $c_1 = 0.5$, $c_2 = 0.01$, and the batch size is set as 500. Hyperparameters can affect the training results and the convergence speed of DRLTO. We initially set the hyperparameters according to [23], and then run grid search on learning rate, batch size, discount factor, etc., to find the optimal hyperparameters (given in Table 2) for our algorithm. These hyperparameters have been applied to all scenarios in our experiments.

4.2 Compared Algorithms

We compare the performance between the DRLTO and the following eight existing algorithms:

- *Optimal*: The exhaustive search is applied to list all possible solutions and find the optimal one with the highest QoS.
- *Local*: All tasks of the DAG are run on the local processor.
- *Remote*: All tasks of the DAG are offloaded to the MEC host.
- *Random*: Each task of the DAG is randomly assigned to the local processor or the MEC host.
- *Greedy*: Each task of the DAG is greedily assigned locally or remotely based on the estimated finish time on the local processor and the MEC host.
- *Round-Robin (RR)*: Tasks of the DAG are alternately scheduled to the local processor and MEC host.
- *HEFT-based*: Tasks are firstly prioritized according to Heterogeneous Earliest Finish Time (HEFT) as in [2, 31]. The prioritized tasks are then scheduled to the resource with earliest estimated finish time.
- *Double Deep-Q Network based Task Offloading (DDQNTO)*: Tang *et al.* [10] combined LSTM, dueling deep Q-network (DQN), and double-DQN techniques to handle the task offloading problem without considering the inner dependency. Specifically, we use the same exploration-exploitation strategy as in the work [10] to train the Q-networks.

4.3 Training Performance and Convergence

We first investigate the training overheads, convergence property, and inference latency in our algorithm. In our experiments, we set two different targets: *latency-optimal (LO)* and *energy-efficient (EE)*. The LO target aims at minimizing the latency, thus we set $\lambda_t = 1.0$ and $\lambda_e = 0.0$. The EE target aims to jointly optimize the latency and energy consumption, so we set $\lambda_t = \lambda_e = 0.5$. We conduct the training process for the DRLTO and record the average reward, the policy loss (i.e., L^{C} as defined in Eq. (20)), and the value loss (i.e., L^{VF} as defined in Eq. (23)). We train DRLTO on our workstation with GeForce RTX 2080. Each episode involves 40 mini-batch updates for the S2S neural network with the time overheads around 30 s/episode. Fig. 5 depicts the training results with the LO target. We notice that the average reward increases sharply at the beginning and steadily grows after 500 episodes. Our experimental results show that the training converges at around 1200 episodes. The total training overheads for a converged policy is around 10 hours. As the training dose not happen

TABLE 3
The comparison of the DRLTO and existing algorithms in terms of average latency (ms) of a DAG with different numbers of tasks (*n*). N/A denotes cases unable to find optimum. LO and EE denote latency-optimal and energy-efficient targets, respectively.

n	Optimal (LO)	Local	Remote	Random	Greedy	RR	HEFT-based	DDQNTO (LO)	DDQNTO (EE)	**DRLTO (LO)**	**DRLTO (EE)**
10	488.61	723.14	694.75	650.01	556.70	646.92	533.25	580.30	690.04	**491.98**	**583.03**
15	661.57	1053.44	991.49	915.54	780.73	886.51	743.20	836.01	987.36	**673.28**	**804.57**
20	851.71	1394.49	1322.87	1170.06	999.73	1134.41	958.82	1100.71	1317.71	**880.89**	**1010.94**
25	N/A	1796.05	1630.79	1430.03	1239.45	1395.91	1194.95	1349.15	1616.12	**1054.31**	**1221.08**
30	N/A	2154.71	1981.61	1727.75	1500.83	1692.38	1454.53	1629.57	1970.39	**1297.08**	**1507.35**
35	N/A	2463.69	2251.70	2043.32	1757.25	2025.92	1719.99	1906.46	2238.88	**1530.58**	**1767.64**
40	N/A	2910.47	2757.82	2360.31	1992.33	2246.18	1939.06	2286.65	2737.24	**1757.42**	**2067.79**
45	N/A	3182.15	2833.30	2442.66	2074.41	2323.27	2043.41	2365.28	2806.76	**1781.23**	**2051.24**
50	N/A	3662.96	3560.65	2897.15	2431.63	2754.37	2391.01	2948.88	3538.77	**2200.68**	**2533.18**

