Layer Dependency-aware Learning Scheduling Algorithms for Containers in Mobile Edge Computing

Zhiqing Tang, Jiong Lou, and Weijia Jia, *Fellow, IEEE*

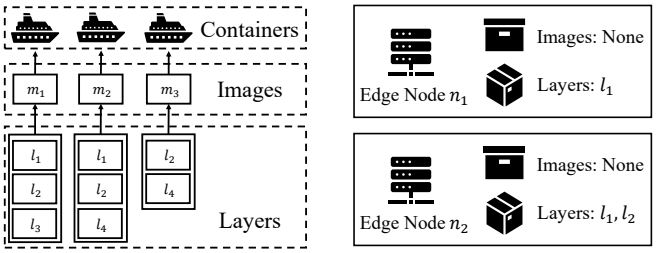
Abstract—Due to the features of lightweight and easy deployment, the use of containers has emerged as a promising approach for Mobile Edge Computing (MEC). Before running the container, an image composed of several layers must exist locally. However, it has been conspicuously neglected by existing work that task scheduling at the granularity of the layer instead of the image can significantly reduce the task completion time to further meet the real-time requirement and resource efficiency in resource-limited MEC. To bridge the gap, considering the complex dependency between layers and images, a novel layer dependency-aware container scheduling algorithm is proposed to reduce the total task completion time. Specifically: 1) We model the online layer dependency-aware scheduling problem for containers in a heterogeneous MEC, considering the layer download time and task computation time. 2) A policy gradient algorithm is proposed to solve this problem, and the high-dimensional and low-dimensional relations for layer dependencies are extracted with improved action selection. 3) Experiments based on the real-world data trace show that the proposed algorithm outperforms the image-based and layer-based baseline algorithms by 54% and 19% on average, respectively.

Index Terms—Mobile edge computing, dependency-aware scheduling, container, reinforcement learning.

1 Introduction

Mobile Edge Computing (MEC) is becoming more and more popular in recent years. Various heterogeneous edge nodes are deployed at the edge of the core network to provide or supplement computing capabilities [1]. By deploying mobile applications at edge nodes, application latency can be significantly reduced, such as AR and VR applications [2], [3], etc. To effectively utilize resources and deploy applications on edge nodes, the container is widely used [4]–[7]. Before running a container, an image file must exist locally, including the code, binaries, system tools, configuration files, etc. [8]. Otherwise, it must be downloaded from a registry [9].

However, the download time of multiple images will be very long since the limited bandwidth resources in MEC, which takes up a higher proportion of the total task completion time as most tasks are delay-sensitive and do not last long [7], [10], [11]. Much of existing work to reduce the download time is cloud-oriented, focusing on changes to the registry [12], [13]. Slacker [14] reduces startup times by fetching individual files from the registry on demand, which would scale poorly when the latency is getting larger. Cntr [15] and Pocket [16] move common parts of multiple containers to a common daemon process. All these approaches require substantial changes to the application and mitigate the advantage of container isolation.



(a) Containers, images, and layers (b) Edge nodes and layers

Fig. 1: An illustrative example of containers, images, and layers in MEC.

It has been conspicuously neglected by existing work that each image is stored in the unit of the layer [17]. The layers can be shared by multiple images. For example, as shown in Fig. 1(a), the image m_1 is comprised of three layers l_1, l_2, and l_3. While the layer l_1 is shared by images m_1 and m_2. Existing container cluster management tools such as Kubernetes [18], KubeEdge [19], K3s [20], Akraino [21], etc., make the scheduling decisions at the granularity of the image. They consider the image to exist if and only if all layers required by the image exist locally. Thus, the scheduler cannot fully use the local layer information, and the selected node may download many duplicate layers [11], [22]. For example, it is assumed that an image m_1 consists of three layers. As shown in Fig. 1(b), one of the layers exists on one node, and two exist on the other node. From the image-based scheduling perspective, the two nodes are the same, which is coarse-grained.

- *Corresponding author: Weijia Jia.*
- *Zhiqing Tang and Jiong Lou are with Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, 200240, China, and also with BNU-UIC Joint AI Resrach Institute, Beijing Normal University & UIC (Zhuhai), Guangdong, 519087, China. E-mail: {domain, lj1994}@sjtu.edu.cn*
- *Weijia Jia is with BNU-UIC Institute of Artificial Intelligence and Future Networks Beijing Normal University (BNU Zhuhai), Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College Zhuhai, Guangdong, 519087, PR China. E-mail: jiawj@uic.edu.cn*

It is very promising to make the scheduling decisions at the granularity of the layer instead of the image to reduce the download time and then further reduce the task completion time in MEC. Nevertheless, the following challenges need to be solved. First, how to fully extract and utilize the complex layer dependencies. Existing researchers have proposed a layer-match algorithm based on local layer size for non-heterogeneous MEC [7]. However, images have different download times on different nodes due to the heterogeneity of edge nodes in a real MEC scenario. Besides, there exist hidden dependencies between different tasks since the layer sharing among multiple images. These layer dependency features are numerous and sparse, and there are correlations and hidden relationships between the features. Common machine learning methods such as Convolutional Neural Network (CNN) or Recurrent Neural Network (RNN) cannot perform feature extraction well. To fully extract the layer dependencies, in this paper, a Factorization Machines (FM) based layer interaction feature extraction method is proposed [23]. Different dimensions of layer dependency are extracted with weight-sharing embedding layers and FM layers [24], which are further combined to assist in scheduling tasks.

The second challenge is making online scheduling decisions based on the extracted layer dependency to gain long-term benefits in less task completion time, i.e., the sum of download time and computation time. Compared with heuristic algorithms, the Reinforcement Learning (RL) algorithm can fully consider the impact of continuous decisions [25]. Moreover, the long-term benefits and the impact of layer dependency can be fully considered with a reward function. Thus, RL-based algorithms are suitable for online decision-making, and a policy gradient-based RL algorithm is further proposed to reduce the task completion time [26].

In this paper, we are the first team to model the task scheduling problem at the granularity of the layer in heterogeneous MEC, aiming at the minimization of the total task completion time. A Layer Dependency-aware Learning Scheduling (LDLS) algorithm is proposed based on the policy gradient RL algorithm. The resources of heterogeneous edge nodes, the features of tasks, and the layer dependencies are fully considered the input state. The FM-based method is used to extract the layer dependency features. Moreover, constraints are added on the action selection to avoid terrible actions, e.g., scheduling tasks to heavily loaded nodes or nodes with insufficient storage space, which has always been a complicated problem [27]. The RL agent's policy network and value function are also carefully designed to make the feature embeddings and combinations. Finally, experiments are conducted based on real-world data trace to verify the performance of the algorithm. The data is crawled from Docker Hub [28]. The proposed algorithm is compared with the default scheduling algorithm of Kubernetes and the state-of-the-art layer-based heuristic algorithms. Experimental results show that the proposed algorithm has better performance than all baseline algorithms.

The contributions are summarized as follows.

1) We formulate the newly identified Layer Dependency-aware Scheduling (LDS) problem in heterogeneous MEC scenarios to minimize the overall task completion time. The download time of layers and the computation time of task execution are considered with heterogeneous edge nodes.
2) To continuously make the online scheduling decisions, a novel LDLS algorithm based on policy gradient RL is proposed with improved action selection. An FM-based embedding method is proposed to extract both the high-dimensional and low-dimensional layer dependency features.
3) Real-world data sets are used for evaluation. The proposed algorithms are compared with several state-of-the-art baselines. The experimental results show that the LDLS algorithm outperforms the image-based and layer-based algorithms by 54% and 19% on average, respectively.

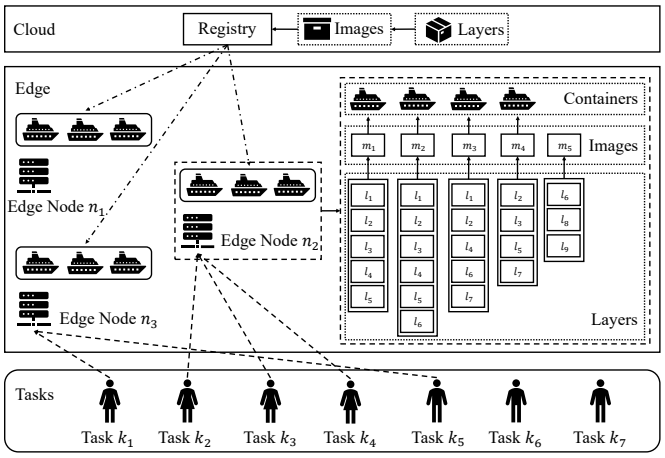


Fig. 2: An example of layer dependency-aware scheduling in MEC.

The rest of the paper is organized as follows. In Section 2, the related work and motivation are introduced. System model and problem formulation are described in Section 3. Dependency scheduling algorithm is proposed in Section 4. Performance is evaluated in Section 5. Some issues are discussed in Section 6 and Section 7 concludes the paper.

2 Related Work and Motivation

2.1 Container in Mobile Edge Computing

The container, image, layer, and their relations are illustrated in Fig. 2.

Container: Containers are based on lightweight OS-level virtualization technology that isolates and manages an application's resource usage and optionally provides tools for managing the application's dependencies. The significant benefits of containers are lightweight resource isolation and container images [7].

Image: An image is a read-only template with instructions for creating a container, which includes all its dependencies, including the code, binaries, system tools, and configuration files, etc.

Layer: Containers are stored by a layered file system. Each layer encapsulates a set of files and directories put

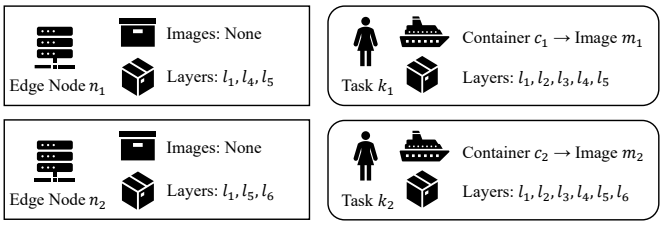


Fig. 3: Edge nodes and tasks

TABLE 1: Layer size

Layer	l_1	l_2	l_3	l_4	l_5	l_6
Size	1 MB	2 MB	2 MB	3 MB	6 MB	20 MB

together when the image is built and associated with a collision-resistant hash digest taken over its content.

As shown in Fig. 2, containers are running on edge nodes to tackle the tasks from a set of users. A container is an instance of an image. Each image contains several layers, and the layers can be shared by multiple images. For example, the image m_1 contains layers l_1 - l_5. And the image m_3 contains layers l_1, l_2, l_4, l_6, and l_7. The layers l_1, l_2, and l_4 are shared by images m_1 and m_3.

