Learning Scheduling Bursty Requests in Mobile Edge Computing Using DeepLoad

Ning Chen[a], Sheng Zhang[a,*], Jie Wu[b], Zhuzhong Qian[a], Sanglu Lu[a]

[a]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*

[b]*Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA*

Abstract

The emergence of Mobile Edge Computing (MEC) alleviates the large transmission latency resulting from the traditional cloud computing. For the compute-intensive requests such as video analysis, mobile users prefer to obtain a desired quality of experience (QoE) with neglected latency and reduced energy consumption. The popularity of smart devices allows users to release a run of compute-intensive as well as latency-sensitive requests anywhere, which may lead to bursty requests. A single resource-constrained edge server nearby is capable of handling a small amount of requests quickly, yet it seems helpless when encountering bursty compute-intensive requests. Despite the abundance of recently proposed schemes, the majority focus on efficiently scheduling pending requests in a single edge server, and ignored the potential role of edge collaboration to schedule bursty requests. Besides, while some recent studies proposed to finish a task using multiple devices, they focused on collaboration between mobile devices rather than between edge servers. Hence, we propose DeepLoad, a S2S system that schedules the bursty requests with a collaborative method using reinforcement learning (RL). DeepLoad decouples the scheduling decision into *AP selection* for setting the access point and *workload redistribution* for collaborative servers. DeepLoad trains a neural network model that picks decisions for each re-

[*]Corresponding author. Tel.: +86 25 83681369

Email addresses: `ningc@smail.nju.edu.cn` (Ning Chen), `sheng@nju.edu.cn` (Sheng Zhang), `jiewu@temple.edu` (Jie Wu), `qzz@nju.edu.cn` (Zhuzhong Qian), `sanglu@nju.edu.cn` (Sanglu Lu)

quest based on observations collected by mobile devices. DeepLoad learns to make scheduling decisions solely through the resulting performance of historical decisions rather than rely on pre-programmed models or specific assumptions for the environment. Naturally, DeepLoad automatically learns the scheduling algorithm for each request and obtains a gratifying QoE. We aim to maximize the fraction of requests finished before their attached deadlines. Based on the Shanghai taxi trajectory data set, we design a simulator to obtain abundant samples, and leverage two GeForce GTX TITAN Xp GPUs to train the Actor-Critic network. Compared to the state-of-the-art bandwidth-based and server resources-based methods, DeepLoad can achieve a significant improvement in average fraction.

Keywords: bursty requests, edge collaboration, deep reinforcement learning

1. Introduction

The rise of 5G has greatly strengthened the connection between humans and machines. Meanwhile, compute-intensive applications such as interactive gaming, image/video processing, augmented/virtual reality and face recognition, are becoming popular on mobile devices, and these applications pursue low delay and low energy consumption. With the emergence of the Mobile Edge Computing (MEC) paradigm [1–4], the data and computation are pushed away from centralized cloud computing infrastructures to the logical edge of a network, thereby enabling analytics and knowledge generation to occur closer to the mobile users, which mitigates the problem of high energy consumption, limited capability of mobile devices and the unexpected WAN latency.

In the current MEC paradigm, an edge cloud covers a huge service area, in which the users can send diverse requests to this single edge cloud for processing. Generally, a single edge cloud is sufficient to handle a small amount of requests quickly. However, we observed that more and more requests are compute-intensive and bursty, and thus cannot be efficiently handled by a single resource-constrained edge cloud. Taking the multi-player Virtual Reality (VR) game as an example, such as From Other Suns, or Seeking Dawn. Generally, VR [5, 6] has stringent performance requirements including a *fps* (i.e. frames per second) above 60, and a *motion-to-photon* latency below $20ms$, and yet the compute-intensive rendering processing becomes the key obstacle to satisfy such stringent requirements. What's more, during the interaction of VR, many players are likely to release rendering requests simultaneously, which inevitably causes the bursty arrival of compute-intensive requests. For an edge cloud, it not only needs to render the foreground and background for each action, but also needs to synchronize the entire VR scene by sending real-time images to each user, which results in unpredictable computational and communication costs. In such a dilemma[1], a single resource-constrained edge cloud may seem helpless.

Existing works have fallen short of handling bursty request in edge computing. Most of them [7–10] focused on offloading bursty requests to either of a local device, an edge server, or the remote cloud data center. Studies for edge collaboration highlight its advantages in D2D (device to device). For example, Chen et al. [11] proposed a D2D framework to achieve energy-efficient collaborative task execution at the network-edge for mobile users. Wu et al. [12] proposed the SVC scheme to achieve a flexible video distribution. Guo et al. [13] proposed edge-cloud collaborative computation offloading, but ignored S2S (edge server to edge server). Next, we will put forward the S2S framework and present the novel algorithm to schedule bursty requests.

In this paper, we consider a general edge scenario of bursty requests. In a local area network (LAN) environment, the densely-distributed access points (APs), which refer to wireless access points, small cell base stations and other hardware that can receive and forward requests from end devices, are connected with each other through back-haul links via X2 interface or S1 interface of Core Network. In light of tremendous transmission, the S1 interface is probably the best option. Practically, the edge servers configured with a limited number of services are deployed at APs, so that mobile users can offload requests attached with a deadline to an AP nearby. We will use AP and edge server interchangeably in the sequel of the paper, since an edge server is often attached with an AP. How to make an efficient scheduling when encountering numerous newly released requests is a common but not well-solved problem. What's worse, edge servers maintain long queues in which considerable workloads wait in line to be processed, and the network is degraded to an inferior state. To address it using collaboration, two fundamental problems must be taken into consideration as follows:

[1]We can offload partial frames to adjacent servers for rendering may mitigate the expensive cost of single edge cloud. Of course we ought to eliminate the correlations of inter-frames and guarantee the playback order of recombined VR video.

▷ **AP selection.** Generally, each AP covers a specific service area, and multiple APs are accessible to a user concurrently. When a user releases a request, the first step is to select an optimal AP in terms of available network bandwidth and server resources. Picking the AP in a random manner may lead to network degradation and server overload.

▷ **Workload redistribution.** Since the AP is selected, the user offloads the newly released request to the edge server co-located with the selected AP. We estimate the completion time of handling this request only in this AP. If the valuation is bigger than the attached deadline, we fall back on the selected AP's adjacent servers, and offload partial workload to them. However, it is challenging to determine the optimal fraction of workload for each AP, given the unpredictable future workloads.

The features of a heterogeneous edge environment, such as the mobility of users and the variability of network bandwidth, leads to intractable decisions for the above two problems. Additionally, the *workload redistribution* relies on the outcome of *AP selection*. In this paper, we pursue a black-box approach for scheduling bursty requests that embraces inference while not relying on detailed analytical performance modeling. Motivated by the recent inspiring achievement of deep reinforcement learning (DRL)[14–16] that shows efficient decision-making in dynamic environments, we propose *DeepLoad*, an intelligent edge-based scheduler that is customized for bursty requests and makes coordinated decisions. Given the known network and server information, the mobile user can make a quick scheduling decision for each request, which includes the selection of the first access point and the percentage of workloads being redistributed among the neighbors of the first AP. DeepLoad learns a policy purely based on the known information, without foreseeing the future.

DeepLoad depicts its policy as a neural network that maps "raw" observations (e.g., workload of request, available link bandwidth and server resources) to the scheduling decision. The neural network incorporates a rich diversity of observations into the scheduling policy in a scalable and expressive way. During training, DeepLoad starts knowing nothing and gradually learns to make better scheduling decisions through reinforcement, in the form of reward signals that reflect user QoE (the request completion time) for past decisions. DeepLoad leverages a state-of-the-art asynchronous advantage actor-critic network model (A3C)[15] to train the policy network, which takes the edge network situation, server statuses and request features as inputs and selects an optimal action through the output (e.g., action distribution). To obtain abundant samples and evaluate DeepLoad's performance, we design a BRS simulator using trace-driven based on the Shanghai taxi trajectory data set, and train the actor-critic network with numerous episodes. Finally, we set several control experiments, and the results demonstrate the superiority of DeepLoad compared with state-of-the-art strategies.

We summarize our contributions here as follows.

▷ We consider a general but tricky scenario, where bursty requests attached with personalized deadlines are released simultaneously without prior notice, and then we formulate it as a long-term optimization problem, which aims to maximize the number of requests whose completion times are superior to the deadlines.

▷ We propose DeepLoad, an intelligent DRL-based scheduler for bursty requests. Given the known information about edge network situation, server statuses and request features, DeepLoad is capable of scheduling each request efficiently. What's more, we provide the specific definition of state, action and reward in DeepLoad, which enables the RL-agents of DeepLoad to train its actor-critic network continuously.

The remainder of this paper is organized as follows. Section 2 introduces a motivation example. Section 3 describes our system model followed by problem formulation. Section 4 details our design of DRL-based algorithm. Section 5 comprehensively evaluates the performance of DeepLoad with several control experiments. We review some related work in Section 6 with the conclusion in Section 7.