TABLE 4
The comparison of DRLTO and existing algorithms in terms of QoS (targeting at EE) with different numbers of tasks (*n*), N/A denotes cases unable to find optimum. EE denotes energy-efficient target.

n	Optimal (EE)	Local	Remote	Random	Greedy	RR	HEFT-based	DDQNTO (EE)	**DRLTO (EE)**
10	0.394	0	0.360	0.223	0.194	0.223	0.219	0.339	**0.389**
15	0.410	0	0.367	0.228	0.222	0.252	0.246	0.353	**0.399**
20	0.426	0	0.373	0.254	0.244	0.266	0.265	0.357	**0.415**
25	N/A	0	0.390	0.273	0.270	0.290	0.286	0.381	**0.434**
30	N/A	0	0.388	0.265	0.262	0.281	0.276	0.377	**0.426**
35	N/A	0	0.390	0.255	0.252	0.267	0.260	0.380	**0.426**
40	N/A	0	0.373	0.270	0.275	0.290	0.288	0.368	**0.417**
45	N/A	0	0.398	0.285	0.295	0.307	0.302	0.391	**0.441**
50	N/A	0	0.363	0.274	0.290	0.298	0.296	0.357	**0.411**

very frequently, the training overheads are acceptable. For the training loss, both the value loss and the policy loss approximate to zero after 500 episodes, which shows the good convergence property of the DRLTO. After training, we obtain two trained S2S neural networks, one of which is used for the LO target and the other is used for the EE target. Those trained networks will then be deployed back to the UE. To evaluate the inference speed, we use our laptop with CPU only (2.6 GHz 6-Core Intel Core i7) to do the inference. We feed 100 DAGs with $n = 15$ to the trained S2S neural networks for network inference. As a result, the total inference latency is around 108 ms with 1.08 ms per DAG. Compared to the time overheads of offloading tasks (as shown in Table 3), the inference overhead can be neglect. Note that, we do not involve any inference optimisation method to our trained model. However, many existing works show that the inference speed can be further improved on mobile devices. In the following sections, we evaluate DRLTO among different scenarios.

4.4 Evaluation with Different Task Numbers

First, we investigate the latency, energy consumption, and QoS of different offloading algorithms with varying numbers of tasks. In this case, we fix the transmission rate at 7 Mbps. When aiming at the LO target, we focus on the latency. Table 3 lists the average latency of executing a DAG with different numbers of tasks. The *Optimal (LO)* is implemented using exhaustive search to find the optimal solution. However, the *Optimal (LO)* has exponential time complexity, thus it is unable to find optimum in a reasonable amount of time when $n \geq 20$. The *Local* and the *Remote* both perform poorly in this scenario, which are even worse than the *Random*. *DDQNTO (LO)* cannot learn effective policy, which obtains higher average latency than the heretic search algorithm *HEFT-based*. Compared to the existing algorithms, *DRLTO* with the LO target significantly outperforms all heuristic baselines (from the *Local* to the *HEFT-based*) and the advanced DRL-based methods (i.e., *DDQNTO*). Moreover, *DRLTO (LO)* approximates the optimal solution when $n \leq 20$.

When aiming at the EE target, we jointly consider the latency and energy consumption. The comparison results of the energy consumption and QoS with different numbers of tasks are presented in Fig. 6 and Table 4, respectively. Offloading computation-intensive applications to the MEC host can help reduce the energy consumption on UE, therefore the *Remote* achieves the lowest energy consumption. However, the *Remote* has the highest latency as shown in Table 3. *DDQNTO (EE)* learns offloading policies that have similar energy consumption and latency as *Remote* in all cases. It seems that *DDQNTO (EE)* fails to learn a good trade-off between energy consumption and latency. On the contrary, *DRLTO* with the EE target can learn the optimal policy by taking both the latency and energy into account. Comparing the results in Table 3 and Fig. 6, we find that *DRLTO* with the EE target achieves the lowest energy consumption bar *Remote* and *DDQNTO (EE)*, while it still obtains acceptable latencies (close to the *Greedy*). Furthermore, Table 4 gives explicit results related to the QoS of all evaluated algorithms. Obviously, the QoS of the *Local* is always zero because we define QoS as a measurement of the gain/loss of an algorithm compared to the *Local*. Moreover, *DRLTO* with the EE target achieves the maximal QoS compared to all baseline algorithms (i.e., from *Local* to