All layers contained in the requested image must exist on the edge node. Otherwise, it needs to be downloaded from the registry in the cloud. The registry stores all images comprised of layers. Layers are stacked on top of each other in a particular order to form the requested image. However, the construction order of the layer has no impact on the scheduling decision. Because before constructing the image, all the layers contained in this image must exist locally. Only the existence of the layer will affect the scheduling decisions. So from the perspective of task scheduling, the layers contained in each image form a set, not a sequence.

Containers have been widely deployed in cloud data centers [29]–[32]. However, current container deployment approaches are not suitable for the MEC scenarios. First, pulling images from the registry in the cloud to an edge node takes a long time over high-latency, or low-bandwidth links [7]. Then, limited edge resources combined with user mobility means that applications deployed at each edge node change frequently [33]. Placing a local registry or cache at every edge node can be expensive and unrealistic [7]. Much of existing work to reduce the startup time is cloud-oriented, focusing on changes to the registry, which mitigate the advantage of container isolation [12]–[16], [34], [35].

2.2 Reinforcement Learning

Reinforcement learning has been widely used for task scheduling in MEC since it can make continuous real-time decision-making. Tang et al. [36] investigate the channel model in the heterogeneous network and propose a novel deep RL-based algorithm to allocate radio resources in MEC dynamically. Qian et al. [37] apply a Deep Q-Network (DQN) method to solve the user cache optimization and base station cache optimization problems in MEC. Xiong et al. [38] propose a deep RL approach to minimize the long-term weighted sum of average completion time of jobs and the average number of requested resources. Xu et al. [39] leverage RL techniques to automatically configure the control and networking systems under a dynamic industrial MEC environment. Wang et al. [40] use an RL network to solve the joint node selection and cache replacement problem in MEC.

RL is very suitable for solving the LDS problem because the impact of the continuous decision is significant when layer dependency is considered. However, no RL-based method is ever proposed for such an LDS problem to our best knowledge. Besides, the complexity of the MEC heterogeneous environment also brings challenges to RL's state input. To better understand the scheduling process, more discussions are given in the following subsection.

TABLE 2: Download size for different scheduling policies

Method		k_1	k_2	Layers	Size
Imaged-based	Greedy	n_1	n_1	n_2 l_2, l_3 n_2 l_6	24 MB
	Optimal	n_1	n_1	n_1 l_2, l_3 n_2 l_6	24 MB
Layer-based	Greedy	n_1	n_2	n_1 l_2, l_3 n_2 l_2, l_3, l_4	11 MB
	Optimal	n_2	n_2	n_1 None n_2 l_2, l_3, l_4	7 MB

2.3 Case Study

As shown in Fig. 3, it is assumed that there are two edge nodes n_1 and n_2 (The detailed notations are defined in the next section). On node n_1, there exist three layers l_1, l_4, and l_5. And on node n_2, there are layers l_1, l_5, and l_6. Suppose there are two different types of image m_1 and m_2. Image m_1 is composed of layers $l_1 - l_5$. And image m_2 is composed of layers $l_1 - l_6$. There are two users generating tasks k_1 and k_2, and requesting containers c_1 and c_2, respectively. Besides, containers c_1 and c_2 are based on images m_1 and m_2, respectively.

It is assumed that the arrival of task k_1 and task k_2 are in sequence. Since the decision is made online, task k_2 is not known when task k_1 is scheduled. The sizes of the layers are shown in TABLE 1. The decision-making results are analyzed as follows.

The scheduling results of different policies are shown in TABLE 2. First, let us consider image-based scheduling. Since node n_1 and node n_2 do not have all the layers required by images m_1 and m_2, there is no image on node n_1 or n_2 from the perspective of the scheduler, i.e., node n_1 and node n_2 are the same. In this case, the scheduling results are the same (assuming to traverse in order), i.e., first task k_1 is scheduled to node n_1, and then when task k_2 is scheduled, there is still no image m_2 on both nodes, so it is scheduled to node n_1. When task k_1 is scheduled, layers l_2 and l_3 need to be downloaded, and the download cost is $2 + 2 = 4$ MB. When task k_2 is scheduled, the layer l_6 is needed. As a result, the total download size is $4 + 20 = 24$ MB.

Secondly, when the tasks are scheduled based on layer. For greedy strategy based on download cost, when task k_1 is scheduled, the download cost required by node n_1 is $2+2 = 4$ MB (Layers l_2 and l_3 are needed) and the download cost required by node n_2 is $2 + 2 + 3 = 7$ MB (Layers l_2, l_3,

and l_4 are needed), so task k_1 is scheduled to node n_1. Then when task k_2 is scheduled, the download cost required by node n_1 is 20 MB (Layer l_6), and the cost required by node n_2 is $2+2+3=7$ MB (Layers l_2, l_3, and l_4), so task k_2 is scheduled to node n_2. The total cost is 11 MB.

However, the greedy algorithm based on layer download size is not optimal. For example, if task k_1 is scheduled to node n_2, and then task k_2 is also scheduled to node n_2, the total download size is only $2+2+3=7$ MB (Layers l_2, l_3, and l_4), which is better than the greedy algorithm. This requires that the possible impact between the continuous decisions can be considered when scheduling. Coincidentally, the reward of RL algorithms can solve this problem very well, which is suitable for solving continuous decision-making problems [25].

Based on the observations, it is projected that the layer dependency-aware scheduling can effectively tackle with through RL, which has been ignored by current work.

3 System Model and Problem Formulation

In this section, the system model of MEC is first defined. Then the cost is introduced. Finally, the LDS problem is formulated and analyzed.

3.1 System Model

In MEC, edge nodes are deployed close to users. The user connects to the nearest base station. Some delay-sensitive tasks from users are first transmitted to the base stations, then scheduled to different edge nodes for processing. The requested services are created on the corresponding edge nodes. Currently, the services require containers to run, and containers' operation requires container images, which are further composed of layers. For ease of reference, the main notations used in this paper are summarized in TABLE 3.

To illustrate the problem, an MEC scenario is considered. A set of tasks $\mathbf{K}=\{k_1,k_2,...,k_{|\mathbf{K}|}\}$ is generated from different users and offloaded to edge nodes for processing, where $|\cdot|$ is used to indicate the number of elements in the set, e.g., $|\mathbf{K}|$ is the number of tasks. To process the tasks, a group of different containers $\mathbf{C}=\{c_1,c_2,...,c_{|\mathbf{C}|}\}$ is created and deployed on a set of edge nodes. Each container needs an image file. The set of images is denoted as $\mathbf{M}=\{m_1,m_2,...,m_{|\mathbf{M}|}\}$. Since requesting a container is equivalent to requesting the corresponding image, and the difference between a container and an image is only a writable container layer, the two concepts are unified [8]. In other words, the task requests the container, and the container needs the corresponding layers. The set of layers is denoted as $\mathbf{L}=\{l_1,l_2,...,l_{|\mathbf{L}|}\}$.

The set of edge nodes is denoted as $\mathbf{N}=\{n_1,n_2,...,n_{|\mathbf{N}|}\}$, which is deployed at the edge of core network. And the remote cloud is considered as another edge node $n_{|\mathbf{N}|+1}$ with unlimited computation resources. For a node $n\in\mathbf{N}$, it maintains three sets: the set of running containers $\mathbf{C}_n(t)\subset\mathbf{C}$, the set of local images $\mathbf{M}_n(t)\subset\mathbf{M}$, and the set of local layers $\mathbf{L}_n(t)\subset\mathbf{L}$. Besides, each node has its CPU frequency f_n, bandwidth b_n, and storage space d_n. The number of containers that each node can run simultaneously is limited, i.e., the node n can run up to C_n containers at the same time.

TABLE 3: Notations

Section 3	
$\mathbf{K}$	Task set
k	Task ($k\in\mathbf{K}$)
$\|\mathbf{K}\|$	Number of tasks
$\mathbf{C}$	Container set
c	Container ($c\in\mathbf{C}$)
$\mathbf{M}$	Image set
$\mathbf{L}$	Layer set
l	Layer ($l\in\mathbf{L}$)
$\mathbf{N}$	Edge node set
n	Edge node ($n\in\mathbf{N}$)
$n_{\|\mathbf{N}\|+1}$	Remote cloud
$\mathbf{C}_n(t)$	Running container set on node n at time t ($\mathbf{C}_n(t)\subset\mathbf{C}$)
$\mathbf{M}_n(t)$	Local image set on node n at time t ($\mathbf{M}_n(t)\subset\mathbf{M}$)
$\mathbf{L}_n(t)$	Local layer set on node n at time t ($\mathbf{L}_n(t)\subset\mathbf{L}$)
f_n	CPU frequency of node n
b_n	Bandwidth of node n
d_n	Storage space of node n
C_n	Max number of running containers on node n
$\mathbf{L}_c$	Layer set contained in container c
x_c^l	Variable to indicate whether container c contains layer l
d_l	Size of layer l
p_k	Requested CPU resource of task k
c_k	Requested container of task k
n_k	Assigned node of task k
u_k^n	Variable to indicate whether task k is scheduled to node n
$y_n^l(t)$	Variable to indicate whether layer l is on node n at time t
$z_n^l(t)$	Download finish time for layer l on node n
T_k^d	Download time of task k
T_k^c	Computation time of task k
T_k	Total task completion time of task k
Section 4	
s_t	Edge system state
$s_t^{n,l}$	Layer state for node n
$T_n^r(t)$	Total remaining download time
$s_t^{n,r}$	Resource state for node n
$s_t^{\mathbf{N}}$	State for all nodes
$s_t^{k,n,e}$	Estimated number of layers that need to be downloaded
$s_t^{k,n,d}$	Estimated layer download time
$s_t^{k,n,w}$	Estimated waiting time
$s_t^{k,n,p}$	Estimated computation time
s_t^k	State for the task k
a_t	Action
r_t	Reward function
$T_n^d(t)$	Download finish time
T_n^d	Download time of node n
$R(\tau)$	Cumulative reward
γ	Discount factor ($\gamma\in(0,1)$)
π	Policy
$A^\pi(s,a)$	Advantage function
$V^\pi(s)$	Value function
$Q^\pi(s,a)$	State-action value function
$\hat{A}_t$	Estimator of the advantage function
$L(\theta)$	Loss function
$\mathbf{D}$	Replay memory
D^n	Counter of transitions
π^*	Best policy
u_{a_t}	Prejudgment of the action a_t
u^N	Maximum number of sampling times
u^n	Sampling times

Moreover, the set of layers contained in container $c \in \mathbf{C}$ is $\mathbf{L}_c = \{x_c^l | l \in \mathbf{L}\}$, where $x_c^l = 1$ if the container c contains layer l. Otherwise, $x_c^l = 0$. Besides, the size of layer $l \in \mathbf{L}$ is d_l. For each task $k \in \mathbf{K}$ generated from a user at time t, the requested CPU resource is p_k, and the requested container is c_k. After scheduling, the node assigned by this task is represented as $n_k = \{u_k^n | n \in \mathbf{N} \cup \{n_{|\mathbf{N}|+1}\}\}$, where $u_k^n = 1$ if the task k is scheduled to node n, otherwise, $u_k^n = 0$.