2. Motivation

In this section, we analyze an inspiring example to better illustrate the main idea of this paper.

We consider a LAN environment, where all APs are connected by backhaul links that can be used for inter-AP communications, edge servers configured with several specific services are deployed at APs, and each user is within the service area of multiple APs. Each user independently generates compute-intensive requests at the beginning of each time slot. Now, we focus on a particular user u that can connect to AP_1 or AP_2 directly as shown in Fig. 1. At this time point, u releases a new request of type R with a total workload of 96, an input size of 300 and a deadline of 48. Each edge server deployed at an AP maintains multiple FCFS queues that contain numerous pending workloads waiting to be processed as Tab. 1 lists. In this example,

Table 1: Current states of edge server.

Info. \ Servers	E_1	E_2	E_3	E_4	E_5	E_6
Available cores	4	3	8	4	4	6
Pending workload	80	60	120	40	60	120

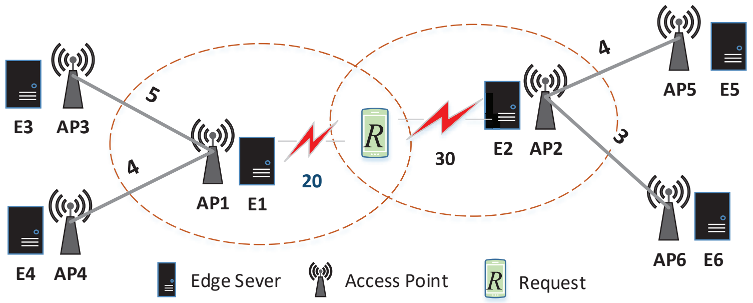
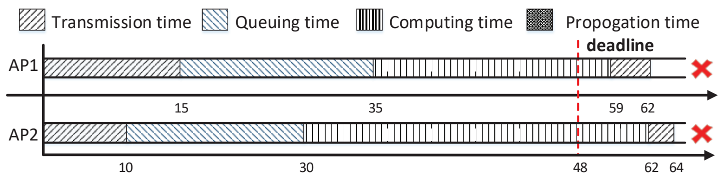


Figure 1: A motivation example. Servers are deployed at APs that are connected with each other through back-haul link. The bandwidth of uplink and downlink, and propagation delays of inter-APs are marked.

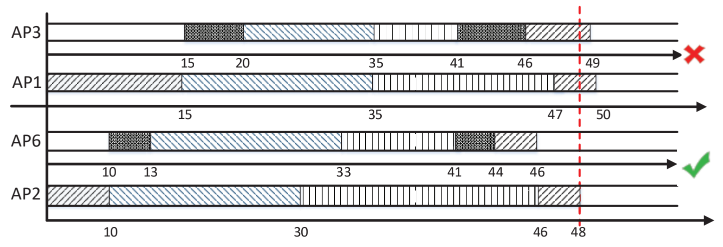
since the type of request generated by u is R, we only consider the pending workload of type R on each server. Similarly, we only list the number of cores assigned to the service of type R at each server. The number besides a link between user u and an AP is the bandwidth. As we know, the back-haul links between APs have much higher bandwidth than access links between users and APs. Therefore, we only consider the propagation delay between APs. Hence, the number besides a link between two APs is the propagation delay. Fig. 2 shows three different offloading strategies and their completion times.

Fig. 2a illustrates the traditional scheme. Mobile user u first selects either AP_1 or AP_2 to connect, and then sends request R to the server co-located with the selected AP. From release to completion, request R goes through four diverse delays, including data upload delay t_1, queuing delay t_2, computing delay t_3, and the resulting download delay t_4. Without loss of generality, we assume the size of the results is probably one fifth of the input size. For instance, if AP_1 is selected, we have $t_1 = \frac{300}{20}$, $t_2 = \frac{80}{4}$, $t_3 = \frac{96}{4}$, and $t_4 = \frac{0.2\times300}{20}$, leading to a completion time of 62, which is larger than the deadline 48. Unfortunately, the deadline still cannot be met if we switch to AP_2.

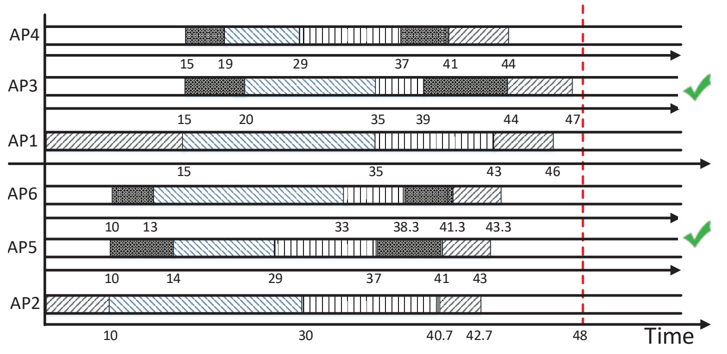
In practice, we redistribute the workloads of R, and offload partial compu-



(a) Traditional strategy. $u \to AP_1$ or $u \to AP_2$.



(b) Collaborative strategy with one neighbor. $u \to AP_1 \to \{AP_3\}$ or $u \to AP_2 \to \{AP_6\}$.



(c) Collaborative strategy with two neighbors. $u \to AP_1 \to \{AP_3,\ AP_4\}$ or $u \to AP_2 \to \{AP_5,\ AP_6\}$.

Figure 2: Comparisons of diverse scheduling strategies.

tation to AP_1's adjacent servers (i.e., AP_3 and AP_4 in this example). Thanks to the small propagation delay of inter-APs, this method may enable this request to get a desired completion time.

Fig. 2b shows the single collaboration strategy, in which the first selected AP (e.g., AP_1) sends half of the workload to one of its neighboring servers (e.g., AP_3), and the workflow is $u \to AP_1 \to \{AP_3\}$ in the top half of Fig. 2b. Similarly, we can calculate the completion times to be 50 and 48 in the top and bottom half of Fig. 2b, respectively. Fig. 2c shows the double collaboration strategy, in which the first selected AP (e.g., AP_1) sends one third of the workload to each neighbor (i.e., AP_3 and AP_4), and the workflow is

$u \rightarrow AP_1 \rightarrow \{AP_3,\ AP_4\}$ in the top half of Fig. 2c. The completion times are 47 and 43.3 in the top and bottom half of Fig. 2c, respectively. As a result, the completion time has a significant improvement over Fig. 2b, which motivates us to utilize edge collaboration and carefully redistribute the workload.

Note that the example is a simple offline scenario that doesn't consider the unpredictable bursty requests. When users anywhere release requests independently, the offloading decision made by each individual user may collide. For example, at the current time t, from the perspective of user u, it seems that offloading the workloads of the request to AP_2 and AP_6 in Fig. 2b could finish the request before deadline; however, some other users may also decide to offload its workloads to AP_2 or AP_6, which is unknown to u. Such an online and bursty, yet realistic, setting makes traditional offline decisions sub-optimal. These challenges together motivate us to make efficient scheduling in a learning method without priori knowledge of future.

3. System Model And Problem Formulation

3.1. System Model

Fig. 3 shows the local area network (LAN) we consider in this paper. Suppose that there are $|\mathcal{U}|$ users, N APs, $|\mathcal{M}|$ types of services in total, $\mathcal{U} = \left\{u_1, u_2, \cdots, u_{|\mathcal{U}|}\right\}$, $\mathcal{AP} = \{AP_1, AP_2, \cdots, AP_N\}$ and $\mathcal{M} = \left\{m_1, m_2, \cdots, m_{|\mathcal{M}|}\right\}$, respectively. The bandwidths of the uplink and downlink between u_i and AP_j are $r_{up}^{(ij)}$ and $r_{down}^{(ij)}$, respectively. As we know, the back-haul links between APs have much higher bandwidth than access links between users and APs, thus, we only consider the propagation delay between APs. We denote by $l_{(j,k)}$ the propagation delay between AP_j and AP_k. In addition, we define a function $\mathcal{N}(x)$ to denote the set of APs, each of which is directly accessible from x, and x is referred to as a user or an AP.

Main notations are summarized in Tab. 2.

3.1.1. Request Model

We divide the time of interest into multiple slots of equal length, $\mathcal{T} = \left\{t_1, t_2, \cdots, t_{|\mathcal{T}|}\right\}$. Each user releases at most one request at any time slot. We use R_i^t to represent the type of request released by user u_i at slot t. Without causing any confusion, we set $\mathcal{R}_i^t$ to 0 if u_i doesn't release any request at slot t. Let ddl_i^t denote the deadline of R_i^t.