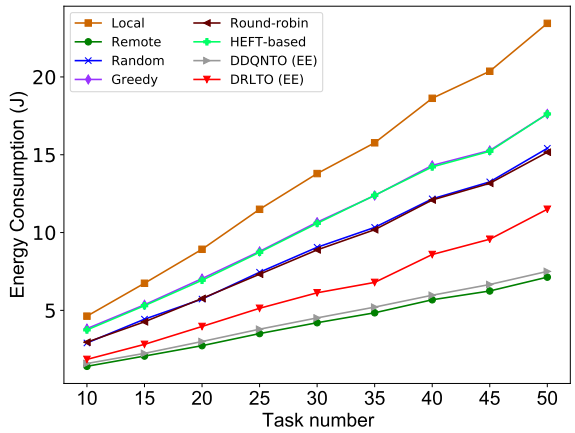

Fig. 6. The comparison of the DRLTO and existing algorithms in terms of average energy consumption (targeting at EE) with different numbers of tasks. EE denotes energy-efficient target.

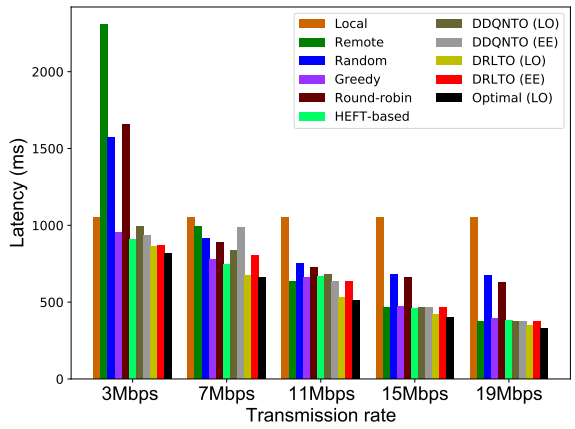

Fig. 7. The comparison of the DRLTO and existing algorithms in terms of average latency with different transmission rates. LO and EE denote latency-optimal and energy-efficient targets, respectively.

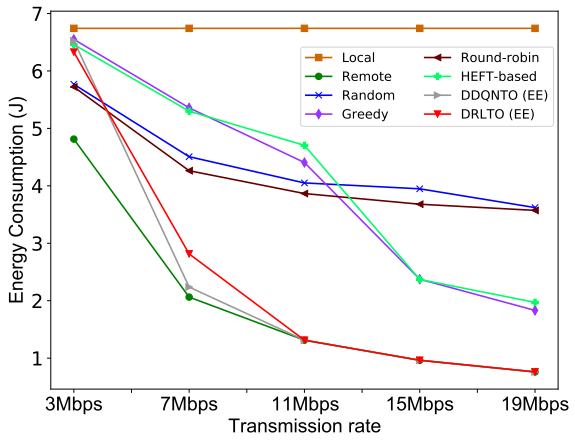

Fig. 8. The comparison of the DRLTO and existing algorithms in terms of average energy consumption (targeting at EE) with different transmission rates. EE denotes energy-efficient target.

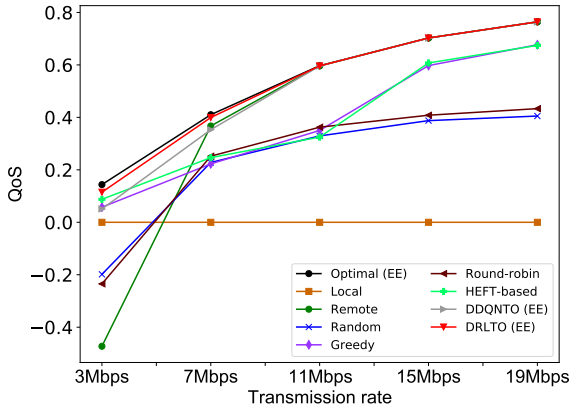

Fig. 9. The comparison of the DRLTO and existing algorithms in terms of QoS (targeting at EE) with different transmission rates. EE denotes energy-efficient target.