3.2 Cost

In the MEC scenario, user tasks' completion time is closely related to user experience [41]. Generally, task completion time mainly includes the initialization time (download time) and the computation time. Since the transmission time from the user to the wireless base station is very small compared with the download time and computation time, it is not considered [7], [42], [43]. For example, a face recognition task or an object detection task usually takes hundreds of megabytes to download an image, but only a few hundreds of kilobytes to transfer the task data [2], [44], [45].

It is assumed that task k requesting container c is scheduled to node n at time t. The times are calculated as follows.

Download Time: To calculate the download time, the variable $y_n^l(t) \in \{0, 1\}$ is introduced. If layer l is on node n at time t, $y_n^l(t) = 1$, 0 otherwise. $z_n^l(t) \in [0, +\infty)$ is used to denote the download finish time for layer l on node n. If layer l already exists or has not started to download, then $z_n^l(t) = t$.

For each node, it is assumed that only one layer can be downloaded at one time. Each node has a layer download queue. If a new layer needs to be downloaded, it is added to this queue and may have to wait for another layer in the download process. Therefore, the download time for task k can be obtained as the maximum download finish time of all required layers:

$$T_k^d = \max_{l \in \mathbf{L}} \left(z_n^l(t) \times x_c^l \right) - t. \tag{1}$$

Computation Time: The computation time is the processing time of the task k on the node n, which can be obtained as follows:

$$T_k^c = \frac{p_k}{f_n}. \tag{2}$$

As a result, the total task completion time of task k is calculated as follows:

$$T_k = T_k^d + T_k^c. \tag{3}$$

If the task is scheduled to the cloud, all layers must be downloaded every time and cannot be shared since the cloud is serverless [46]. In addition, scheduling to the cloud incurs some additional costs. So we do not consider the situation where all tasks are scheduled to the cloud.

3.3 Problem Formulation

In this subsection, some constraints are first introduced, then the layer dependency-aware scheduling problem is formulated and analyzed.

Constraints: It is assumed that the scheduler is located at the remote cloud or a master node [4]. When scheduling, the decision made needs to meet the limit of the number of containers running simultaneously of the node and the storage resource limit of the node. The container number limit is described as:

$$|\mathbf{C}_n(t)| \leq C_n, \quad \forall t, \forall n. \tag{4}$$

And the storage resource limit of each node is defined as follows:

$$\sum_{l \in \mathbf{L}} \left(1 - y_n^l(t)\right) \times d_l \leq d_n, \quad \forall t, \forall n. \tag{5}$$

Moreover, each task should be scheduled to only one node or the cloud, which is represented as:

$$\sum_{n \in \mathbf{N} \cup \{n_{|\mathbf{N}|+1}\}} u_k^n = 1, \quad \forall k. \tag{6}$$

Problem Formulation: We aim to minimize the overall task completion time from a long-term perspective, which is defined in Eq. (3). The target is to find the best strategy which can minimize the overall time while obeying the constraints. Therefore, the LDS problem in MEC is defined as follows:

Problem 1.

$$\min T = \sum_{k \in \mathbf{K}} T_k, \tag{7}$$

$$\begin{aligned} s.t. \quad & (4) - (6) \\ & x_c^l \in \{0, 1\}, y_n^l(t) \in \{0, 1\}, z_n^l(t) \in [0, +\infty), \\ & \forall n \in \mathbf{N}, \forall k \in \mathbf{K}, \forall l \in \mathbf{L}. \end{aligned} \tag{8}$$

Problem Analysis: Problem 1 is an advanced bin-packing problem, which is NP-hard and can only be solved heuristically. The goal is to make online decisions in a dynamic MEC system and obtain long-term benefits. However, decisions are made according to a deterministic strategy at each time slice for most of the existing heuristic algorithms. This strategy is fixed and cannot consider the dynamic MEC environment and the impact of continuous decisions, which makes them unstable in MEC. For meta-heuristic algorithms, all future information needs to be known if used to solve this problem from a long-term perspective. Nevertheless, the tasks arriving in the future are unknown, and the MEC environment is dynamically changing. So it is challenging to use meta-heuristic algorithms to solve this problem. In addition, if a meta-heuristic algorithm is used only to solve the decision of one single time slice, this requires many rounds of iteration, which is very time-consuming. Moreover, the impact between the previous and subsequent time slices cannot be considered in this way. Getting better decisions for a single time slice does not mean getting better results in the long term, as already explained in the case study in Section 2.3. As a result, most of the existing heuristic and meta-heuristic algorithms are unstable in a real MEC environment. They cannot consider the impact of continuous decisions and are unable to achieve fast decision-making when facing large-scale problems.

In this problem, the first-order transition probability of the tasks' resource demand is quasi-static for an extended period and not uniform distribution by adequately choosing the time slice duration [47]. Moreover, the arrival of tasks and the environment's update have the memoryless

property [4]. Therefore, this problem can be modeled as a Markov Decision Process (MDP).

Reinforcement learning-based algorithms are suitable for solving MDP problems [48]. In RL algorithms, at each time t, the RL agent collects system state s_t, and calculates the reward during last time slice r_{t-1}. Then, the agent selects action a_t according to a pre-defined strategy. After performing the action, the system transits to the new state s_{t+1} in the next time slice. Similarly, the RL agent repeats the above operations, i.e., calculating reward r_t and selecting new action a_{t+1} according to s_{t+1}. Based on the collected state, action, reward, and a proper discount factor, a value can be calculated to denote the expected long-term return with discount, as opposed to the short-term reward. The reward is an immediate signal received in a given state, while value is a long-term expectation. The RL agent might receive a low, immediate reward even as it selects an action with great potential for long-term value. By value function, the RL agent can optimize the policy and make decisions from a long-term perspective.

There are many kinds of RL algorithms. One of the essential branching points is whether the agent has a model or learns a model of the environment [25]. With a model, a function can predict state transitions and rewards. If the agent wants to use a model in the MEC scenario, it must learn the model purely from experience. However, various heterogeneous features and other MEC biases are learned, resulting in an agent that performs well concerning the learned model but behaves sub-optimally or terribly in the real MEC environment. Also, model-based RL is not very robust and cannot adapt to changing MEC environments [26]. Therefore, the model-free RL algorithm is selected.

With model-free RL, there are two main approaches to representing and training agents: policy-based approach (e.g., policy gradient) and value-based approach (e.g., Q-learning) [25], [49], [50]. The policy-based approach learns a policy explicitly as $\pi_\theta(a_t|s_t)$, while the Q-learning approach learns an approximator $Q_\theta(s_t, a_t)$. Then, the decision is made according to the policy or approximator. In MEC, Q-learning has the following apparent shortcomings compared with the policy gradient method: 1) The state of the MEC system is huge and complicated, which makes it very difficult for Q-learning to represent and calculate the Q-value well. While the policy gradient method can extract and represent the MEC system with the policy network. 2) The output of Q-learning is a deterministic strategy. Nevertheless, there are usually multiple suitable scheduling strategies during a time slice in MEC. Although Q-learning has an action exploration mechanism, it only randomly selects actions to avoid falling into the local optimum. Therefore, the performance is not as good as the policy-based method that directly outputs stochastic strategies. 3) When selecting actions, the action with the max Q-value is selected. This makes Q-learning more likely to select overestimated values, which will result in over-optimistic estimates of Q-values. The policy gradient method directly evaluates the policy network through the value function. And the action is selected according to the policy, which avoids the overestimation caused by Q-value and tends to make the training stable and reliable. Although the first shortcoming can be solved by combining deep neural networks with Q-learning, the latter two shortcomings are inherent problems. As a result, the policy gradient method can achieve better performance in the MEC environment.

4 Our Algorithms

In this section, the algorithm settings are first introduced. Then, the LDLS algorithm is illustrated.

4.1 Algorithm Settings

The main components of RL are the agent and the MEC environment. The agent makes scheduling decisions. To train an agent, the state, action, reward, and policy are needed.

State: A state s_t is a complete description of the MEC environment, which contains two aspects: the nodes and the task. To fully explore the layer dependency on each node and consider the computation resources, the state of node n is divided into the following two parts.

Layer Information: To fully extract the layer dependency information, the layer distribution on each node is very significant, i.e., $y_n^1(t) \cdots y_n^{|\mathbf{L}|}(t)$. Besides, if the layer is downloading, the remaining download time is also critical, which is denoted as $z_n^1(t) \cdots z_n^{|\mathbf{L}|}(t)$. Finally, due to the heterogeneity of layers, the size of each layer also affects decision-making. As a result, the layer state for node n can be denoted as:

$$s_t^{n,l} = \begin{bmatrix} y_n^1(t) & \cdots & y_n^{|\mathbf{L}|}(t) \\ z_n^1(t) & \cdots & z_n^{|\mathbf{L}|}(t) \\ d_{l_1} & \cdots & d_{l_{|\mathbf{L}|}} \end{bmatrix}. \tag{9}$$

Resource Information: Node resources related to scheduling mainly include the CPU frequency f_n and bandwidth b_n. The total remaining download time $T_n^r(t)$ on the node is also essential information for the agent, which is obtained as:

$$T_n^r(t) = T_k^d - t. \tag{10}$$

Then, the resource state for node n is denoted as:

$$s_t^{n,r} = \left[f_n, b_n, T_n^r(t)\right]. \tag{11}$$

Finally, the state for all nodes are defined as follows:

$$s_t^{\mathbf{N}} = \left\{ s_t^{n,l}, s_t^{n,r} | n \in \mathbf{N} \right\}. \tag{12}$$

Secondly, for task k, the set of requested layers $\mathbf{L}_k$ can be obtained by the requested container c_k as $\mathbf{L}_k = \mathbf{L}_c$. Besides, some estimated evaluations are carried out to let the agent better understand each node's situation. Specifically, for the task k, the number of layers that need to be downloaded $s_t^{k,n,e}$, layer download time $s_t^{k,n,d}$, waiting time $s_t^{k,n,w}$, and computation time $s_t^{k,n,p}$ required for this task to be scheduled to node n is calculated. This information is critical for the agent and directly affects which node the task is scheduled to. The estimated information is obtained as follows:

$$\begin{aligned} s_t^{k,n,e} &= \sum_{l \in \mathbf{L}} x_c^l \times \left(1 - y_n^l(t)\right), \\ s_t^{k,n,d} &= \sum_{l \in \mathbf{L}} \frac{x_c^l \times \left(1 - y_n^l(t)\right) \times d_l}{b_n}, \\ s_t^{k,n,w} &= \max_{l \in \mathbf{L}} \left(z_n^l(t) \times x_c^l\right) - t. \end{aligned} \tag{13}$$

And $s_t^{k,n,p}$ is obtained according to Eq. (2). Then the state for the task is obtained as follows:

$$s_t^k = \{\mathbf{L}_c, p_k\} \cup \left\{ s_t^{k,n,e}, s_t^{k,n,d}, s_t^{k,n,w}, s_t^{k,n,p} | n \in \mathbf{N} \right\}. \quad (14)$$

Finally, the state is defined as:

$$s_t = s_t^{\mathbf{N}} \cup s_t^k. \quad (15)$$

Action: The action space is the combination of the edge node set $\mathbf{N}$ and the remote cloud $n_{|\mathbf{N}|+1}$, which can be denoted as:

$$a_t \in \mathbf{N} \cup \{n_{|\mathbf{N}|+1}\}. \quad (16)$$

Reward: The reward function r_t is critically important. The agent's goal is to maximize the reward, while in MEC, the goal is to minimize the task completion time, so the reward is obtained as $r_t = -T_k$.