In fact, the requests released by mobile users can be viewed as specific jobs, which are processed through the corresponding services installed in

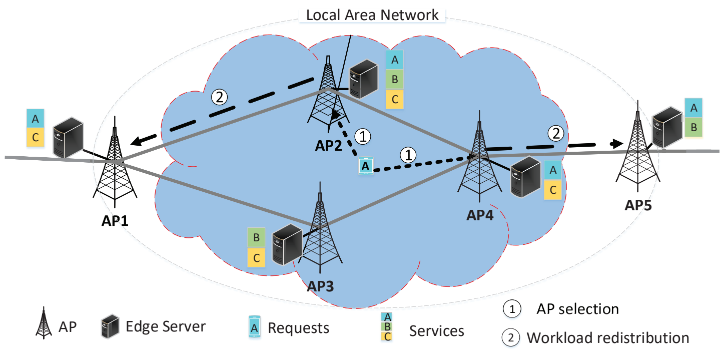


Figure 3: In a LAN environment, APs are connected by back-haul links; each edge server is configured with a limited number of services.

the edge servers. Based on [17, 18], a request can be divided into multiple mutually independent tasks in a fine-grained manner, and each task can be executed independently in an edge server that configured with the corresponding service for that type of request. For example, a real-time video analytics request needs to analyze real-time images from different cameras. Since the pictures taken by different cameras are independent of each other, we can divide this job into multiple independent tasks, and each task analyzes the pictures from the same camera.

We measure the workload of a request based on the scale of its input data. We denote the bytes of the input data of R_i^t as B_i^t. Without loss of generality, denote by ω (in CPU cycles per byte) the number of clock cycles a microprocessor would perform per byte of data; the specific value of ω usually depends on the nature of the request, e.g., the time and space complexity. Then, the workload of R_i^t, denoted by W_i^t, is ωB_i^t.

3.1.2. Edge Server Model

Generally, the edge servers are deployed at APs, and each AP can accommodate up to one edge server. The edge servers are in charge of managing the resources and virtualizing the resources by means of VM or Docker. Each edge server has limited storage and computing capability, and in this paper we pay more attention to the computing capability, which is measured by the number of cores, and each core has equal processing power f (in cycles per second). An edge server is usually resource-constrained, thus it can only be configured with a limited number of services. We use S_j^i to indicate whether AP_j has service m_i and use c_j^i to represent the number of cores assigned to service m_i at server AP_j. Therefore, for an edge server AP_j with C_j cores, we have $\sum_{m_i \in \mathcal{M}} S_j^i c_j^i \leq C_j$. Note that, only if the edge server is configured with the service corresponding to the request, can the request be executed in this edge server.

In this paper, each edge server can handle different types of requests simultaneously. For the same type of requests, an edge server maintains a request queue, and processes these requests based on a first-come, first-served (FCFS) basis. Denote by $Q_{(j,i)}^t$ the FCFS queue for service m_i on edge server AP_j at time slot t.

3.2. Problem Formulation

For any request released by a mobile user, its execution goes through two stages at most, namely *AP selection* and *workload redistribution*. When the user's personalized deadline is satisfied in the first stage, the second stage is not needed. Based on the current information about the network, the edge servers, and request features, the user selects an AP from its neighboring APs for offloading. Then the user uploads the input data to the AP, which first checks whether the deadline can be satisfied if the total workloads are processed by itself. If the estimated delay is no larger than the deadline, then the AP processes all the workloads of this request. Otherwise, the AP performs the *workload redistribution* stage. For example, as Fig. 3 shows, a request A released by some user selects either AP_2 or AP_4 in the AP selection stage, then if its deadline cannot be satisfied, the selected AP would further redistribute the workloads of A among the neighbors of the AP. For a better understanding of *AP selection* and *workload redistribution* in the edge collaboration framework, we start from analyzing its offline scenario with the known network conditions and resource situations of the edge servers. Now, we model the *AP selection* and *workload distribution* in detail.

AP Selection. For request R_i^t produced by user u_i at slot t, u_i can only offload part of its workload to the APs that (1) can be directly accessed by it, and (2) are configured with the corresponding service $m_{R_i^t}$. More formally, denote by AP_i^t the set of such APs, i.e.,

$$\mathcal{AP}_i^t = \mathcal{N}(u_i) \cap \left\{ AP_j | S_j^{R_i^t} = 1 \right\}. \tag{1}$$

We use a binary variable $X_{(i,j)}^t$ to denote whether u_i selects $AP_j \in \mathcal{AP}_i^t$ for offloading at time slot t, where $X_{(i,j)}^t = 1$ (resp. 0) indicates that AP_j is (resp. is not) selected. We have $\sum_{AP_j \in \mathcal{AP}_i^t} X_{(i,j)}^t = 1$. If no workload redistribution is performed, the completion time T_i^t of R_i^t consists of four parts: the upload

Table 2: Main notations used in this paper.

Notation	Meaning
r_{dl}, r_{ul}	the available bandwidth of uplink and downlink
$\mathcal{AP}$	the set of APs (edge servers)
$\mathcal{U}$	the set of mobile users
$\mathcal{M}$	the set of services
$\mathcal{T}$	the set of time slots
$\|\mathcal{S}\|$	the cardinality of set $\mathcal{S}$
$l_{(j,k)}$	the propagation delay between AP_j and AP_k
$\mathcal{N}(x)$	the set of direct neighbors of x
R_i^t	the type of request R_i released by user U_i at slot t
ddl_i^t	the deadline of R_i^t
ω	# of clock cycles a microprocessor performs per byte
B_i^t	the total bytes of the input data of R_i^t
W_i^t	the total workloads of of R_i^t
S_j^i	indicates whether AP_j has service m_i
c_j^i	# of cores assigned to service m_i in AP_j
C_j	# of cores in AP_j
$Q_{(j,i)}^t$	the FCFS queue for service m_i on AP_j at slot t
T_i^t	the completion time of R_i^t
$X_{(i,j)}^t$	indicates whether u_i selects $AP_j \in \mathcal{AP}_i^t$ for offloading at time slot t
$Y_{(i,j)}^t$	% of the workloads of R_i^t processed at AP_j
$Y_{(i,j,k)}^t$	% of the workloads of R_i^t processed at $AP_k \in \mathcal{AP}_i^t$

time $T_{up}^{(i,j,t)}$, the queuing time $T_{queue}^{(i,j,t)}$, the processing time $T_{proc}^{(i,j,t)}$, and the download time $T_{down}^{(i,j,t)}$. Denote by B_i^t and $B_i^{t\prime}$ the sizes of input data and results of R_i^t, respectively. Let $r_{ul}^{(ij)}$ and $r_{dl}^{(ij)}$ be the link bandwidth of uplink and downlink, respectively. Then, we have

$$T_{up}^{(i,j,t)} = \sum\nolimits_{AP_j \in \mathcal{AP}_i^t} X_{(i,j)}^t \frac{B_i^t}{r_{ul}^{(ij)}}, \tag{2}$$

$$T_{queue}^{(i,j,t)} = \sum\nolimits_{AP_j \in \mathcal{AP}_i^t} \sum_{q \in Q_{(j,R_i^t)}^t} X_{(i,j)}^t \frac{W_q}{c_j^{R_i^t} f}, \tag{3}$$

$$T_{proc}^{(i,j,t)} = \sum\nolimits_{AP_j \in \mathcal{AP}_i^t} X_{(i,j)}^t \frac{W_i^t}{c_j^{R_i^t} f}, \tag{4}$$

$$T_{down}^{(i,j,t)} = \sum\nolimits_{AP_j \in \mathcal{AP}_i^t} X_{(i,j)}^t \frac{B_i^{t\prime}}{r_{dl}^{(ij)}}, \tag{5}$$

where $Q_{(j,R_i^t)}^t$ is the FCFS queue for service R_i^t in edge server AP_j, $c_j^{R_i^t}$ represents the number of cores assigned to service R_i^t in AP_j, and $W_i^t = \omega B_i^t$ is the total workload of R_i^t. If $t + T_{up}^{(i,j,t)} + T_{queue}^{(i,j,t)} + T_{proc}^{(i,j,t)} + T_{down}^{(i,j,t)} \leq ddl_i^t$, i.e., R_i^t can be finished before its deadline, then no workload redistribution is needed. Otherwise, we have to make workload redistribution.