DDQNTO) and approximates the optimal solution.

4.5 Evaluation with Different Transmission Rates

Next, we evaluate the performance of the *DRLTO* with different transmission rates. We first target at LO, and the results are shown in Fig. 7. When the transmission rate is low (meaning that the UE is far away from the MEC host), offloading tasks to the MEC host will result in high latency. On the contrary, if the transmission rate is high, offloading tasks can significantly reduce the latency. An efficient algorithm should automatically adapt its offloading policy to various transmission rates. As shown in Fig. 7, *DDQNTO (LO)* cannot learn effective policy, which achieves the worse performance than *HEFT-based* when the transmission rate increases from 3 Mbps to 15 Mbps. In contrast, *DRLTO (LO)* has high adaptability, which outperforms all baseline algorithms (i.e., from *Local* to *DDQNTO*) and approximates the optimal solution with various transmission rates.

When the target is EE, we need take energy consumption into consideration. Figs. 8 and 9 show the energy consumptions and QoS of various algorithms with different transmission rates, respectively. As expected, *Remote* consumes the least energy on UE since all tasks are run remotely. However, the latency of the *Remote* could be high and unacceptable under low or medium transmission rates. As shown in Fig. 9, the *Remote*, *Random*, and *RR* even achieve negative QoS when the transmission rate is 3 Mbps, meaning that they are worse than the *local*. *DRLTO* with the EE target achieves the lowest energy consumption bar the *Remote* and *DDQNTO* when transmission rate is greater or equal than 7 Mbps, while it still leads to acceptable latencies. For example, when the transmission rate is 7 Mbps, *DRLTO* with the EE target obtains a similar latency as the *Greedy* but 50% less energy consumption than the latter, while both *DDQNTO* and *Remote* achieve the higher latency than *Random*. Fig. 9 demonstrates that the QoS of all algorithms (except the *Local*) increases with the transmission rate. This is because the communication cost declines as the transmission rate grows, offloading tasks to remote servers can be beneficial. In addition, *DRLTO* with the EE target obtains the maximal QoS compared with all baselines and approximates the optimal solution.

4.6 Evaluation with/without Dependency

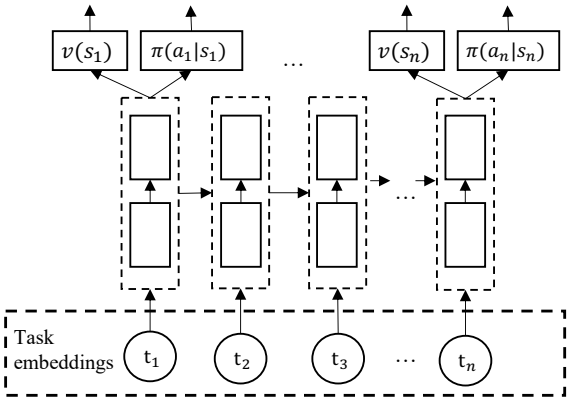

Fig. 10. The neural network architecture without considering the dependency among tasks.

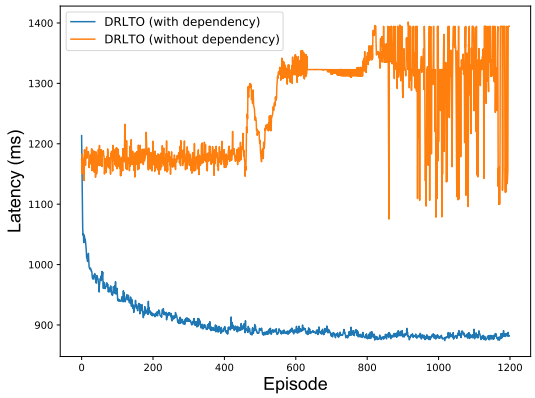

Fig. 11. Evaluation results for DRLTO with/without dependency information.