The environment returns the reward after the action execution. Nevertheless, it is also estimated to do more training to improve the performance. The estimation of reward mainly estimates the download time defined in Eq. (1). Layers are divided into three types: already existing, downloading, and not existing. For layers that already exist, the download time is 0. For layers that are downloading, download finish time is taken as their total finish time, i.e., $\max_{l \in \mathbf{L}} \left(z_n^l(t) \times x_c^l \right)$.

Besides, for those layers that are needed by the requested image but have not yet started to download, the download finish time $T_n^d(t)$ is obtained as follows:

$$T_n^d(t) = \max\left(t, T_n^d(t-1)\right) + \sum_{l \in \mathbf{L}} \frac{d_l \times (1 - y_n^l(t)) \times 1 \cdot \left\{ z_n^l(t) = t \right\} \times x_c^l}{b_n}, \quad (17)$$

where $1 \cdot \{\}$ is the Iverson bracket, which is equivalent to 1 when the condition is satisfied. Otherwise, it is equivalent to 0. At time 0, $T_n^d(0) = 0$. Based on these, the download time of node n can be obtained as:

$$T_n^d = \max\left(\max_{l \in \mathbf{L}} \left(z_n^l(t) \times x_c^l \right), T_n^d(t) \right) - t. \quad (18)$$

The goal is to maximize the cumulative reward over long-term scheduling, which is denoted as $R(\tau)$ with a discount factor $\gamma \in (0, 1)$:

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t. \quad (19)$$

Policy: A policy is a rule used by the agent to decide what actions to take, which is usually denoted by π, i.e., $a(t) \sim \pi(\cdot|s(t))$. The probability of the scheduling process in MEC scenarios is defined as:

$$P(\tau|\pi) = \rho_0(s_0) \Pi_{t=0}^{T-1} P(s_{t+1}|s_t, a_t) \pi(a_t|s_t), \quad (20)$$

where $\rho_0(s_0)$ is the start-state distribution. The expected return denoted by $J(\pi)$ is obtained as:

$$J(\pi) = \int_\tau P(\tau|\pi) R(\tau) = \mathbb{E}_{\tau \sim \pi}[R(\tau)]. \quad (21)$$

The Problem 1 can be transformed into an optimal policy problem, which aims to obtain the optimal policy π^*:

$$\pi^* = \arg\max_\pi J(\pi). \quad (22)$$

4.2 Layer Dependency-aware Learning Scheduling Algorithm

The policy gradient-based LDLS algorithm is introduced in this section, including policy optimization, policy network, and action selection.

Policy Optimization: Policy gradient methods compute an estimator of the policy gradient and plug it into a stochastic gradient ascent algorithm, where the advantage function is crucially important [49]. The advantage function indicates the relative advantage of each action, which is denoted as:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s), \quad (23)$$

where the value function $V^\pi(s)$ gives the expected return if the agent starts in state s and acts according to policy π, which is defined as:

$$V^\pi(s) = \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s]. \quad (24)$$

And the state-action value function $Q^\pi(s, a)$ is defined as:

$$Q^\pi(s, a) = \mathbb{E}_{\tau \sim \pi}\left[R(\tau)|s_0 = s, a_0 = a\right]. \quad (25)$$

Algorithm 1 LDLS

Input: s_0
Output: π^*

```
for epoch = 1, 2, ... do
    Initialize D = ∅, D^n = 0
    Reset environment
    Get state s_0
    for t = 1, 2, ... do
        Run policy π_θold in environment
        Store transition (s_t, a_t, r_t, r_t(θ), s_{t+1}) in D
        D^n ← D^n + 1
        if D^n mod |D| = 0 then
            Compute Â_1, ..., Â_T by Eq. (29)
            Compute L(θ) by Eq. (30)
            Optimize network and update weights θ_old ← θ
        end if
        if done then
            Break
        end if
    end for
end for
Return π*
end
```

The most commonly used gradient estimator $\hat{g}$ has the form of:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right], \quad (26)$$

which is obtained by differentiating the loss function $L_{PG}(\theta)$ of policy gradient:

$$L_{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right], \quad (27)$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at time step t. However, it is appealing to perform multiple steps of optimization on the loss $L_{PG}(\theta)$ using the same scheduling trace. And it often

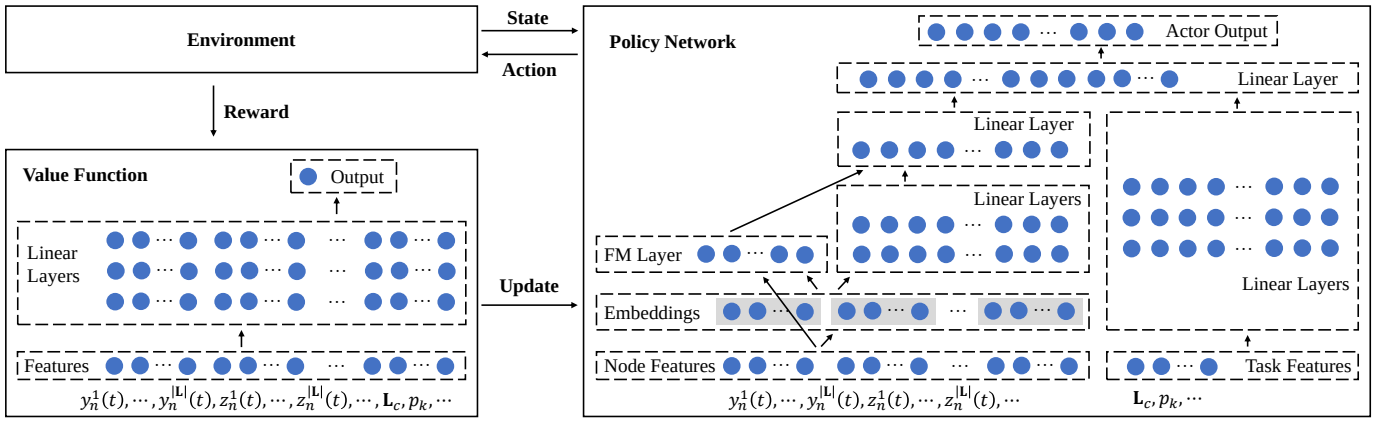


Fig. 4: Overview of the LDLS algorithm. The main components include the policy network and the value function network. The policy network comprises the FM component and several linear neural network layers, and the value function is composed of several linear layers. The main steps of the algorithm include system state observation, action selection, reward calculation, network update, etc.

leads to destructively large policy updates. To solve these problems, the loss function can be customized as [51]:

$$L_{TRPO}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right], \quad (28)$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, and $r_t(\theta_{\text{old}}) = 1$. Besides, the estimated advantage $\hat{A}_t$ is calculated as:

$$\begin{aligned} \hat{A}_t = & -V(s_t) + r_t + \gamma r_{t+1} + \\ & \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T). \end{aligned} \quad (29)$$

Without a constraint, maximization of $L_{TRPO}(\theta)$ would lead to an excessively large policy update. Hence, the loss is further customized into [26]:

$$L(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t \right) \right], \quad (30)$$

where ϵ is a hyperparameter, e.g., $\epsilon = 0.2$. And $\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)$ is used to clip the probability, i.e., removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. Then, the minimum of the clipped and unclipped objective is taken.

The LDLS algorithm is shown in Algorithm 1. For each epoch, the replay memory $\mathbf{D}$ is first initialized as an empty set. D^n is a counter of transitions, where the transition is denoted as $(s_t, a_t, r_t, r_t(\theta), s_{t+1})$. Then, the environment is reset, and the initial state s_0 is obtained. The agent runs the policy in the environment for each decision-making time, i.e., observes the system state, selects an action, and then calculates the reward. The transition is then stored. If enough transitions have been collected, the network is trained as lines 9 - 13. Then, if all tasks are tackled, the epoch is finished. Finally, the best policy π^* is returned.

The interactions between the environment, the value function, and the policy network are shown in Fig. 4. The details of the networks are described as follows.

Policy Network: There is a significant challenge in MEC to model feature interactions among layers effectively. For example, different images may share some layers, and the distribution of these layers in different nodes can affect the scheduling results. Furthermore, the layer dependency interactions are hidden in data and challenging to identify a priori, especially when the number of features is extensive.

The machine learning-based method is promising for feature extraction [23]. Some ideas extend CNN or RNN to predict the feature interactions [52], [53]. However, CNN-based models are biased to the interactions between neighboring features, while RNN-based models are more suitable for sequential dependency.

As mentioned in Section 2.1, when making scheduling decisions, we only consider whether the layer exists locally, without considering the order of the image construction. In other words, there is neither a neighbor feature nor a sequential dependency. None of CNN or RNN apply to layer dependency. FM [24] models pairwise feature interactions as the inner product of latent vectors between features and shows promising results. To derive a learning model that is able to learn feature interactions of all orders in an end-to-end manner, the FM model combined with a deep neural network is adopted [23].

As shown in Fig. 4, the feature extraction layers of the node consist of two components: FM component and deep component. The FM component models pairwise feature interactions as the inner product of respective feature latent vectors. It can capture order-2 feature interactions among layers much more effectively since the layer dependency is sparse. The output of FM is the summation of an addition unit and a number of inner product, which is defined as:

$$h_n = \langle w, e \rangle + \sum_{j_1=1}^{d} \sum_{j_2=j_1+1}^{d} \langle F_i, F_j \rangle e_{j_1} \cdot e_{j_2}, \quad (31)$$

where $w \in \mathbf{R}^d$ and $F_i \in \mathbf{R}^k$. The addition unit $\langle w, e \rangle$ reflects the importance of order-1 features, and the latter inner product unit represent the impact of order-2 feature interactions. d is the dimension of the embedding layers, and e_j is the embedding vector.

