Adaptive Online Decision Method for Initial Congestion Window in 5G Mobile Edge Computing Using Deep Reinforcement Learning

Ruitao Xie, Xiaohua Jia, *Fellow, IEEE*, and Kaishun Wu

***Abstract*—Mobile edge computing provides users with low response time and avoids unnecessary data transmission. Due to the deployment of 5G, the emerging edge systems can provide gigabit bandwidth. However, network protocols have not evolved together. In TCP, the initial congestion window (IW) is such a low value that most short flows still stay in slow start phase when finishing, and do not fully utilize available bandwidth. Naively increasing IW may result in congestion, which causes long latency. Moreover, since the network environment is dynamic, we have a challenging problem—how to adaptively adjust IW such that flow completion time is optimized, while congestion is minimized. In this paper, we propose an adaptive online decision method to solve the problem, which learns the best policy using deep reinforcement learning stably and fast. In addition, we propose an approach to further improve the performance by supervised learning, using data collected during online learning. We also propose to adopt SDN to address the challenges in implementing our method in MEC systems. To evaluate our method, we build an MEC simulator based on ns3. Our simulations demonstrate that our method performs better than existing methods. It can effectively reduce FCT with little congestion caused.**

***Index Terms*—Congestion control, initial congestion window, deep reinforcement learning, mobile edge computing, software-defined networking.**

I. Introduction

MOILE edge computing (MEC) is emerged as a localized cloud. It installs shared storage and computation resources within radio access networks [1], [2], as shown in Fig. 1. Due to its proximity to mobile users, it can provide users with low response time, and avoid unnecessary data transmission. At the same time, software-defined networking (SDN) is utilized in MEC systems to enable intelligent network management and service orchestration [3], [4].

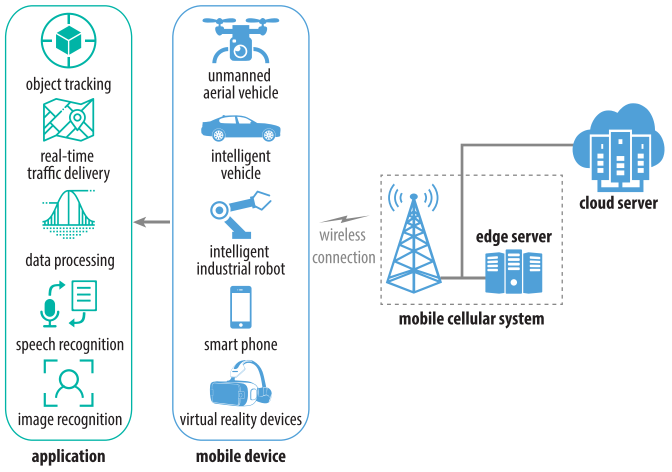


Fig. 1. The architecture of mobile edge computing and some applications benefiting from it.

In edge computing, the system paradigm changes, but the network protocols above it have not evolved together. This significantly inhibits the further reduction of latency. The majority of applications rely on TCP, a widely used transport layer protocol. It dynamically adjusts a congestion window for a sender according to some congestion signal, such as packet loss. The congestion window is started from a fixed initial value. It is around two in the early version of TCP, and is increased to ten in 2010 [5], [6]. However, recent research works suggest that most short flows (such as web search) still stay in slow start phase when finishing and do not fully utilize available bandwidth [7], [8]. 5G MEC will make this situation more severe for two reasons: 1) a great proportion of flows are short. This can be estimated from the existing research works on LTE mobile traffic, such as [9]. 2) incoming 5G cellular systems can provide gigabit bandwidth, which is greatly higher than present networks [10]. This encourages us to increase initial congestion window (IW) in 5G MEC system.

The higher the IW is, the less round trip time it takes to reach the saturate bandwidth. In other words, high IW may lead to low latency. However, naively increasing IW may result

Manuscript received June 23, 2019; revised October 17, 2019; accepted November 6, 2019. Date of publication December 27, 2019; date of current version February 19, 2020. This work was supported in part by the Young Scientists Fund of China NSF under Grant 61802263, in part by the Research Grants Council of Hong Kong under Grant CityU 11214316, in part by the Faculty Research Fund of Shenzhen University under Grant 2019050 and Grant 860/000002110325, in part by the Tencent Rhino-Bird Young Faculty Research Fund, in part by the China NSFC under Grant 6187224 and Grant U1736207, in part by the Guangdong NSF under Grant 2017A030312008, in part by the Fok Ying-Tong Education Foundation for Young Teachers in the Higher Education Institutions of China under Grant 161064, and in part by the Guangdong Province Universities and Colleges Pearl River Scholar Funded Scheme (GDUPS) (2015). *(Corresponding author: Kaishun Wu.)*

R. Xie is with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China (e-mail: drtxie@gmail.com).

X. Jia is with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: csjia@cityu.edu.hk).

K. Wu is with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China, and also with the Peng Cheng Laboratory, PCL Research Center of Networks and Communications, Shenzhen 518066, China (e-mail: wu@szu.edu.cn).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2959187

in congestion, which causes long latency. Most applications targeted by edge computing demand real-time latency, thus congestion may have disastrous impact. Thus, a significant problem arises, that is *how to increase initial congestion window such that flow completion time is optimized, while congestion is minimized*.