We embed the dependent information into the task embeddings and use the encoder network with the attention mechanism to extract features from the task embedding. In order to show how the dependent information influences the results. As shown in Fig. 10, we remove the encoder network with attention mechanism from the S2S neural network and directly input the task embeddings to the decoder network. The output of the decoder network remains the same (i.e., the policy and the value function). More specifically, we remove the adjacent information from the task embeddings. We then train this policy network on the same training dataset as DRLTO targetting at LO and use the same hyperparameters (i.e., learning rate, batch size, and the number of gradient steps per episode). During the training, we evaluate the trained policy at each episode on the testing dataset with $n = 15$. Fig. 11 shows the evaluation results. We find that the training process cannot converge without considering the dependency information. Note that *DDQNTO* proposed in [10] does not involve the dependency information either. The above experiment results show that *DRLTO* can learn better policies than *DDQNTO* in all scenarios. Therefore, dependency information is one of the crucial factors in achieving good performance.

5 Related Work

The MEC offloading problem has attracted considerable research interests. Mach *et al.* [32] gave a thorough survey for offloading architecture and solutions in MEC systems. Many existing studies assume the offloading tasks are independent. Chen *et al.* [4] focused on task offloading in a software-defined ultra-dense network. They formulated the task offloading problem as a mixed-integer non-linear programming (MINLP) problem and solved it by decomposing the problem into two sub-problems. Dinh *et al.* [3] aimed to minimize both latency and energy consumption by jointly optimizing the task allocation decision and the CPU frequency. They proposed two heuristic methods to solve the offloading problem for the cases with both fixed and elastic CPU frequency. Zeng *et al.* [33] formulated the task offloading problem with joint consideration of task scheduling and image placement as an MINLP problem, and proposed a three-stage heuristic algorithm to solve it. When considering the inner dependency of offloading tasks, Lin *et al.* [2] represented mobile applications as task graphs and proposed a heuristic-based algorithm to solve the task offloading problem in MEC. Neto *et al.* [34] proposed a user-level online offloading scheme for MEC, which makes offloading decisions based on the estimation of energy and execution time for the dependent tasks. These existing solutions rely on hand-tuned heuristic or approximation methods. However, turning heuristics for a given task offloading scenario is an expensive job that requires considerable human expertise. Consequently, facing the dynamic MEC scenarios, one might have to do this tuning repeatedly, which is time-consuming and sometimes impractical. By contrast, the proposed DRLTO learns an adaptive model by interacting with the MEC environment, without relying on hand-turned heuristics.

Recently, DRL-based methods have emerged in the solutions to the MEC offloading problem [9, 10, 11, 12, 35, 36, 37]. Li *et al.* [35] proposed a deep Q-Learning based offloading method to jointly optimize the offloading decision and computational resource allocation. Chen *et al.* [36] considered an ultra-dense network, where multiple base stations can be selected for offloading. They also adopted deep Q-Learning to obtain the offloading strategy. Dinh *et al.* [37] focused on multi-user multi-edge-node task offloading problem by using Q-learning in MEC. Zhan [9] *et al.* formulated the task offloading problem as a partially observable Markov decision process (POMDP) and applied a policy gradient DRL-based approach to solve the problem. Zou *et al.* [12] proposed a DRL-based offloading method by utilizing the asynchronous advantage actor-critic algorithm, to reduce the latency and energy consumption. Tang *et al.* [10] incorporated the LSTM, duelling DQN, and double DQN techniques to solve the task offloading problem in MEC. All of these studies assume that the offloading tasks are independent. However, most of the real-world applications consist of dependent tasks, ignoring the task dependency when making offloading decisions can lead to severe performance degradation. To address the above issue, our DRL-based method considers the dependency among tasks through using DAG models for user applications. Different from our previous work [11], DRLTO develops a new learning

model with redefined state space and reward function to optimise the QoS in terms of both the latency and energy consumption. Moreover, this work conducts comprehensive simulation experiments to show the convergence property, latency, energy consumption, and QoS of the DRLTO in dynamic MEC scenarios.

From the theoretical perspective, the offloading problem in MEC systems can be seen as a combinatorial optimization problem. Recent researches attempt to use deep learning-based methods to solve such problems. For example, Pointer Networks [27] are variants of attention-based S2S neural networks. Pointer Networks use attention as a pointer to select a member of the input sequence as the output. This well-designed neural network fits one type of combinatorial optimization problem where the output elements are selected from inputs, e.g., TSP. Kool *et al.* [28] adopted attention layers rather than Pointer Networks as the policy neural network for solving TSP. Dai *et al.* [38] provided a different way to solve TSP with RL by using graph embedding technology for learning an indirect policy. The learned greedy policy behaves like a meta-algorithm that incrementally constructs a solution. These studies aim at solving classical combinatorial optimization problems such as TSP, MaxCut, and Convex Hull with neural networks and RL, which inspired us to integrate S2S architecture with RL to solve the MEC offloading problem.