The deep component is a feed-forward neural network. To obtain better feature interactions among nodes, a h_n is calculated for each node. As shown in Fig. 4, each node's features are grouped as the input. Then, an embedding layer

is used to compress the input vector to a low-dimensional, dense real-value vector before further feeding into the first hidden layer. Moreover, the latent feature vectors F in FM now serve as network weights learned and used to compress the input field vectors to the embedding vectors. The neural network weights are shared between edge nodes, and an output for all nodes is obtained with the node output layer.

Besides, the task features are input to the hidden linear layers, and another output layer is used to obtain the overall feature of the task. After that, the nodes' and the task's outputs are combined to get the policy network's final output.

Action Selection: When selecting actions, the agent samples based on the probability output by the policy network. It does not judge whether the action is reasonable. However, in the MEC scenario, many actions are very terrible. For example, when a selected node runs a large number of containers, which results in high resource utilization, or a node has stored many other layers, there is not enough space to store the required layers, or some other layers must be deleted. To avoid these actions, some constraints are added to the action selection process.

First, for the sampled node $a_t = n$, if the number of containers running on the node exceeds a threshold C_n, then the node is considered to be under a high load, which can be denoted as:

$$u_{a_t}^c = 1 \cdot \{|\mathbf{C}_n(t)| + 1 \leq C_n\}, \tag{32}$$

where $u_{a_t}^c = 1$ means that the action is acceptable, otherwise it is a terrible action.

Algorithm 2 Action Selection

Input: a_t
Output: a_t
1: Initialize $u^n = 0$
2: Get u_{a_t} by Eq. (34)
3: **while** $u_{a_t} < 1$ **do**
4: **if** $u^n \geq u^N$ **then**
5: $a_t = n_{|\mathbf{N}|+1}$
6: break
7: **end if**
8: Sample a_t again
9: $u^n = u^n + 1$
10: Get u_{a_t} for new action by Eq. (34)
11: **end while**
12: Return a_t
13: **end**

Secondly, if the sum of the downloaded and downloading layer sizes on the selected node exceeds a threshold, then scheduling tasks to this node causes the layer to fail to download the required layers or some layers must be deleted to succeed, which can be obtained as:

$$u_{a_t}^s = 1 \cdot \left\{ d_n(t) + \left(1 \cdot \left\{ z_n^l(t) > 0 \right\} + x_c^l\right) \times d_l \leq d_n \right\}, \tag{33}$$

where $d_n(t)$ is the available storage space on node n at time t. If $u_{a_t}^s = 1$, the action is acceptable. Otherwise, it is not a good action. Besides, it is always acceptable to schedule the tasks to the remote cloud at a higher cost. The prejudgment of the action a_t can be summarized as:

$$u_{a_t} = u_{a_t}^c u_{a_t}^s + 1 \cdot \{a_t = n_{|\mathbf{N}|+1}\}. \tag{34}$$

If $u_{a_t} \geq 1$, then a_t is a feasible action. Otherwise, the action is sampled again.

The action selection algorithm is shown in Algorithm 2. A counter u^n is first initialized to 0. The prejudgment u_{a_t} is obtained by Eq. (34). Then, if $u_{a_t} < 1$, which means the action is not feasible, the action is sampled again, as shown in lines 3 - 11. Denote the maximum number of sampling times as u^N and the sampling times as u^n. If the number of samples exceeds the maximum limit of the counter, the task is scheduled to the cloud.

4.3 Computational Complexity Analysis

The analysis of computational complexity is as follows. As shown in Fig. 4, the primary process of the LDLS algorithm consists of four steps: system state observation, action selection, reward calculation, and network update.

First, the system state is obtained according to Eq. (15). As illustrated in Section 3.1, there are $|\mathbf{N}|$ nodes and $|\mathbf{L}|$ layers. Thus, the complexity of Eq. (15) is $O(|\mathbf{N}||\mathbf{L}|)$. Second, the action is selected according to the policy network and updated according to Algorithm 2. For Algorithm 2, it mainly contains a while loop. The sample operation (line 8) in the loop will call the policy network. The time complexity of the policy network is only related to the network size, which can be considered a constant time O_t. So the complexity of Algorithm 2 is $O(u^N O_t)$, where u^N is a constant. More details on the complexity of the policy network will be introduced in the network update step. Third, the reward is defined as $r_t = -T_k$ and calculated according to Eq. (3). So the complexity of reward calculation is $O(|\mathbf{L}|)$. These steps are executed sequentially so that they can be completed in polynomial time.

Moreover, to evaluate the complexity of the network update, a theoretical analysis of the computational complexity of the policy network and value function based on floating point operations (FLOPs) is performed, which is widely used to measure the computational complexity of deep learning models [54], [55].

In the policy network as shown in Fig. 4, for the input of node features, there is an embedding layer, an FM layer, and three linear layers. The embedding layer is a dictionary lookup, so it has 0 FLOPs [56]. For the FM layer, the FLOPs is calculated as $(|\mathbf{N}| + |\mathbf{L}| + 1)d$ [57], where d is the dimension of the embedding layers. Denote the input and output dimensions of the j-th linear layers (from bottom to top) are H_i^j and H_o^j, respectively. The FLOPs of the three linear layers are $(2H_i^1 - 1)H_o^1$, $(2H_i^2 - 1)H_o^2$, and $(2H_i^3 - 1)H_o^3$, respectively [58]. Besides, for the input of task features, there are three linear layers. The total number of FLOPs of these three linear layers is $\sum_{j=4}^{6}(2H_i^j - 1)H_o^j$. Finally, the actor output layer is linear, so its FLOPs is $(2H_i^7 - 1)H_o^7$. Therefore, the total FLOPs of the policy network is $(|\mathbf{N}| + |\mathbf{L}| + 1)d + \sum_{j=1}^{7}(2H_i^j - 1)H_o^j$.

In addition, the value function has three linear layers, so the total FLOPs of value function is $\sum_{j=8}^{10}(2H_i^j - 1)H_o^j$. Usually, a linear layer is followed by a non-linear activation

TABLE 4: Statistics of images and layers

Type	Image	Layer
Number	70	378
Max Size	9371Mb	809Mb
Min Size	291Mb	1Mb
Average Size	3369Mb	414Mb

function, such as a ReLU or a Softmax [59]. It is common not to count these operations, as they only take up a tiny fraction of the overall time. For example, a ReLU is just $y = max(x, 0)$. On a fully-connected linear layer with H_o^j output neurons, the ReLU uses H_o^j of these computations, i.e., it has H_o^j FLOPs. Compared with matrix multiplies and inner products, the FLOPs of the activation function can be ignored. From the analysis above, we can see that the computational complexity of the FM layer is far less than the linear layers. Therefore, compared with traditional deep RL algorithms, combining FM to the policy network will only slightly increase the computational complexity in the process of decision making.

5 Evaluation

In this section, the performance of the proposed algorithm is evaluated. The experimental settings are first introduced. Then, the experimental results are presented and analyzed.

5.1 Experimental Settings

This subsection introduces the data preprocessing, baseline algorithms, parameter settings, and simulator setup.

Data Preprocessing: The popular images are crawled from real-world container data [28]. The relations between images and layers are extracted. After preprocessing, 70 images and 378 layers are extracted for experiments. The average number of layers per image is 8.6, and more statistical information about images and layers is shown in TABLE 4. For each user, the requested container is generated according to uniform or Zipf distribution [60].

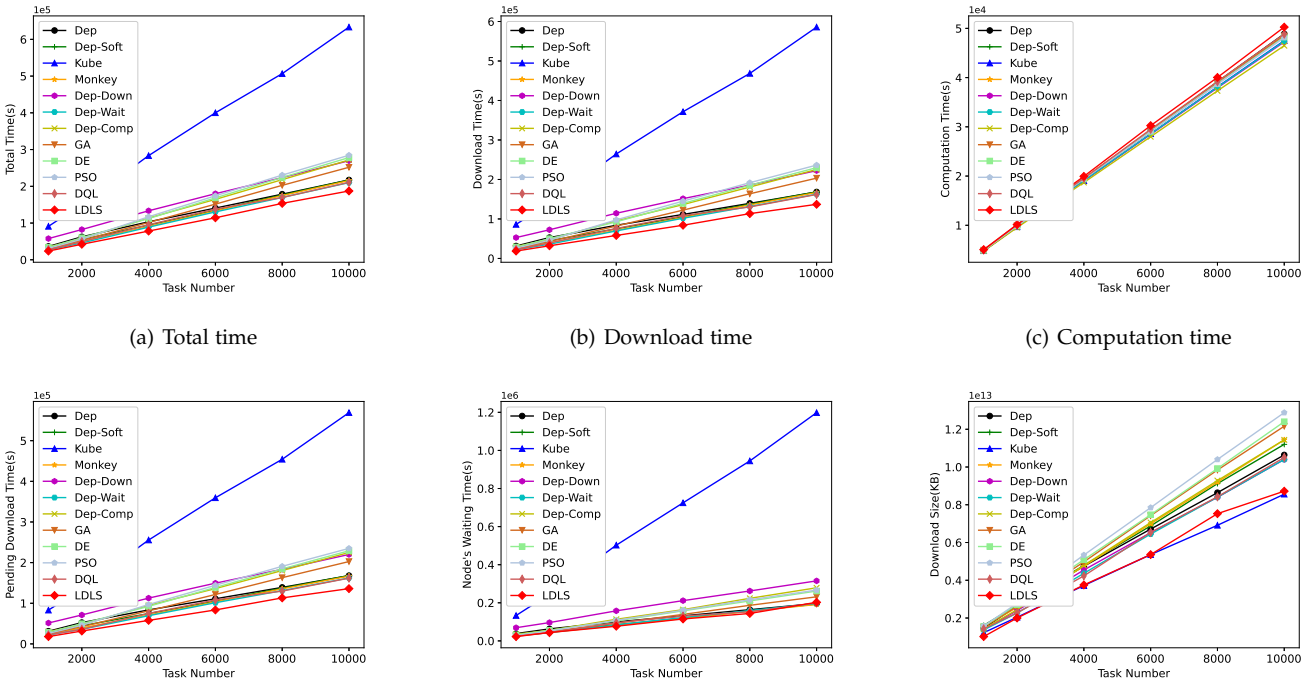
Baselines: To compare the performance, several baselines are conducted. Among these baselines, the Dep and Dep-Soft algorithms are the state-of-the-art layer-based algorithms [7]. The action space of these algorithms is also the set of edge nodes and the remote cloud. If all nodes do not have enough resources, then the task is scheduled to the cloud. The details are as follows.