Workload Redistribution. We redistribute the workload of R_i^t among the direct neighboring APs of the selected AP in *AP Selection*. For each $AP_j \in \mathcal{AP}_i^t$, we define the set of its direct neighboring APs, each of which is configured with the corresponding service $m_{R_i^t}$, as follows:

$$\mathcal{AP}_{(i,j)}^t = \mathcal{N}(AP_j) \cap \left\{ AP_k | S_k^{R_i^t} = 1 \right\}. \tag{6}$$

Without loss of generality, we use $Y_{(i,j)}^t$ and $Y_{(i,j,k)}^t$ to represent the percentage of workload to be processed at AP_j and AP_k, $\forall AP_k \in \mathcal{AP}_{(i,j)}^t$. Therefore, the selected AP_j leaves $Y_{(i,j)}^t W_i^t$ units of workload for its local processing, and sends $Y_{(i,j,k)}^t B_i^t$ units of input for each $AP_k \in \mathcal{AP}_{(i,j)}^t$ simultaneously, after which AP_k puts this workload at the end of its local queue $Q_{(k,R_i^t)}^t$ and processes the workloads in the queue in a FCFS manner. Therefore, for this part of workload of R_i^t that is sent to neighbor servers, it experiences another four phases: being uploaded from AP_j to AP_k, waiting in the queue of service R_i^t in AP_k, being processed by AP_k, and being downloaded from AP_k to AP_j. The time consumed by these four phases are $T_{up}^{(i,j,k,t)}$, $T_{queue}^{(i,j,k,t)}$, $T_{proc}^{(i,j,k,t)}$, and $T_{down}^{(i,j,k,t)}$, respectively. Note that, we use $l_{(i,j)}$ to denote the propagation delay between AP_i and AP_j. Therefore,

$$T_{up}^{(i,j,k,t)} = T_{down}^{(i,j,k,t)} = l_{(j,k)}, \tag{7}$$

$$T_{queue}^{(i,j,k,t)} = \sum\nolimits_{AP_j \in \mathcal{AP}_i^t} \sum\nolimits_{q \in Q_{(k,R_i^t)}^t} X_{(i,j)}^t \frac{W_q}{c_k^{R_i^t} f}, \tag{8}$$

$$T_{proc}^{(i,j,k,t)} = \sum\nolimits_{AP_j \in \mathcal{AP}_i^t} X_{(i,j)}^t \frac{Y_{(i,j,k)}^t W_i^t}{c_k^{R_i^t} f}. \tag{9}$$

Denote by $T_{j\to k}^{(i,t)}$ the time consumption of redistributing $Y_{(i,j,k)}^{t}W_i^t$ units of workload from AP_j to AP_k. Thus,

$$T_{j\to k}^{(i,t)} = T_{up}^{(i,j,k,t)} + T_{queue}^{(i,j,k,t)} + T_{proc}^{(i,j,k,t)} + T_{down}^{(i,j,k,t)}. \tag{10}$$

For AP_j itself, it has to process $Y_{(i,j)}^{t}W_i^t$ units of workload. Denote by $T_{j\to j}^{(i,t)}$ the time consumption of such local processing. It is easy to see,

$$T_{j\to j}^{(i,t)} = \sum_{AP_j\in\mathcal{AP}_i^t} X_{(i,j)}^t \left(\sum_{q\in Q_{(j,R_i^t)}^t} \frac{W_q}{c_k^{R_i^t} f} + \frac{Y_{(i,j)}^t W_i^t}{c_k^{R_i^t} f} \right). \tag{11}$$

Formulation. Based on the above modeling, we now can define the Completion Time T_i^t of request R_i^t. A request is completed if all of its workloads have been processed. If $t + T_{up}^{(i,j,t)} + T_{queue}^{(i,j,t)} + T_{proc}^{(i,j,t)} + T_{down}^{(i,j,t)} \le ddl_i^t$,

$$T_i^t = T_{up}^{(i,j,t)} + T_{queue}^{(i,j,t)} + T_{proc}^{(i,j,t)} + T_{down}^{(i,j,t)}; \tag{12}$$

otherwise,

$$T_i^t = T_{up}^{(i,j,t)} + \max\left\{ T_{j\to j}^{(i,t)}, \max_{AP_k\in\mathcal{AP}_{(i,j)}^t} \left\{ T_{j\to k}^{(i,t)} \right\} \right\} + T_{down}^{(i,j,t)}. \tag{13}$$

We use a binary variable $Z_{(i,j)}^t$ to denote whether the Completion Time T_i^t of request R_i^t is smaller than the ddl_i^t, where $Z_{(i,j)}^t = 1$ (resp. 0) indicates that T_i^t is smaller (resp. is bigger) than the ddl_i^t.

To sum up, we are committed to maximizing the number of requests that finish before deadlines, thus we get the following optimization problem ($\mathcal{P}$):

$$\max_{t\in\mathcal{T}, i\in|\mathcal{U}|} \quad \sum_{t\in\mathcal{T}} Z_{(i,j)}^t \tag{14}$$

$$s.t. \sum_{AP_j\in\mathcal{AP}_i^t} X_{(i,j)}^t = 1, \forall i \in |\mathcal{U}|, \forall t \in \mathcal{T}, \tag{15}$$

$$\sum_{AP_j\in\mathcal{AP}_i^t} X_{(i,j)}^t [\sum_{AP_k\in\mathcal{AP}_{(i,j)}^t} Y_{(i,j,k)}^t + Y_{(i,j)}^t] = 1, \tag{16}$$

$$\sum_{m_i\in\mathcal{M}} c_j^i \le C_j, , \forall AP_j \in \mathcal{AP}, \tag{17}$$

$$X_{(i,j)}^t \in \{0,1\}, \; Y_{(i,j,k)}^t \in [0,1], \tag{18}$$

Eq. (15) ensures that the user can only connect to one AP at one time slot, Eq. (16) implies the total percentage assigned to the current server and its neighboring servers is 1, Eq. (17) is the resource limitation of each server.

Considering that the target variables contain both integers and continuous decimals, we define the problem ($\mathcal{P}$) as a mixed integer problem (MIP), which is a proved *NP-complete* problem. Even though this problem has a pseudo-polynomial solution, it is challenging to achieve fast and effective scheduling with such compute-intensive bursty requests, not to mention that the users are arriving and leaving dynamically. Inspired by the great success in decision-making achieved by the deep reinforcement learning in dynamic environments, we propose a DRL-based method to solve this problem. Nonetheless, this section concluding system and formulation is essential to our DRL model. For example, the total delay T_i^t in Eq. (12) and Eq. (13) are key vital elements for the reward definition, and the final simulation is established based on this system model.

4. DRL-based Algorithm Design

In this section, we first introduce the basic learning mechanism of DeepLoad. Then, we present how we transform the AP selection and workload redistribution into a learning task. Finally, we design a DRL-based algorithm in details.

4.1. Basic Learning Mechanism

Unlike the existing request scheduling strategies using predefined rules or specific heuristics, DRL is committed to learning an effective policy from the past experiences based on the current state and instant reward. To better understand the learning mechanism of DRL, we show the workflow of DeepLoad as illustrated in Fig. 5. A RL-agent interacts with the environment, where the RL-agent is the main component for making scheduling decisions, and the environment is an abstraction that integrates the information about edge network, mobile users, edge servers, and diverse requests. The RL-agent can only observe a small part of the environment, which is called *state*. In this paper, we view each mobile device as a *RL-agent*, the known information about network and edge server as *state*, and the scheduling decision as *action*. At each time slot t, the RL-agent observes a state s_t and chooses an action a_t based on the specific policy π. When the action is done, the current state will transit to the next state s_{t+1} and the agent will receive an instant reward r_t. Through constant interaction with the environment until done, the RL-agent is likely to get higher accumulated rewards. The objective of DRL is to find the best policy π (i.e. action probability distribution in A3C) mapping a state to an action that maximizes the expected discounted accumulated reward as $\mathbf{E}\left[\sum_{t=t_0}^{t_0+|\mathcal{T}|}\gamma^t r_t\right]$, where t_0 is the current time and $\gamma \in (0,1]$ is a factor to discount the future reward.

Note that each RL-agent (i.e., a mobile user) makes scheduling decisions based on probability distributions (i.e. policy π) rather than specific actions, which can potentially avoid excessive loads on a single edge server.

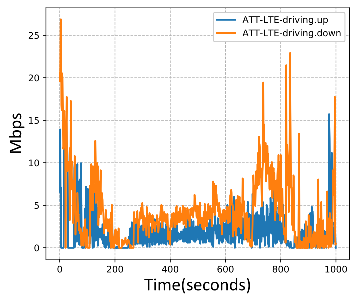
4.2. Algorithm Design

Due to the lack of future knowledge and the state transition probability matrix, as well as the discrete decision space, we propose the model-free DRL-based *DeepLoad*, which is trained using a state-of-the-art actor-critic DRL model called A3C. We introduce the detailed functionality design as follows.

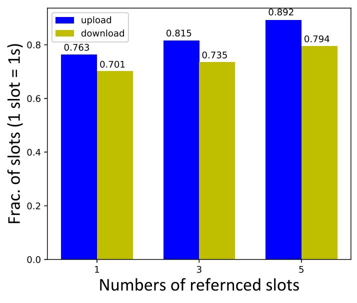
4.2.1. State Space

The state is the observation of a RL-agent (i.e. mobile device or user in our scenario) from the environment. The RL-agent aims to constantly learn policies from historical information to approach the perspective of the God (i.e. have future and global knowledge), thus a comprehensive state is critical to the decision-making efficiency. We take the known information of network, edge server and request into consideration as Fig. 6 shows. Specifically, we list the components as follows.