6 Conclusion and Future Work

In this paper, we investigate the task offloading problem in MEC considering task dependency, with the aim of jointly optimizing the latency and energy consumption. To effectively adapt to dynamic scenarios, we propose a new offloading scheme that embeds DRL training and inference procedures into the MEC system. Specifically, we model the offloading problem as an MDP and combine an S2S neural network to approximate both the policy and value function of the MDP. An efficient policy gradient method is then applied for training the S2S neural network. The training results show that the DRLTO achieves excellent stability and convergence with reasonable training and inference overheads. Through comparing with the existing state-of-the-art heuristic and DRL-based algorithms, we demonstrate that the DRLTO has strong adaptability among different MEC scenarios and can obtain near-optimal solutions.

Our future work intends to solve task offloading problem with a shared resource allocation strategy (i.e., UEs share a common resource pool on an MEC server and the offloaded tasks will compete for the common resources). One potential solution for the task offloading problem with shared resources is to use multi-agent reinforcement learning [39]. Each UE in the MEC will be treated as an individual agent and learns its own policy to achieve a common goal (e.g., achieving the maximal total QoS for all involved UEs) by collaborating with the other agents.

Acknowledgment

This work was partly supported by the EU Horizon 2020 INITIATE project under the Grant Agreement No.101008297.

References

[1] D. Sabell, V. Sukhomlinov, L. Trang, S. Kekki, P. Paglierani, R. Rossbach, X. Li, Y. Fang, D. Druta, F. Giust *et al.*, "Developing software for multi-access edge computing," *ETSI White Paper*, vol. 20, 2019.

[2] 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 Transactions on Services Computing*, vol. 8, no. 2, pp. 175–186, 2015.

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

[4] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 587–597, 2018.

[5] M. Du, Y. Wang, K. Ye, and C. Xu, "Algorithmics of cost-driven computation offloading in the edge-cloud environment," *IEEE Transactions on Computers*, vol. 69, no. 10, pp. 1519–1532, 2020.

[6] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Reward-oriented task offloading under limited edge server power for multi-access edge computing," *IEEE Internet of Things Journal*, 2021.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference of Learning Representations, ICLR*, 2016.

[9] Y. Zhan, S. Guo, P. Li, and J. Zhang, "A deep reinforcement learning based offloading game in edge computing," *IEEE Transactions on Computers*, vol. 69, no. 6, pp. 883–893, 2020.

[10] M. Tang and V. W. Wong, "Deep reinforcement learning for task offloading in mobile edge computing systems," *IEEE Transactions on Mobile Computing*, 2020.

[11] 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 Communications Magazine*, vol. 57, no. 5, pp. 64–69, 2019.

[12] J. Zou, T. Hao, C. Yu, and H. Jin, "A3c-do: A regional resource scheduling framework based on deep reinforcement learning in edge scenario," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 228–239, 2021.

[13] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 43–56.

[14] X. Sun and N. Ansari, "Adaptive avatar handoff in the cloudlet network," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 664–676, 2019.

[15] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, "Avatar: Mobile distributed computing in the cloud," in *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2015, pp. 151–156.

[16] W. Zhan, C. Luo, G. Min, C. Wang, Q. Zhu, and H. Duan, "Mobility-aware multi-user offloading optimization for mobile edge computing," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 3, pp. 3341–3356, 2020.

[17] H. Cao and J. Cai, "Distributed multiuser computation offloading for cloudlet-based mobile cloud computing: A game-theoretic machine learning approach," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 1, pp. 752–764, 2017.

[18] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning, ICML*. PMLR, 2016, pp. 1928–1937.

[21] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," in *Reinforcement Learning*. Springer, 1992, pp. 5–32.

[22] A. Doerr, M. Volpp, M. Toussaint, T. Sebastian, and C. Daniel, "Trajectory-based off-policy deep reinforcement learning," in *International Conference on Machine Learning, ICML*. PMLR, 2019, pp. 1636–1645.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[24] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *International Conference of Learning Representations, ICLR*, 2016.