1) **Dep** [7]: Dep is a layer-based scheduling algorithm. It counts the distribution of all layers required by the requested image on each node to calculate a score based on the size of the existing layers. Then the scheduling decision is made based on the score.
2) **Dep-Soft** [7]: Dep-Soft is a modified version of Dep algorithm. The score is calculated the same as the Dep algorithm. Besides, a threshold is set, and a node is randomly selected among all nodes that exceed the threshold.
3) **Kube** [18]: Kube is an image-based scheduling algorithm. It is one of the default scheduling algorithms of Kubernetes. When scheduling, it counts the distribution of all requested images on each node and calculates a score based on the size of the existing images; then, the schedule is based on the score.
4) **Monkey** [7]: Monkey is a random algorithm; it randomly selects a node for scheduling each time.
5) **Dep-Down**: Dep-Down is modified based on the Dep algorithm. When calculating the score, the layer size is changed to the estimated download time.
6) **Dep-Wait**: Dep-Wait modifies the download time in the Dep-Down algorithm to the waiting time of layers that are downloading.
7) **Dep-Comp**: Dep-Comp modifies the download time in the Dep-Down algorithm to the sum of task computation time and download time.
8) **GA** [61]: Genetic Algorithm (GA) is a meta-heuristic algorithm inspired by the process of natural selection by relying on biologically inspired operators such as mutation, crossover, and selection.
9) **DE** [62]: Differential Evolution (DE) is a meta-heuristic algorithm that optimizes a problem by iteratively trying to improve a candidate solution concerning a given measure of quality.
10) **PSO** [63]: Particle Swarm Optimization (PSO) is a meta-heuristic algorithm. It solves a problem by having a population of candidate solutions (particles) and moving them around in the search space according to a simple mathematical formula over the particle's position and velocity.
11) **DQL** [48]: Deep Q-Learning (DQL) is a value-based RL algorithm that combines deep neural networks and Q-learning to solve high-dimensional state space problems.

Parameter Settings: The node's available storage space is randomly set between 5GB and 15GB, and the available bandwidth is randomly set between 60Mbps and 90Mbps. The CPU frequency is randomly generated between 0.8GHz and 1.2GHz. The default node number is 10, while the node number is also a variable in some experiments. Besides, the bandwidth and CPU frequency of the cloud is set to 100Mbps and 1GHz, respectively. Furthermore, the default task number is 2000. In non-heterogeneous MEC, each node's default bandwidth, CPU frequency, storage space, and maximum container number are set to 70Mbps, 0.9GHz, 15GB, and 10, respectively.

For GA, DE, and PSO algorithms, according to the evaluation, the performance is no longer improved after 300 iterations, so their iteration rounds are all set to 300. The dimensions of the objective function are all 1, the population size is set to 10, and the range of the independent variable is $[0, |N|]$, i.e., the set of all edge nodes and the remote cloud. For the DQL algorithm, two three-layer neural networks are used to extract node features and task features, respectively, and then another two-layer neural network is used to merge the features to output. The learning rate is set to 0.01, the discount parameter is set to 0.9, and the threshold in action selection is set to 0.1 [4], [64].

Simulator: An MEC simulation environment is implemented with Python, which mainly includes the classes of edge node, container, image, layer, task, scheduler, etc. An environment is created based on these classes to return the reward, state, etc. Moreover, the environment is online



(a) Total time (b) Download time (c) Computation time

(d) Pending download time (e) Waiting time (f) Download size

Fig. 5: Performance with different task number in heterogeneous MEC (Uniform distribution)

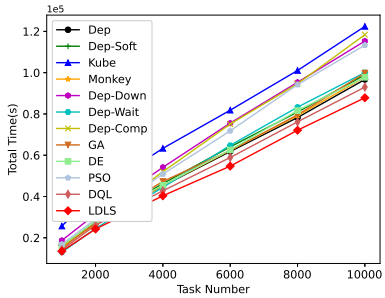


Fig. 6: Performance with different task number in heterogeneous MEC (Zipf distribution)

updated according to the action selected by the agents.

5.2 Experimental Results

To illustrate the proposed LDLS algorithm's performance, the experiments are first conducted in a heterogeneous MEC scenario. Secondly, to verify each resource's impact, more experiments are conducted in the non-heterogeneous MEC scenario. Finally, additional experiments on the LDLS algorithm are supplemented. The experimental results are illustrated in scientific notation, e.g., 1e4 equals 1×10^4.

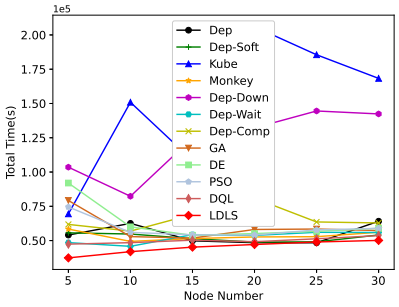
Heterogeneous MEC: The performance of the proposed LDLS algorithm against the baseline algorithms with different task numbers and node numbers in heterogeneous MEC are shown in Figs. 5, 6, and 7, respectively.

Task Number: The total task completion time with different task number is shown in Fig. 5(a). It can be seen that the LDLS algorithm outperforms the baselines. On average, the total time of the LDLS algorithm is reduced by 23%, 19%, 72%, 15%, 41%, 11%, 30%, 23%, 31%, 31%, 14% compared with Dep, Dep-Soft, Kube, Monkey, Dep-Down, Dep-Wait, Dep-Comp, GA, DE, PSO, and DQL algorithms, respectively.

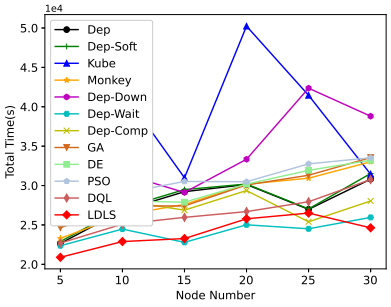
Moreover, the total time is composed of download time and computation time, as shown in Figs. 5(b) and 5(c), respectively. As the number of tasks increases, the LDLS algorithm reduces the download time more evidently. This is because more and more layers need to be downloaded, which leads to an increase in download time, and it accounts for an increasing proportion of the total time. Therefore, the LDLS algorithm chooses to sacrifice the computation time to reduce the download time and obtain better long-term performance.

Furthermore, the download time can be divided into two parts: the time of layers that are being downloaded (downloading time) and the time of layers waiting to be downloaded (pending download time). Since the downloading time is generally less than the pending download time (only if there are no pending download layers, the downloading time is more significant), only the pending download time is illustrated in Fig. 5(d). It can be seen that the LDLS algorithm is more inclined to choose nodes with a shorter pending download time. However, the Kube algorithm cannot make a good decision, which leads to a much longer download time as the number of tasks increases. In addition, comparing Figs. 5(d) and 5(e), it can be found that the pending download time is much smaller than the waiting time of the node, which is an advantage for layer-based algorithms.

Finally, the total download size of each algorithm is

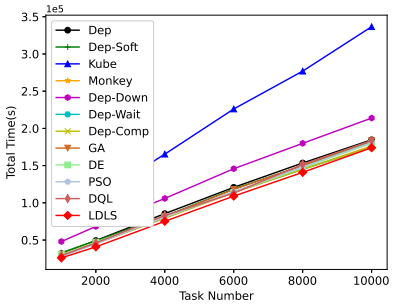


(a) Uniform distribution

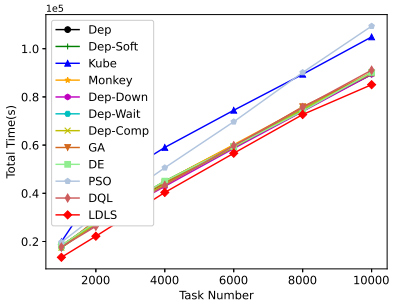


(b) Zipf distribution

Fig. 7: Performance with different node number in heterogeneous MEC



(a) Uniform distribution



(b) Zipf distribution

Fig. 9: Performance with different task number in non-heterogeneous MEC

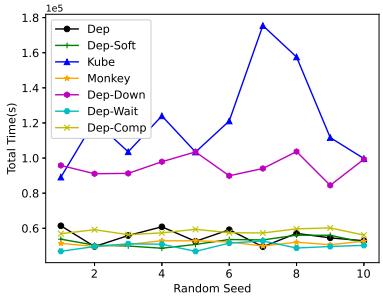


Fig. 8: Performance with different random seed in heterogeneous MEC (Uniform distribution)

shown in Fig. 5(f). The algorithm with a smaller download size does not always perform better. This is because the download time on heterogeneous edge nodes is not only related to download size but also related to bandwidth, which is analyzed in the following experimental results in non-heterogeneous MEC.

The results of the task generated with Zipf are shown in Fig. 6. As the number of tasks increases, the advantage of the total task completion time of the LDLS algorithm becomes more prominent. Compared with the baseline algorithms, LDLS reduces the total task time by up to 36%.

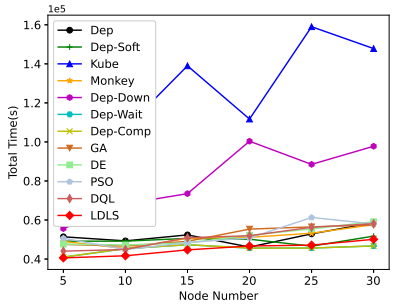
Node Number: As shown in Figs. 7(a) and 7(b), the LDLS algorithm has an advantage regardless of the task generation method. Besides, the performance of the Kube algorithm is unstable with different node numbers. This is because the Kube algorithm is an image-based scheduling algorithm. It cannot judge the distribution of layers, leading to terrible decisions, thereby repeatedly downloading many layers. Further experiments show that the performance of these baseline algorithms has a great relationship with the random seed.

Random Seed: Fig. 8 shows the performance of algorithms with the random seed from 1 to 10 when all other parameter settings remain unchanged. The performance of the Kube algorithm is very different, even if the random seed changes very little. This is because the traversal order in this algorithm is determined according to the random seed. Thus, different traversal orders cause the image-based scheduling algorithm to make different choices, which may cause a lot of repeated layer downloads on different nodes.