▷ Estimated bandwidth vector for uplink $\boldsymbol{b^u}$ and downlink $\boldsymbol{b^d}$. We denote them as $\boldsymbol{b^u} = \left\langle r_{ul}^1, r_{ul}^2, \cdots, r_{ul}^N \right\rangle$ and $\boldsymbol{b^d} = \left\langle r_{dl}^1, r_{dl}^2, \cdots, r_{dl}^N \right\rangle$, where r_{ul}^i is the uplink bandwidth from local user to AP_i, and r_{dl}^j represents the downlink bandwidth from AP_j. We set $r_{ul}^k = r_{dl}^k = 0$ if user u_i is not within the service area of AP_k. Considering that we have no means to get the real-time bandwidth for current. Many requests are initiated from mobile devices over cellular networks like LTE, which experience frequent bandwidth fluctuation [19]. To illustrate the variability of bandwidth, we depict two network traces from the Mahimahi[20] project as Fig. 4 shows. Across the upload and download traces, we made the following observations: (1) Periods of extreme low/high are uncommon: only 14.5% of the time, the upload bandwidth is 0 or larger than 10 *Mbps*, and 14.9% for the download bandwidth; (2) The



(a) Available network bandwidth



(b) Bandwidth estimation

Figure 4: Bandwidth fluctuation over time. (a) The bandwidth of uplink and downlink. (b) The Y-axis denotes the fraction of slots, in which the bandwidth is within $[80\%, 120\%]$ of the average bandwidth of past 1, 3 or 5 slots.

bandwidth of the next slot is closely related to the values of past several slots: as Fig. 4b shows, for uploading capacity, 76.3% slots own less than 20% bandwidth variation compared to the previous one slot, and it reaches 89.2% when referring to past five slots. The download capacity shares the similar rule with upload capacity. Hence, we view the weighted bandwidth of previous k slots as the estimated bandwidth of slot $t+1$,

$$r^i_{ul}(t+1) = \sum_{j=t-k+1}^{t} \omega_j r^i_{ul}(j), \tag{19}$$

where $\omega_m < \omega_n$ if $m < n$, and $\sum \omega_j = 1$. We estimate the downlink bandwidth in the similar method.

▷ Estimated propagation delays of inter-APs. We denote it as $\boldsymbol{b^p} = \left\langle l_{(i,j)} | i \neq j, i, j \in \{1, 2, ..., N\} \right\rangle$. Practically, the APs are connected with each other through S1 interface of Core Network (rarely X2 interface for transferring amounts of data). Hence, it remains relatively stable. To reflect the variance of different APs pair, we still view it as an input for DeepLoad.

▷ Pending workload $\boldsymbol{w}$ of each edge server, $\boldsymbol{w} = \langle w_1, w_2, \cdots, w_N \rangle$. Assume that each AP periodically and frequently sends a heartbeat message to the users in its proximity, which includes the amount of workloads to be processed in each queue on this server. Although it is a delayed message relative to the current, it makes sense to evaluate the pending workload.

▷ Critical features of current request. The input size B of request affects the transmission delay, the workload W determines the processing delay, and

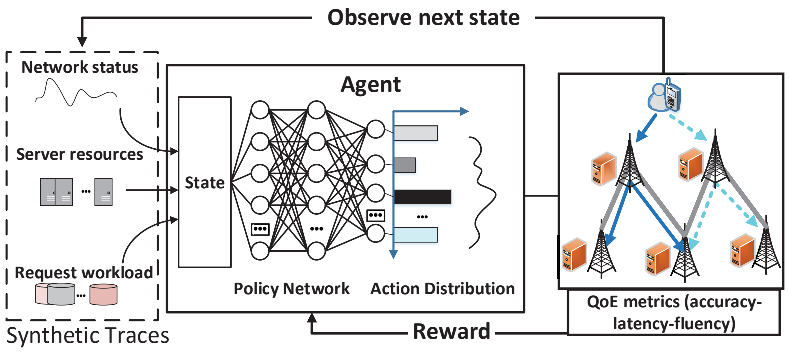


Figure 5: Illustration of the basic learning mechanism. RL-agent trains the policy network through continuous interaction with the environment. For each state observed from the environment, the RL-qgent can make a quick decision based on the action distribution.

deadline ddl represents the user's expectation.

We integrate the above components together and present the state as $s_t = \left\{ \boldsymbol{b^u},\ \boldsymbol{b^d},\ \boldsymbol{b^p},\ \boldsymbol{w},\ B,\ W,\ ddl \right\}$.

4.2.2. Policy

In our proposed DeepLoad, a RL-agent needs to take an action for request scheduling when receiving a state s_t. In our scenario, a request goes through two stages (i.e. AP selection and workload redistribution) from its generation to completion, and the second stage relies largely on the first stage, because the servers in the second stage are adjacent to the server in the first stage. Thus, we jointly consider both of AP selection and workload redistribution. Thus the action space is represented as $\{AP_k, AP_k^1, \cdots, AP_k^z, P_k, P_k^1, \cdots, P_k^z\}$, where AP_k is the first AP, P_k is the percentage of workloads processed at AP_k, AP_k^z and P_k^z denote a neighboring server of AP_k and the corresponding offloading proportion, thus, $P_k + \sum_{1 \leq i \leq z} P_k^i = 1$ and $P_k, P_k^i \in \{0, 1, \cdots, 100\}$. Therefore, the action space is bounded. However, the value of the workloads of a request is continuous and unbounded, so there are infinite $(state, action)$ pairs. We cannot store them in tabular form and solve the problem using traditional methods, e.g., Q-learning and SARSA. Fortunately, the A3C model addresses this issue perfectly, which uses a neural network [9] to represent a policy π, and the adjustable parameter of the neural network is referred to as the policy parameter θ. Therefore, we can present the policy as $\pi\left(a_t | s_t; \theta\right) \rightarrow [0, 1]$, indicating the probability of taking action a_t at state s_t.

Note that the different types of requests have different policies; if there are $|\mathcal{M}|$ types of requests, $|\mathcal{M}|$ policies are needed.

4.2.3. Reward

Once applying an action a_t to the state s_t, a RL-agent will receive an instant reward r_t from the environment. Recall that in the problem formulation, our optimization objective is to maximize the number of requests that finish before their deadlines. If the deadline can be met in the first stage (i.e., AP selection), the second stage can be skipped directly. To mitigate the risk of privacy leakage and the expensive communication cost of inter-APs, users may prefer to execute their requests without edge collaboration. However, when the estimated delay of the first stage is worse than the deadline, the workload redistribution stage is needed. To reflect the risk of privacy leakage and the cost of edge collaboration, we define the reward as

$$reward=\begin{cases}\frac{ddl_i^t-T_i^t}{ddl_i^t}, \text{ if redistribution is performed,}\\ \frac{ddl_i^t-T_i^t}{T_i^t}, \text{ otherwise,}\end{cases} \tag{20}$$

where ddl_i^t is the attached deadline of u_i at time slot t, and T_i^t is the completion time of u_i. T_i^t is modeled in details in section 3. If P_k in the action is 100, then the total workload is executed in the edge server deployed at the selected AP. Apparently, the reward may be a negative when the T_i^t can't meet the ddl_i^t, which is acceptable on account of considering a maximized accumulative reward. According to Eq. (20), for the same completion time T_i^t, the reward without the workload redistribution stage is much more attractive.

4.2.4. DRL Model Training Methodology

The configuration space is bounded, but the sophisticated state space seems infinite, thus there are endless (s_t, a_t) pairs. Instead of storing the value of each (s_t, a_t) pair in tabular form, e.g. Q-table, we adopt A3C, which uses a neural network to represent a policy π as Fig. 6 shows, and the adjustable parameter of the neural network is referred to as the policy parameter θ. Therefore, we can present the policy as $\pi\left(a_t|s_t;\theta\right) \rightarrow [0,1]$, indicating the probability of taking action a_t at state s_t. The objective of DRL is to find a best policy π mapping a state to an action that maximizes

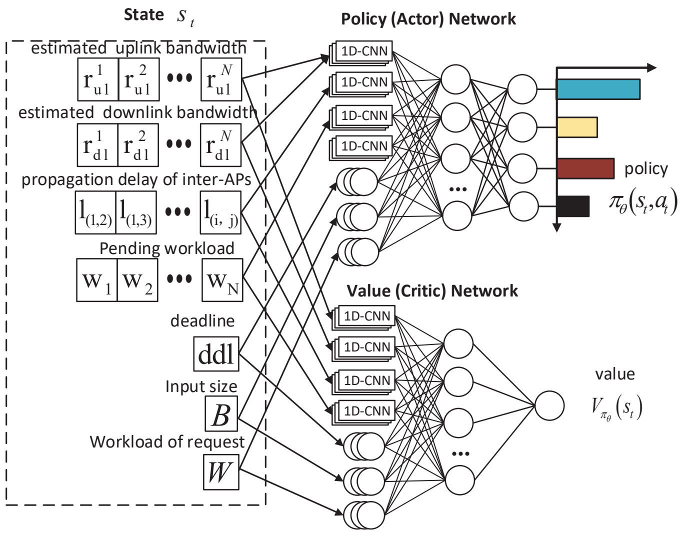


Figure 6: The Actor-Critic algorithm that DeepLoad uses to generate scheduling policies.

the expected accumulative discounted reward as

$$J(\theta) = \mathbf{E}\left[\sum_{t=t_0}^{t_0+|\mathcal{T}|} \gamma^t r_t\right], \tag{21}$$

where t_0 is the current time and $\gamma \in (0,1]$ is a factor to discount the future reward.