This problem has two challenges. The first is how to build an accurate and general model which describes the influence of IW over the optimization objectives. Such modeling is hard to be accurate and general at the same time, as it requires manual analysis of all the intricacies of a complex communication system. The second challenge is how to formulate congestion. In existing works [7], congestion is always measured by RTT. High RTT implies congestion happening, while low RTT implies the opposite. The objective is to minimize a weighted sum of throughput and negative RTT. However, it is not easy to choose a suitable weight to balance those conflicting objectives.

To overcome the first challenge, we propose to formulate the problem as a Markov decision process (MDP) and use reinforcement learning to solve it. This approach learns a decision policy from experiences, by promoting good decisions and penalizing bad ones. For the second challenge, we consider that flow completion time (FCT) nicely indicates congestion, as suggested in [11], because once a flow suffers congestion, its FCT would increase drastically due to long retransmission delay.

We use neural network to represent the policy function, as existing works [12]–[15] show that it can learn features automatically and generalize well. The policy takes system state as the input and IW as the output. We select a set of factors to represent the system state. Finally, we solve the problem and obtain the best policy network using A3C, one of classic actor-critic algorithms.

When using reinforcement learning, we still encounter several challenges. The first is how to make the algorithm converge stably and fast. The second issue arises under a dynamic network environment. That is, a converged policy model under an old environment may perform poorly under a new environment. The third issue is how to implement our method in MEC systems. We address the above issues and make the following contributions:

- We propose a bracket of techniques to ensure the convergence of our algorithm: the usage of batch normalization to provide an effective exploration, reusing an IW decision to mitigate forward dependency and stabilize the training, and a mechanism of early feedback to updating policy timely.
- We propose an adaptive method to adjust the policy model according to the dynamic network environment.
- In order to obtain one generalized policy model, we propose an method that elegantly combines reinforcement learning and supervised learning.
- We propose a SDN-based implementation, which not only works in a single MEC system, but also enables collaborative learning among multiple MEC systems.

To evaluate our method, we build an MEC simulator based on ns3 and a module of 5G mmWave cellular network. We compare our method with SmartIW, an existing IW setting algorithm designed for search engines [7], [8], and a normal static IW setting. Our extensive simulations demonstrate that our method performs better than the others. It can effectively reduce average FCT with little congestion caused. Moreover, it is adaptive to dynamic traffic.

The rest of this paper is organized as follows. Section II presents an overview of the decision problem of intelligent IW in MEC and its MDP formulation. Section III gives an overview of our method and the techniques to ensure convergence. Section IV presents how to design policy function. Section V and Section VI present the online learning algorithm and the adaptive learning algorithm respectively. In Section VII, we discuss the details related to implementation and several issues on supervised learning. In Section VIII, we introduce simulation setup. The simulations and performance evaluations are presented in Section IX. Section X introduces the related works. Section XI concludes the paper.

II. Problem Overview and Formulation

Mobile edge computing aims to improve the performance of latency-sensitive applications and computation-intensive applications, by providing storage and computation resources at the edge of the network. We illustrate the architecture of MEC with Figure 1. An edge server (or a cluster of edge servers) is deployed within a Radio Access Network (RAN) [1]. A variety of mobile applications may request data from or offload computation to the edge sever rather than remote cloud servers, such as searching for smart phones, requesting real-time traffic information for vehicles, recognizing objects for unmanned aerial vehicles, and analyzing sensor data for IOT devices and so on.

Most applications use the TCP protocol to implement communications. That is, a client first establishes a TCP connection with the edge server, and then it sends a request to the server, and next the server sends a response back. Once a connection is established, each sender sends data at a dynamic rate managed by a congestion control algorithm. It initially sets the congestion window as a small value, usually 2 segments or 10 segments, and then increases or decreases the window according to some network feedback.

Increasing IW can reduce FCT and improve throughput. However, pushing IW to its maximum may result in more congestion events and longer FCT due to retransmission delay. There exists the best IW such that FCT is minimized without extra congestion caused. The best IW depends on system situation. If the system load is low, then there are small number of concurrent flows, an increased IW for each flow will reduce FCT without causing congestion, because the aggregate transmission rate is far less than network bandwidth. On the other hand, if the system load is high, then the same increased IW will result in severe congestion and long FCT, because at this time the aggregate transmission rate exceeds network bandwidth. Since system situation is dynamic, an adaptive decision is required. Thus, our problem is to select IW for each flow according to dynamic system situation, such that FCT is minimized, while no extra congestion is caused.