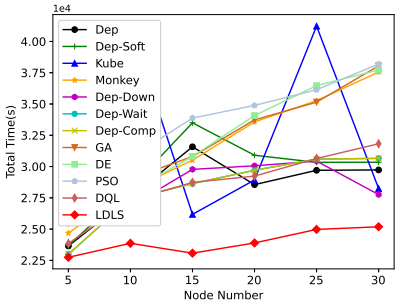
In summary, the LDLS algorithm reduces the total time than the Dep, Dep-Soft, Kube, Monkey, Dep-Down, Dep-Wait, Dep-Comp, GA, DE, PSO, and DQL algorithms in heterogeneous MEC by 17%, 16%, 54%, 13%, 33%, 9%, 26%, 17%, 22%, 26%, and 10% on average, respectively.

Non-heterogeneous MEC: To further compare the impact of some key parameters in MEC, experiments are conducted in non-heterogeneous MEC as shown in Figs. 9 to 12.

Task Number: The performance with different task numbers in non-heterogeneous MEC with uniform and Zipf task distribution is shown in Figs. 9(a) and 9(b), respectively. It can be concluded that the LDLS algorithm's performance is better than baseline algorithms no matter what kind of distribution. Also, the advantage of the LDLS algorithm on tasks generated based on the Zipf distribution is more



(a) Uniform distribution



(b) Zipf distribution

Fig. 10: Performance with different node number in non-heterogeneous MEC

evident. Since it is a non-heterogeneous MEC, the total task completion time increases almost linearly as the number of tasks increases.

Node Number: The results are shown in Figs. 10. In most cases, the LDLS algorithm performs better regardless of the task generation method. As the number of nodes increases, the gap between the LDLS algorithm and the baseline algorithm has narrowed. The reason is that the computation resources increase with larger node numbers; even random algorithms can achieve better results.

It can be concluded that the performance of tasks generated by Zipf is better than uniform. Therefore, in the following experiments, only the performance of tasks with uniform distribution is compared in non-heterogeneous MEC.

Bandwidth: The performance with different bandwidth is shown in Fig. 11. From Fig. 11(a), it can be concluded that the LDLS algorithm performs much better than other algorithms when the bandwidth is small. As shown in Fig. 11(b) and Fig. 11(c), both the download time and computation time of the LDLS algorithm are less than other algorithms when the bandwidth is small. The LDLS algorithm has learned a better strategy for scheduling the tasks to the cloud when bandwidth is insufficient since the download time is relatively long with small bandwidth. Besides, as the bandwidth increases, the LDLS algorithm's performance always remains better than baseline algorithms.

Container Number: The container number determines the maximum number of containers that each node can run simultaneously, which can be regarded as a limitation of computation resources. The larger the container number, the more computation resources. As shown in Fig. 12(a), when the container number becomes more extensive, the Kube algorithm's performance gets worse, while the LDLS algorithm remains stable.

CPU Frequency: The CPU frequency determines the processing speed of the node, thereby affecting the task completion time. As shown in Fig. 12(b), the total task completion time becomes less when the CPU frequency increases. This is because when the CPU frequency becomes larger, the processing speed becomes faster, and the computation time decreases.

Storage Space: Finally, the performance of different storage spaces is shown in Fig. 12(c). When the storage space is relatively limited, the LDLS algorithm chooses to offload tasks to the cloud to reduce the computation time greatly.

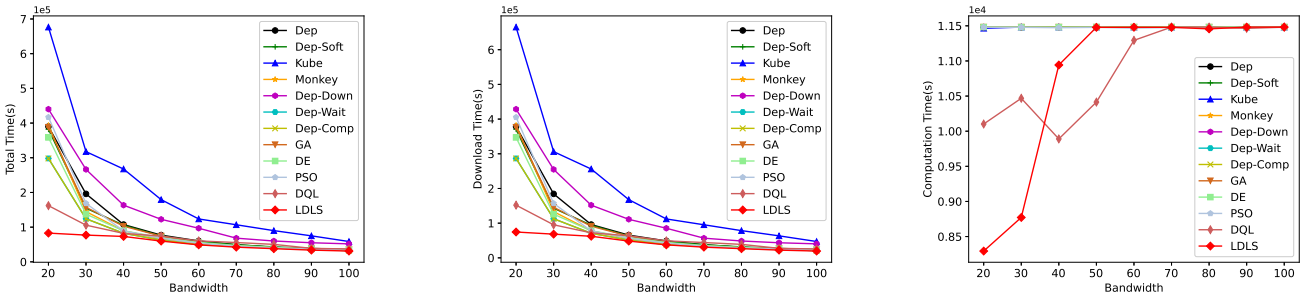
LDLS: To further illustrate the performance of the LDLS algorithm, more experiments are conducted. First, the Cumulative Distribution Function (CDF) of task completion time is shown in Fig. 13. It can be seen that the proportion of tasks with a shorter task completion time of the LDLS algorithm is more than baseline algorithms. Thus the average task completion time of the LDLS algorithm is shorter than baseline algorithms. This proves that the LDLS algorithm can optimize the scheduling decisions from a long-term perspective.

Fig. 14 shows the reward, the loss of the policy network, the loss of the value function, and the probability of action selection of the LDLS algorithm. It can be seen from Fig. 14(a) that the LDLS algorithm converges after several hundred rounds of training, which shows that the algorithm has good convergence. The loss curves in Figs. 14(b) and 14(c) further show the convergence. Fig. 14(d) shows the sum of the probabilities according to the actions selected by the policy network in each epoch. The greater the sum of the probabilities, the more good actions selected by the policy network, i.e., the better the performance of the policy network. It can be seen from Fig. 14(d) that although the loss is no longer reduced after several rounds of the epoch, the probability of selecting a good action becomes greater, i.e., the performance of the policy network is getting better and better over time.

6 Discussion

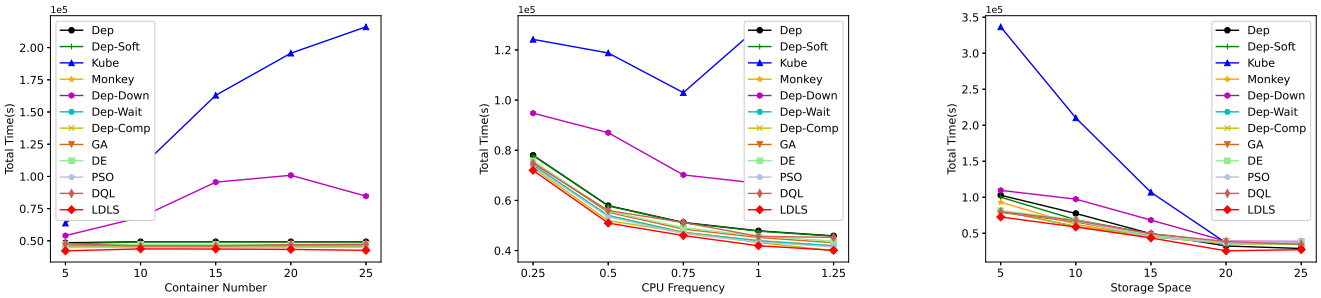
From the experimental results, we can see the effectiveness of our algorithms. The following issues deserved further investigation.

Task transmission cost. In our MEC scenario, the user tasks are first transmitted to the base stations, then scheduled to different edge nodes for processing. The task transmission time mainly includes the time for the user to transmit task data to the base station and the time to transmit the data from the base station to the node. The selection of the user base station is made according to the underlying protocol, which is not within the scope of this paper. Then the time to transmit task data to the base station is considered a constant value for each user task. Besides, edge nodes are generally connected through an intranet with a stable bandwidth [42], [43], so the transmission time between the base station and different nodes can also be considered a



(a) Total time (b) Download time (c) Computation time

Fig. 11: Performance with different bandwidth in non-heterogeneous MEC (Uniform distribution)



(a) Container number (b) CPU frequency (c) Storage space

Fig. 12: Performance with different container number, CPU frequency, and storage space in non-heterogeneous MEC (Uniform distribution)

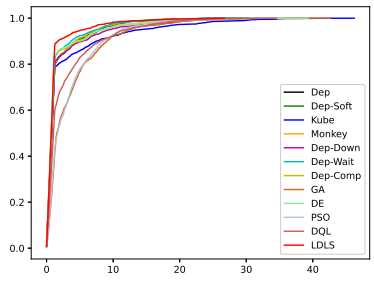


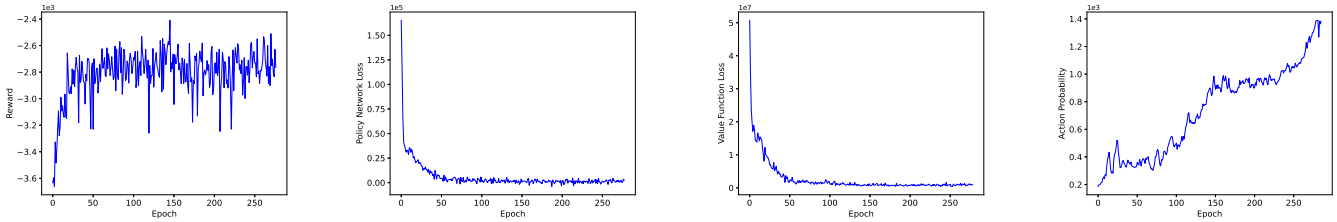
Fig. 13: CDF of the task completion time

small constant value. As a result, the task transmission time is equivalent to a constant value for each user task.

Compared with the download time of the images and the computation time of the tasks, the task transmission time is very short. For example, as shown in TABLE 4, the average size of layers is 414Mb. In comparison, the photo size that needs to be transferred after preprocessing is only about 4 - 8Mb [45], [65]. Assuming that the bandwidth from the task to the base station is only 10Mbps, the transmission time is then about 0.4 - 0.8 seconds. As shown in Fig. 5, in the case of 2000 tasks, the average download time of image-based scheduling (Kube algorithm) is about 70 seconds, while the average download time of the LDLS algorithm is about 16 seconds. Based on these observations, we do not consider the transmission time in the system model and focus on optimizing the download time and computation time [7].

Furthermore, in this paper, we focus on optimizing the task completion time on the node from the granularity of the layer. Considering the task transmission time, then the location of the wireless base station, the wireless channel condition between the user and the base station, the mobility of users, and the handover of user tasks need to be considered. This will make the system model very complex, which is left for our future work.

Layer Sequence. From the perspective of task scheduling, the layers form a set instead of a sequence when extracting layer dependencies. The set of layers can be downloaded in an arbitrary order when downloading an image. In this paper, on each edge node, layers are downloaded in the order requested by the task, and good experimental results have been obtained. However, we can further optimize the layer download sequence. For example, when determining the layer download sequence of multiple images on a single edge node, the layer download sequencing problem is equal to a classic precedence-constrained single machine job scheduling problem $1|prec|\sum t_i$ [66], where 1 means the single machine, $prec$ means jobs have precedence constraints, and $\sum t_i$ denotes that the objective is to minimize the total completion time of all jobs. Jointly optimizing the task scheduling and the layer download sequencing problem in an online manner is challenging and interesting, and we will leave it as future work.