4.2.5. Policy Gradient Training

The actor-critic network used by DeepLoad is trained with *policy gradient method*, whose key idea is to estimate the gradient of the expected total reward by observing the trajectories of executions obtained by following the policy. We highlight the key steps of the algorithm, focusing on the intuition. The policy gradient of $J(\theta)$ with respect to θ, to be used for policy network update for slot t, can be calculated as follows [21]:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\sum_{t\in\mathcal{T}} \nabla\boldsymbol{\theta} \log\left(\pi_{\boldsymbol{\theta}}\left(s_t, a_t\right)\right) A^{\pi_{\boldsymbol{\theta}}}\left(s_t, a_t\right)\right], \tag{22}$$

where $A^{\pi_{\boldsymbol{\theta}}}\left(s_t, a_t\right)$ is the advantage function that represents the gap between the expected accumulative reward when we deterministically select a_t at state s_t following π_θ and the expected reward for actions drawn from policy $\pi_{\boldsymbol{\theta}}$.Indeed, the advantage function reflects how much better a current specific action is compared to the "average action" taken based on the policy. Intuitively, we reinforce the actions with positive advantage value $A^{\pi_{\boldsymbol{\theta}}}(s, a)$, but degrade the actions with negative advantage value $A^{\pi_{\boldsymbol{\theta}}}(s, a)$.

In particular, the RL-agent extracts a trajectory of scheduling decisions for the bursty requests and uses the empirically computed advantage $A(s_t, a_t)$ as an unbiased estimated $A^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)$. The update rule of actor network parameter θ follows the policy gradient,

$$\theta \leftarrow \theta + \alpha \sum_{t \in \mathcal{T}} \nabla_\theta \log \pi_\theta (s_t, a_t) A(s_t, a_t), \tag{23}$$

where α is the learning rate. The marrow behind this update law is summarized as follows, the gradient direction $\nabla_\theta \log \pi_\theta (s_t, a_t)$ indicates how to change parameter θ to improve $\pi_\theta (s_t, a_t)$ (i.e., the probability of action a_t at state s_t). Eq. (23) goes a step along the gradient descent direction. The specific step size is up to the advantage value $A^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)$. Hence, the goal of each update is to reinforce actions that empirically have better feedbacks. To compute the advantage value $A(s_t, a_t)$ for a given sample, we need to get the estimated *value function* $v^{\pi_{\boldsymbol{\theta}}}(s)$, *i.e.*, the total expected reward starting at state s following the policy $\pi_{\boldsymbol{\theta}}$. As Fig. 6 shows, the role of *critic network* is to learn an estimated $v^{\pi_{\boldsymbol{\theta}}}(s)$ from observed rewards. We update the critic network parameters θ_v based on the *Temporal Difference* [22] method,

$$\theta_v \leftarrow \theta_v - \alpha' \sum_t \nabla_{\theta_v} (r_t + \gamma V^{\pi_\theta}(s_{t+1}; \theta_v) - V^{\pi_\theta}(s_t; \theta_v))^2, \tag{24}$$

where $V^{\pi_\theta}(s_t; \theta_v)$ is the estimated $v^{\pi_{\boldsymbol{\theta}}}(s_t)$ that is produced by the critic network, and α' is the learning rate. We take a specific experience $\langle s_t, a_t, r_t, s_{t+1} \rangle$ – take action a_t for state s_t, obtain instant reward r_t, and transit to next state s_{t+1}– as an example, we estimate the advantage value $A(s_t, a_t)$ as $r_t + \gamma V^{\pi_\theta}(s_{t+1}; \theta_v) - V^{\pi_\theta}(s_t; \theta_v)$. Note that the critic network does nothing to train the actor network other than evaluate the policy of the actor network. In the actual AR scenario, only the actor network is involved in making configuration decisions.

To reach an adequate exploration for the RL agent during training to discover better policies, thereby reducing the risk of falling into suboptimal, we add an *entropy regularization* [15] term to encourage exploration. This practice is significant to help the agent converge to a fine policy. Correspondingly, we modify Eq. (23) to be

$$\theta \leftarrow \theta + \alpha \sum_{t} \nabla_{\theta} \log \pi_{\theta}(s_t, a_t)\, A\left(s_t, a_t\right) + \beta \nabla_{\theta} H\left(\pi_{\theta}\left(\cdot | s_t\right)\right), \tag{25}$$

where β is entropy weight that is set to a large value and decreases over time to emphasize improving rewards, and $H(\cdot)$ is the policy entropy to encourage exploration by pushing θ in the direction with higher entropy.

4.2.6. Parallel Training

To further enhance exploration and speed up training. As shown in Fig. 7, we use a parallel approach to obtain abundant training samples quickly. We start n threads (i.e. agents) at the same time, and adopt diverse environment settings (e.g., diverse network traces and AR videos). Different agents are likely to experience different states and transitions, thus avoiding the correlation. Specifically, each agent continuously collects their samples (tuple $\{s_t, a_t, r_t, s_{t+1}\}$), and uses the actor-critic algorithm to compute a gradient and perform a gradient descent step as Eq. (24) and Eq. (25), independently. Then, each agent pushes its actor parameters to the central agent, which integrates the parameters, and generates a global actor network. Finally, each agent pulls the global model from the central agent, and starts the next training episode until the global actor network is convergent. Since the actor-critic network has been well trained, we can take a fast and accurate action based on the action probability distribution for each encoding slot.

5. Performance Evaluation

In this section, we validate the performance of DeepLoad with extensive data-driven simulations. In addition, we verify its efficiency through several control experiments.

5.1. BRS Simulator

An ideal well-trained DeepLoad needs numerous samples. It is unrealistic to train DeepLoad through continuous trial and error in the real scenario due to the unbearable cost. Instead, we design a BRS (Bursty Requests Scheduler) simulator that best matches the real scenario.

First of all, we depict the distribution of APs as well edge servers, and simulate the arrival model of bursty requests. Several vital characteristics of Shanghai taxi trajectory data set [23] that inspire us are as follows: (1)

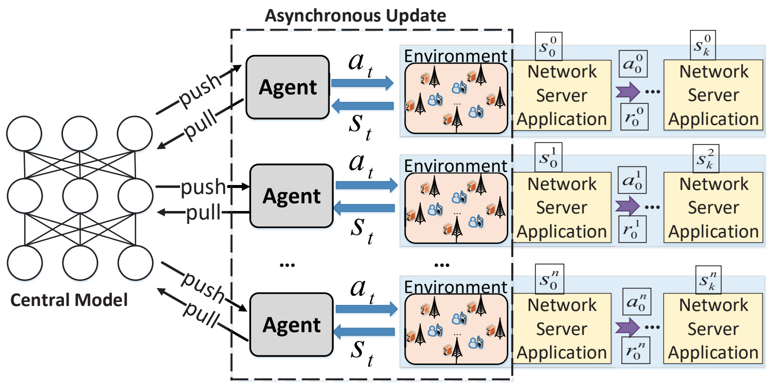


Figure 7: The training methodology of A3C model. A3C adopts multi-thread technology to train the actor-critic network. Each thread, which can be viewed as a RL-agent, trains its own network independently, and interacts with the main network through pull-push method.

it records the real-time GPS location (i.e., the longitude and latitude) of each taxi; (2) it documents the time stamps of entering and leaving the common infrastructures such as stations or intersections for each taxi. Hence, the variant GPS location of each taxi can be applied to show the user's mobility. Based on the taxi's location in the same slot, we mark several regions called Point of Interests (PoIs) that have crowded taxis with similar location, and these PoIs are perfect candidates for deploying APs and edge servers. Practically, the PoIs is likely to be a station or an intersection. Note that each taxi may be in the coverage of several PoIs simultaneously. For example, the stops are close to intersections. What's more, we count the number of taxis residing in each PoI at each slot based on the time stamps, and thus we use the residing taxis at each slot to represent the bursty requests identified by tuple $\langle W, B, ddl \rangle$. Specifically, we select the data records for 28 February, 2007. By preprocessing, we know the number of PoIs is 12. Each time slot is 5 minutes, thus a day has 288 time slots, which form an episode. Note that this time slot of 5 minutes is just for sampling, and the actual time slot towards bursty request is far less than it, perhaps only 1 second.