[25] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems, NIPS*, 2014, pp. 3104–3112.

[26] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International Conference of Learning Representations, ICLR*, 2015.

[27] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems, NIPS*, 2015, pp. 2692–2700.

[28] W. Kool and M. Welling, "Attention, learn to solve routing problems!" in *International Conference of Learning Representations, ICLR*, 2019.

[29] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2013.

[30] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," in *Advances in Neural Information Processing Systems, NIPS*, 2016.

[31] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[32] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communication Survey & Tutorial*, vol. 9, no. 3, pp. 1628–1656, 2017.

[33] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3702–3712, 2016.

[34] J. L. D. Neto, S.-Y. Yu, D. F. Macedo, J. M. S. Nogueira, R. Langar, and S. Secci, "Uloof: a user level online offloading framework for mobile edge computing," *IEEE Transactions on Mobile Computing*, vol. 17, no. 11, pp. 2660–2674, 2018.

[35] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for mec," in *Wireless Communications and Networking Conference, WCNC, 2018 IEEE*. IEEE, 2018, pp. 1–6.

[36] 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 of Things Journal*, vol. 6, no. 3, pp. 4005–4018, 2018.

[37] T. Q. Dinh, Q. D. La, T. Q. Quek, and H. Shin, "Learning for computation offloading in mobile edge computing," *IEEE Transactions on Communications*, vol. 66, no. 12, pp. 6353–6367, 2018.

[38] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems, NIPS*, 2017, pp. 6348–6358.

[39] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

Jin Wang received the BEng and MEng degrees in computer science from the University of Electronic Science and Technology of China (UESTC), Chengdu, China, in 2014 and 2017, respectively. He is currently working toward the PhD degree in computer science at the University of Exeter. His research interests include deep reinforcement learning, applied machine learning, cloud and edge computing, and computer system optimization.

Jia Hu received the BEng and MEng degrees in electronic engineering from the Huazhong University of Science and Technology, China, in 2006 and 2004, respectively, and the PhD degree in computer science from the University of Bradford, UK, in 2010. He is a senior lecturer of computer science at the University of Exeter. His research interests include edge-cloud computing, resource optimization, applied machine learning, and network security.

Geyong Min received the BSc degree in computer science from the Huazhong University of Science and Technology, China, in 1995, and the PhD degree in computing science from the University of Glasgow, United Kingdom, in 2003. He is a professor of high performance computing and networking with the Department of Computer Science within the College of Engineering, Mathematics and Physical Sciences at the University of Exeter, United Kingdom. His research interests include computer networks, wireless communications, parallel and distributed computing, ubiquitous computing, multimedia systems, modeling and performance engineering.

Wenhan Zhan is a Senior Experimentalist in Computer Science at the University of Electronic Science and Technology of China (UESTC), Chengdu, China. He received his B.E., M.Sc., and Ph.D. degrees from UESTC, in 2010, 2013, and 2020, respectively. From 2018 to 2019, he worked as a Visiting Scholar in the Department of Computer Science at the University of Exeter, UK. His research interests mainly lie in Distributed System, Cloud Computing, Edge Computing, and Artificial Intelligence.

Albert Y. Zomaya is currently the Chair Professor of High Performance Computing & Networking in the School of Computer Science, University of Sydney. He is also the Director of the Centre for Distributed and High Performance Computing which was established in late 2009. Professor Zomaya was an Australian Research Council Professorial Fellow during 2010-2014 and held the CISCO Systems Chair Professor of Internetworking during the period 2002–2007 and also was Head of School for 2006–2007.

Nektarios Georgalas is a Principal Researcher with the Applied Research department of British Telecom. In his current role, he leads two collaborative research programmes with key BT partners delivering innovations in the areas of cloud, data centres, network virtualisation, smart cities, IoT and mobility. During his career with BT, since 1998, he has managed numerous collaborative and internal research projects in areas such as network management, market-driven data management systems, policy-based management, distributed information systems, SOA/Web services, model driven design and development of telecoms OSS, cloud and NFV. He is inventor and co-inventor of 11 patents. He has also authored more than 60 papers in international journals and conferences. He has served as general co-chair, programme cochair, programme committee and keynote speaker or invited panelist in top international IEEE academic and TMForum conferences.