(a) Reward (b) Loss of the policy network (c) Loss of the value function (d) Probability of action selections

Fig. 14: Reward, loss of the policy network, loss of the value function, and probability of action selections of the LDLS algorithm

7 Conclusions

We have formulated the layer dependency-aware scheduling problem in heterogeneous MEC. A policy gradient-based algorithm with well-extracted layer dependency information and improved action selection effectively reduces the task completion time with limited resources. Compared with the state-of-the-art baseline algorithms, the proposed LDLS algorithm outperforms the image-based and layer-based algorithms by 54% and 19%, respectively.

This is our first attempt at layer dependency-aware scheduling. We have obtained inspiring experimental results, proving that scheduling at the granularity of the layer can effectively reduce the task completion time in heterogeneous MEC. On this basis, much work can be done. For example, tasks can be grouped and scheduled together to use the layer dependency information further. Moreover, the download sequence of the layers can be further considered to optimize the waiting time of layer download for tasks. Besides, the registry's deployment on the edge nodes or layer caching with some selected edge nodes is also up-and-coming to further reduce the task completion time. These will be considered in our future work.

Acknowledgments

This work is supported by Guangdong Key Lab of AI and Multi-modal Data Processing, Chinese National Research Fund (NSFC) Project No. 61872239; BNU-UIC Institute of Artificial Intelligence and Future Networks funded by Beijing Normal University (Zhuhai) and AI-DS Research Hub, BNU-HKBU United International College (UIC), Zhuhai, Guangdong, China.

References

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.

[2] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[3] H. Wang and J. Xie, "User preference based energy-aware mobile ar system with edge computing," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1379–1388.

[4] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.

[5] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, 2019.

[6] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2018.

[7] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[8] I. Miell and A. H. Sayers, *Docker in practice*. Manning Publications, 2016.

[9] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig *et al.*, "Improving docker registry design based on production workload analysis," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, 2018, pp. 265–278.

[10] 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.

[11] C. Li, J. Bai, and J. Tang, "Joint optimization of data placement and scheduling for improving user experience in edge computing," *Journal of Parallel and Distributed Computing*, vol. 125, pp. 93–105, 2019.

[12] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. Warke, M. Mohamed, and A. Butt, "Slimmer: Weight loss secrets for docker registries," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 517–519.

[13] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving perfect layers out of docker images," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[14] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 181–195.

[15] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight {OS} containers," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 199–212.

[16] M. Park, K. Bhardwaj, and A. Gavrilovska, "Toward lighter containers for the edge," in *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[17] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[18] T. L. Foundation. Kubernetes. [Online]. Available: https://kubernetes.io

[19] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 373–377.

[20] K. Project. K3s: Lightweight kubernetes. [Online]. Available: https://k3s.io

[21] T. L. Foundation. Akraino. [Online]. Available: https://www.lfedge.org/projects/akraino/

[22] J. Zhang, X. Zhou, T. Ge, X. Wang, and T. Hwang, "Joint task scheduling and containerizing for efficient edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 2086–2100, 2021.

[23] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: a factorization-machine based neural network for ctr prediction," *arXiv preprint arXiv:1703.04247*, 2017.

[24] S. Rendle, "Factorization machines," in *2010 IEEE International Conference on Data Mining*. IEEE, 2010, pp. 995–1000.

[25] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

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

[27] S. Miryoosefi, K. Brantley, H. Daumé III, M. Dudík, and R. Schapire, "Reinforcement learning with convex constraints," *arXiv preprint arXiv:1906.09323*, 2019.

[28] D. Inc. Docker hub. [Online]. Available: https://hub.docker.com/

[29] B. Sonkoly, M. Szabó, B. Németh, A. Majdán, G. Pongrácz, and L. Toka, "Fero: Fast and efficient resource orchestrator for a data plane built on docker and dpdk," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 243–251.

[30] Y. Xu, W. Chen, S. Wang, X. Zhou, and C. Jiang, "Improving utilization and parallelism of hadoop cluster by elastic containers," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 180–188.

[31] L. Lv, Y. Zhang, Y. Li, K. Xu, D. Wang, W. Wang, M. Li, X. Cao, and Q. Liang, "Communication-aware container placement and reassignment in large-scale internet data centers," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 540–555, 2019.

[32] T. Benjaponpitak, M. Karakate, and K. Sripanidkulchai, "Enabling live migration of containerized applications across clouds," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 2529–2538.

[33] A. Tiwari, B. Ramprasad, S. H. Mortazavi, M. Gabel, and E. d. Lara, "Reconfigurable streaming for the mobile edge," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, 2019, pp. 153–158.

[34] L. Civolani, G. Pierre, and P. Bellavista, "Fogdocker: Start container now, fetch image later," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 51–59.

[35] Containerd. Stargz snapshotter. [Online]. Available: https://github.com/containerd/stargz-snapshotter

[36] F. Tang, Y. Zhou, and N. Kato, "Deep reinforcement learning for dynamic uplink/downlink resource allocation in high mobility 5g hetnet," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 12, pp. 2773–2782, 2020.

[37] Y. Qian, R. Wang, J. Wu, B. Tan, and H. Ren, "Reinforcement learning-based optimal computing and caching in mobile edge network," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2343–2355, 2020.

[38] X. Xiong, K. Zheng, L. Lei, and L. Hou, "Resource allocation based on deep reinforcement learning in iot edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1133–1146, 2020.

[39] H. Xu, X. Liu, W. Yu, D. Griffith, and N. Golmie, "Reinforcement learning-based control and networking co-design for industrial internet of things," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 5, pp. 885–898, 2020.

[40] X. Wang, R. Li, C. Wang, X. Li, T. Taleb, and V. C. Leung, "Attention-weighted federated deep reinforcement learning for device-to-device assisted heterogeneous collaborative edge caching," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 1, pp. 154–169, 2020.

[41] M. Li, P. Si, and Y. Zhang, "Delay-tolerant data traffic to software-defined vehicular networks with mobile edge computing in smart city," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 10, pp. 9073–9086, 2018.

[42] A. Cloud. Alibaba cloud container service for kubernetes (ack). [Online]. Available: https://www.alibabacloud.com/product/kubernetes

[43] ——. Create an enhanced edge node pool. [Online]. Available: https://www.alibabacloud.com/help/doc-detail/201459.htm

[44] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[45] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman, "Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture," in *2012 IEEE symposium on computers and communications (ISCC)*. IEEE, 2012, pp. 000 059–000 066.

[46] C. Cicconetti, M. Conti, and A. Passarella, "A decentralized framework for serverless edge computing in the internet of things," *IEEE Transactions on Network and Service Management*, 2020.

[47] Z. Han, H. Tan, G. Chen, R. Wang, Y. Chen, and F. C. Lau, "Dynamic virtual machine management via approximate markov decision process," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2016, pp. 1–9.

[48] 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, pp. 529–533, 2015.

[49] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour *et al.*, "Policy gradient methods for reinforcement learning with function approximation." in *NIPs*, vol. 99. Citeseer, 1999, pp. 1057–1063.

[50] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[51] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[52] Q. Liu, F. Yu, S. Wu, and L. Wang, "A convolutional click prediction model," in *Proceedings of the 24th ACM international on conference on information and knowledge management*, 2015, pp. 1743–1746.

[53] Y. Zhang, H. Dai, C. Xu, J. Feng, T. Wang, J. Bian, B. Wang, and T.-Y. Liu, "Sequential click prediction for sponsored search with recurrent neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, no. 1, 2014.

[54] Q. Liu, T. Xia, L. Cheng, M. Van Eijk, T. Ozcelebi, and Y. Mao, "Deep reinforcement learning for load-balancing aware network control in iot edge systems," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[55] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 242–253, 2020.

[56] T. Contributors. Embedding. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html

[57] Y. Sun, J. Pan, A. Zhang, and A. Flores, "Fm2: Field-matrixed factorization machines for recommender systems," in *Proceedings of the Web Conference 2021*, 2021, pp. 2828–2837.

[58] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[59] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[60] C. Wang, C. Liang, F. R. Yu, Q. Chen, and L. Tang, "Computation offloading and resource allocation in wireless cellular networks with mobile edge computing," *IEEE Transactions on Wireless Communications*, vol. 16, no. 8, pp. 4924–4938, 2017.

[61] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[62] K. V. Price, "Differential evolution," in *Handbook of optimization*. Springer, 2013, pp. 187–214.

[63] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4. IEEE, 1995, pp. 1942–1948.

[64] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning." in *AAAI Conference on Artificial Intelligence (AAAI)*, 2016, pp. 2094–2100.

[65] B. Benjdira, T. Khursheed, A. Koubaa, A. Ammar, and K. Ouni, "Car detection using unmanned aerial vehicles: Comparison between faster r-cnn and yolov3," in *2019 1st International Conference on Unmanned Vehicle Systems-Oman (UVS)*. IEEE, 2019, pp. 1–6.

[66] G. J. Woeginger, "On the approximability of average completion time scheduling under precedence constraints," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2001, pp. 887–897.

Zhiqing Tang received the B.S. degree from School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015 and is currently a Ph.D. candidate in Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include mobile edge computing, resource allocation, and reinforcement learning.

Jiong Lou received the B.S. degree from Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2016 and is currently a Ph.D. candidate in Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include mobile edge computing, resource allocation, and reinforcement learning.

Professor Weijia Jia is currently a Chair Professor, Director of BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (Zhuhai) and VP for Research of BNU-HKBU United International College (UIC) and has been the Zhiyuan Chair Professor of Shanghai Jiao Tong University, China. He was the Chair Professor and the Deputy Director of State Kay Laboratory of Internet of Things for Smart City at the University of Macau. He received BSc/MSc from Center South University, China, in 82/84 and Master of Applied Sci./PhD from Polytechnic Faculty of Mons, Belgium in 92/93, respectively, all in computer science. From 93-95, he joined German National Research Center for Information Science (GMD) in Bonn (St. Augustine) as a research fellow. From 95-13, he worked at City University of Hong Kong as a professor. His contributions have been recognized as optimal network routing and deployment, anycast and QoS routing, sensors networking, AI (knowledge relation extractions; NLP, etc.), and edge computing. He has over 600 publications in the prestige international journals/conferences and research books, and book chapters. He has received the best product awards from the International Science & Tech. Expo (Shenzhen) in 2011/2012 and the 1st Prize of Scientific Research Awards from the Ministry of Education of China in 2017 (list 2). He has served as area editor for various prestige international journals, chair and PC member/skeynote speaker for many top international conferences. He is the Fellow of IEEE and the Distinguished Member of CCF.