Furthermore, the initialization of requests (i.e., B, W, ddl) as well as edge servers, and the simulation of a dynamic edge environment (i.e. the time-variant network condition and server status) are also supposed to be well-crafted. We introduce a general-used mechanism named *transparent request offloading*, in which every mobile user knows nothing about other users, including the location, the type of request, and not to mention the

Table 3: Initial parameter setting.

Types	Parameters
Input size	$[3000, 4000]$
Total workload	$[400, 600]$
Pending workload	$[125, 175]$
Propagation delay of inter-APs	$[25, 35]$

Table 4: Actor-Critic network design of DeepLoad

Types	Actor network	Critic network
Input layer	4×1D-CNN+3	4×1D-CNN+3
Hidden layer	$400 \times 400 \times 400$	$400 \times 400 \times 400$
Output layer	$\|action\_space\|$	1

exact value of $\langle W, B, ddl \rangle$. This mechanism is in line with the ideology of DRL, and the characteristic of edge environment. Specifically, we model the fluctuation of uplink and downlink bandwidth (i.e., r_{ul} and r_{dl}) in a trace-driven method from the Mahimahi[20] project recorded the transmitted MTU-sized packed per millisecond. We make some statistics and modifications, and obtain the bandwidth per millisecond. In BRS, the input scale B, total workload W and deadline ddl of request, and the pending workload w_q in each edge server is initialized from uniform distribution with different parameters as Table 3 shows. Particularly, we set the deadline ddl from range $\left[\frac{B}{r(avg)}+\frac{W}{c(avg)}-10, \frac{B}{r(avg)}+\frac{W}{c(avg)}+10\right]$, where $r(avg)$ is average bandwidth, and $c(avg)$ represents the average computing capability.

5.2. Training Testbed and Benchmark

We train the DeepLoad learning model using the A3C model. The general Actor-Critic networks structure is illustrated as Fig. 6, they share the same parameters of the input layer and hidden layer, but output the action distribution and value, respectively. The detailed design is listed in Table 4. To enhance the convergent rate and training efficiency, we leverage two GeForce GTX TITAN Xp GPUs. In DeepLoad, an episode including 288 slots can be viewed as a training sequence in DRL. We set the following benchmarks to further evaluate the performance of DeepLoad:

- **SSP** (Single Server Processing). Each request is processed only in the server selected in *AP selection*. The user first calculates the estimated delay for each accessible AP, and then selects the optimal one.

- **DSP** (Double Server Processing). Each request is processed in the server selected in *AP selection* and one of its adjacent server. The user first calculates the estimated delay for each accessible AP, and then selects the optimal one. If the estimated completion time is bigger than the *ddl*, the selected AP will offload half of the workload to one of its adjacent server with the most available resources.

- **LOCP** (Link Optimal Collaborative Processing). In *workload redistribution*, the selected AP chooses two neighboring servers with the lowest propagation delay, and then offloads some percentages of workloads to them. Note that the offloading proportions are the same as the target proportions in the action chosen by DeepLoad.

- **QOCP** (Queue Optimal Collaborative Processing). The selected AP selects two neighboring servers with the minimum amount of pending workloads in the queue, and then offloads some percentages of workloads to them. QOCP and DeepLoad have the same target proportions.

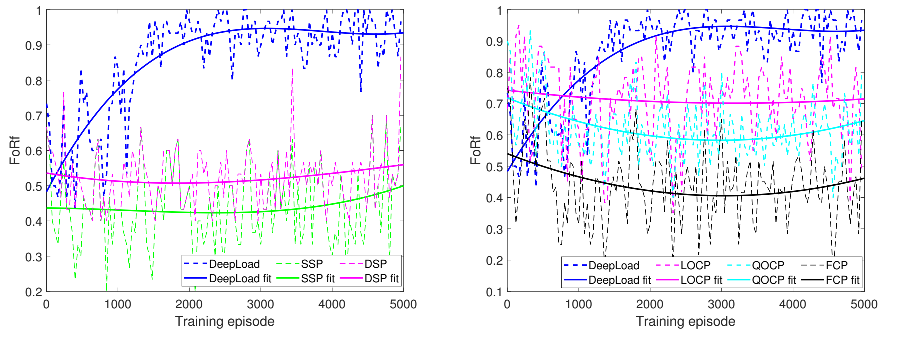
- **FCP** (Fair Collaborative Processing). The selected AP selects two neighboring servers and offload one third of the workloads to them.

5.3. Effectiveness and Impact Factor of DeepLoad

In this subsection, we make a horizontal and vertical comparison for DeepLoad. We first compare our model with benchmarks in the following aspects to analyze its effectiveness. Then, we show the impact of the inherent parameter setting of DeepLoad on the performance.

5.3.1. Effectiveness of DeepLoad

In our scenario, we aim to maximize the number of requests whose completion times are ahead of their deadlines. Since the number of users at each time slot is dynamically changing, we use the Fraction of Requests finished (**FoRf**) before deadline to measure the effectiveness of DeepLoad. As illustrated in Fig. 8, with the increasing number of training episodes, the FoRf of DeepLoad approaches 1, which implies almost all of the requests can be completed before deadlines. As Fig. 8a shows, even with an optimal AP selection, traditional SSP performs poorly with a FoRf less than 0.5, which is largely due to the unpredictable queuing time. DSP mitigates this dilemma,



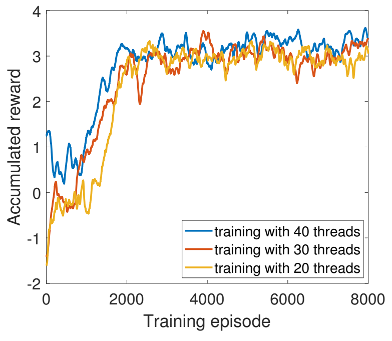
(a) DeepLoad vs. SSP and DSP (b) DeepLoad vs. LOCP, QOCP and FCP

Figure 8: Comparison of DeepLoad and other strategies. Y-axis denotes the fraction of requests finished before deadlines (i.e. FoRf) in current episode.

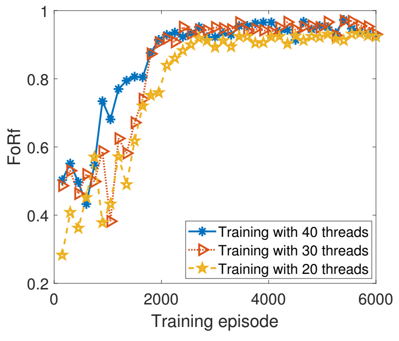
but it still performs badly. The edge servers adopt FCFS such that each new incoming request is arranged at the end of the queue. As Fig. 8b shows, in the early episodes of training, Deepload focuses on exploration such that its average effectiveness is even inferior to LOCP, QOCP and FCP, yet with more training episodes, DeepLoad has a significant performance improvement and shows superiority to them. Note that DeepLoad can greatly alleviate the burden produced by the bursty requests, but cannot fundamentally eliminate such burden, because there may be too many requests that exceed the processing power of the LAN. Therefore, the FoRf of DeepLoad presents a fluctuating state but maintains at a high value.

5.3.2. Impact Factor

The internal parameter settings are critical to the model performance. Take the number of parallel threads and the learning rate as two examples. We train our DeepLoad through 10,000 episodes under different thread numbers and learning rates. As described in Section IV, we use a parallel approach (i.e. multithreading technology) to obtain abundant training samples and adopt diverse environment settings. Generally, the more threads the training adopts, the broader scope the DeepLoad can explore, and thus almost all exploratory sequences can be gathered. In Fig. 9a, the accumulated reward of DeepLoad trained with 40 threads surpasses the other two cases, while the final FoRf with different threads are almost the same as shown in Fig. 9b. Secondly, we show the relationship between the accumulated reward

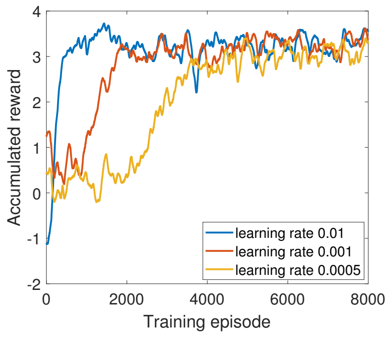


(a) Accumulated reward.

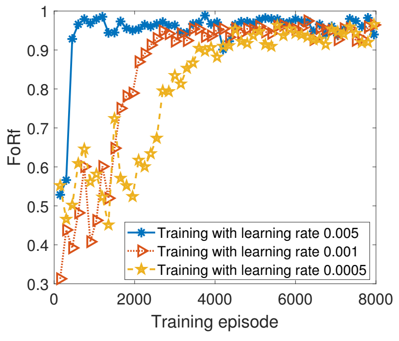


(b) FoRf.

Figure 9: The effects of thread numbers on the results.



(a) Accumulated reward.



(b) FoRf.

Figure 10: The effect of learning rate on the results.

and the learning rate in Fig. 10. It is clear to see that the model with a higher learning rate reaches the peak interval faster (e.g. convergence) but gains a more dramatic fluctuation. Generally, the selection of learning rate depends on the system-level need. A high learning rate means fast convergence speed and high volatility, while a low learning rate means slow convergence speed and low volatility.

6. Related Work

6.1. Scheduling Compute-intensive Requests with Cooperative Methods

Although the new emerging edge computing paradigm brings lots of possibilities to process the compute-intensive requests efficiently by pushing the services closer to the end users [1–3], it is challenging to handle bursty requests via a single edge server. In [7], the authors adopted an ARM big.LETTLE architecture, and aimed to minimize the energy consumption through a better requests scheduling policy. In [9], the authors proposed VideoStorm, a video analytics system that explores the accuracy-resource trade-off in improving analytics quality and lag. In [10], the authors proposed OREO, which jointly optimizes dynamic service caching and task offloading to address service heterogeneity, unknown system dynamics, spatial demand coupling and decentralized coordination. In [24], the authors propose a hierarchical model with intra-fog and inter-fog resource management. In [25], the authors proposed an effective task scheduling approach with stochastic time cost for computation offloading in mobile edge computing. Some other studies [8, 26, 27] focused on scheduling requests to either of local device, single edge edge, or remote cloud for execution, while we propose DeepLoad to process bursty requests through edge server collaboration.

6.2. DRL-based Application

Recently, Deep Reinforcement Learning (DRL) has shown its superiority in many fields. In [14], the authors first used Deep Q-Network to learn policies from sensor input for decision making. In [28], the authors presented a comprehensive application of DRL in communication and network. In [29, 30], the authors adopted DQN-based computation offloading strategies for IoT devices, which aim to achieve automatic scheduling. In [31–34], the authors considered a multi-user MEC system, and proposed A3C-based optimization framework to tackle resource allocation for MEC. In [35], the authors used the A3C algorithm to select the optimal bitrate for future video chunks purely based on the past experience. In [36], the authors applied DRL to the traffic engineering problem. In [37], the authors proposed ReLeS for Multipath TCP, which supports a real-time packet scheduling. To our best knowledge, DeepLoad is the first to apply DRL to solve the bursty requests scheduling problem in edge computing environments.

7. Conclusion

In this paper, we consider a general edge scenario of bursty requests, in which we aim to learn an efficient scheduling policy. We first formulate it as a long-term optimization problem that maximizing the number of requests finished before deadlines, which is referred to as a NP-complete problem.

Inspired by the great achievements in decision-making of DRL in dynamic environments, we propose DeepLoad, an intelligent scheduler for bursty requests via deep reinforcement learning in edge environments. Finally, based on the real data set, we design a LAN simulator to collect abundant samples, and train the actor-critic network with numerous episodes. In addition, we design several control experiments, and further demonstrate the superiority of DeepLoad compared to several baseline algorithms.

References

[1] P. Mach, Z. Becvar, Mobile edge computing: A survey on architecture and computation offloading, IEEE Communications Surveys & Tutorials 19 (3) (2017) 1628–1656.

[2] Y. Mao, C. You, J. Zhang, K. Huang, K. B. Letaief, Mobile edge computing: Survey and research outlook, arXiv preprint arXiv:1701.01090.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet of Things Journal 3 (5) (2016) 637–646.

[4] M. Xiao, J. Wu, L. Huang, R. Cheng, Y. Wang, Online task assignment for crowdsensing in predictable mobile social networks, IEEE TMC 16 (8) (2017) 2306–2320.

[5] Y. Li, W. Gao, Muvr: Supporting multi-user mobile virtual reality with resource constrained edge cloud, in: IEEE/ACM Symposium on Edge Computing, IEEE, 2018, pp. 1–16.

[6] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, N. Dai, H.-S. Lee, Furion: Engineering high-quality immersive virtual reality on today's mobile devices, IEEE TMC.

[7] Y. Geng, Y. Yang, G. Cao, Energy-efficient computation offloading for multicore-based mobile devices, in: IEEE INFOCOM, IEEE, 2018, pp. 46–54.

[8] M.-H. Chen, B. Liang, M. Dong, Joint offloading and resource allocation for computation and communication in mobile cloud with computing access point, in: IEEE INFOCOM, IEEE, 2017, pp. 1–9.

[9] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, M. J. Freedman, Live video analytics at scale with approximation and delay-tolerance, in: 14th USENIX NSDI, USENIX, 2017, pp. 377–392.

[10] J. Xu, L. Chen, P. Zhou, Joint service caching and task offloading for mobile edge computing in dense networks, in: IEEE INFOCOM, IEEE, 2018, pp. 207–215.

[11] X. Chen, L. Pu, L. Gao, W. Wu, D. Wu, Exploiting massive d2d collaboration for energy-efficient mobile edge computing, IEEE TWC 24 (4) (2017) 64–71.

[12] D. Wu, Q. Liu, H. Wang, D. Wu, R. Wang, Socially aware energy-efficient mobile edge collaboration for video distribution, IEEE TMM 19 (10) (2017) 2197–2209.

[13] H. Guo, J. Liu, Collaborative computation offloading for multiaccess edge computing over fiberwireless networks, IEEE Transactions on Vehicular Technology 67 (5) (2018) 4514–4526. doi:10.1109/TVT.2018.2790421.

[14] 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 518 (7540) (2015) 529.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in: ACM ICML, ACM, 2016, pp. 1928–1937.

[16] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, D. Meger, Deep reinforcement learning that matters, in: Thirty-Second AAAI, 2018.

[17] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, F. Bai, Hermes: Latency optimal task assignment for resource-constrained mobile computing, IEEE TMC 16 (11) (2017) 3056–3069.

[18] M.-H. Chen, B. Liang, M. Dong, Joint offloading decision and resource allocation for multi-user multi-task mobile cloud, in: IEEE ICC, IEEE, 2016, pp. 1–6.

[19] K. Winstein, A. Sivaraman, H. Balakrishnan, Stochastic forecasts achieve high throughput and low delay over cellular networks, in: 10th USENIX NSDI, USENIX, 2013, pp. 459–471.

[20] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, H. Balakrishnan, Mahimahi: Accurate record-and-replay for http, in: 2015 USENIX ATC, USENIX, 2015, pp. 417–429.

[21] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: NIPS, 2000, pp. 1057–1063.

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

[23] Shanghai taxi trajectory traces, `http://wirelesslab.sjtu.edu.cn/taxitracedata.html`.

[24] W. Zhang, Z. Zhang, H.-C. Chao, Cooperative fog computing for dealing with big data in the internet of vehicles: Architecture and hierarchical resource management, IEEE Communications Magazine 55 (12) (2017) 60–67.

[25] W. Zhang, Z. Zhang, S. Zeadally, H.-C. Chao, Efficient task scheduling with stochastic delay cost in mobile edge computing, IEEE Communications Letters 23 (1) (2018) 4–7.

[26] S. Bi, Y. J. Zhang, Computation rate maximization for wireless powered mobile-edge computing with binary computation offloading, IEEE TWC 17 (6) (2018) 4177–4190.

[27] J. Du, L. Zhao, J. Feng, X. Chu, Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee, IEEE TCOM 66 (4) (2018) 1594–1608.

[28] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, D. I. Kim, Applications of deep reinforcement learning in communications and networking: A survey, IEEE Communications Surveys & Tutorials.

[29] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, W. Zhuang, Learning-based computation offloading for iot devices with energy harvesting, IEEE Transactions on Vehicular Technology 68 (2) (2019) 1930–1941.

[30] M. G. R. Alam, M. M. Hassan, M. Z. Uddin, A. Almogren, G. Fortino, Autonomic computation offloading in mobile edge for iot applications, FGCS 90 (2019) 149–157.

[31] J. Li, H. Gao, T. Lv, Y. Lu, Deep reinforcement learning based computation offloading and resource allocation for mec, in: IEEE WCNC, IEEE, 2018, pp. 1–6.

[32] Y. He, F. R. Yu, N. Zhao, V. C. Leung, H. Yin, Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach, IEEE Communications Magazine 55 (12) (2017) 31–37.

[33] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, M. Bennis, Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning, IEEE Internet of Things Journal 6 (3) (2019) 4005–4018.

[34] C. Zhang, Z. Liu, B. Gu, K. Yamori, Y. Tanaka, A deep reinforcement learning based approach for cost-and energy-aware multi-flow mobile data offloading, IEEE TCOM E101.B (7) (2018) 1625–1634.

[35] H. Mao, R. Netravali, M. Alizadeh, Neural adaptive video streaming with pensieve, in: ACM SIGCOMM, ACM, 2017, pp. 197–210.

[36] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, D. Yang, Experience-driven networking: A deep reinforcement learning based approach, in: IEEE INFOCOM, IEEE, 2018, pp. 1871–1879.

[37] H. Zhang, W. Li, S. Gao, X. Wang, B. Ye, Reles: A neural adaptive multipath scheduler based on deep reinforcement learning, in: IEEE INFOCOM, IEEE, 2019, pp. 1648–1656.