Note that in wireless network, the channel condition for different clients is very volatile, it varies with client position, physical occlusion, signal reflection, etc. While these are important factors, we leave these micro control issues to congestion control algorithm which caters more to situations of individual clients. We focus on setting IW according to the system traffic, which is a macro control and has a widespread effect over the system. Besides, there are many other factors impacting on TCP sending rate, such as flow control, window increment and decrement mechanisms in congestion control algorithm. The works on these factors to improve transmission performance are orthogonal to our work.

For MEC applications, there are uplink flows (a client to the edge server) and downlink flows (in reverse direction). It is easy to adjust IW for downlink flows, since only the edge server is involved and central control is possible. In contrast, it is hard to adjust IW for uplink flows due to distributed clients.

The intelligent IW decision problem is hard to solve in a traditional way, because it needs to model the influence of IW on FCT and congestion. Such modeling is hard to be accurate and general at the same time, as it requires manual analysis of all the intricacies of a complex communication system. Instead, we use reinforcement learning (RL) to solve the problem, which adapts to a new network situation through learning without requiring an explicit model of the situation in question. We formulate the problem as an MDP in following.

In MEC, the edge server receives a sequence of requests, and returns a response for each request. Our task is to determine an IW for the transmission of each response according to network state. Our goal is to simultaneously minimize the average FCT of all responses and congestion.

Let f_t denote the response flow starting at t^{th} timestep. We determine the initial congestion window for each response, denoted by a_t, according to the system state at that time, denoted by s_t. The decision process forms a trajectory $\{s_0, a_0, s_1, a_1, s_2, a_2, \ldots\}$. When flow f_t finishes transmission, we obtain its FCT, denoted by d_t. IW decisions are made according to a policy. Let θ denote the parameters of the policy, and then our problem is formulated as selecting the best θ to minimize the expected accumulated FCT for each state s_t:

$$\min_{\theta} \quad \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k d_{t+k}\right], \tag{1}$$

where γ is a discount factor in $(0, 1]$. If γ is zero, then the equation becomes $\min_\theta \mathbb{E}[d_t]$, which is to minimize the expected FCT of any flow. If γ is nonzero, then for $t = 0$ the term in bracket becomes $d_0 + \gamma d_1 + \gamma^2\ d_2 + \ldots$ It means the IW of flow f_0 not only affects its own FCT, but also affects FCT of subsequent flows with diminishing effect. Although only FCT appears in the above objective function, congestion also affects this objective. Since once a congestion happens, the FCT of suffered flows must increase a lot.

III. Overview of Method

In this section, we first give an overview of our method, and then discuss two techniques helping convergence.

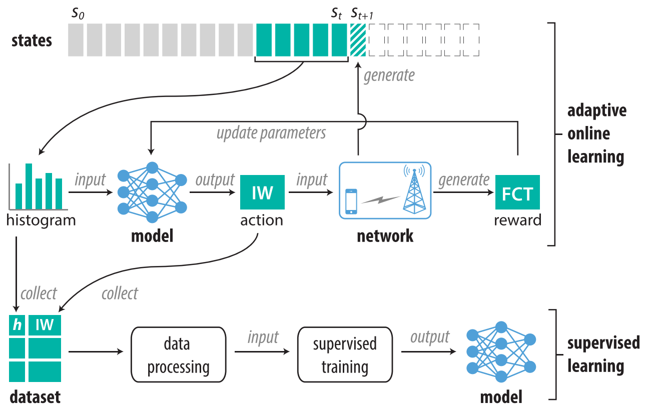


Fig. 2. The method consists of an adaptive online learning algorithm and a supervised learning.

A. Method

As shown in Fig. 2, our method consists of two sequential parts. The first part is an adaptive online learning algorithm to learn the best policy, which is the core of our method. It is based on the classic asynchronous advantage actor-critic (A3C) algorithm. As illustrated by the top part in Fig. 2, the neural policy model receives a histogram input computed from raw states and outputs an IW decision. Then in the network, a flow sets its initial congestion window as the IW given by the policy model and transmits data. This generates a flow completion time (FCT) as a reward signal and a new network state s_{t+1}. RL algorithm uses the FCT to update the parameters of the policy model. We will introduce the online learning algorithm in Section V and then further improve it by introducing the adaptive algorithm in Section VI.

As the online learning goes, our algorithm finds the best decisions under different network environments. During this process, we collect the histogram input and the action output to form a dataset. After some data processing, the dataset is used to train a new neural model to predict action output (IW) using supervised learning, as illustrated by the bottom part in Fig. 2. It becomes a classification problem and can be easily solved with a gradient descent algorithm. Finally, we obtain a static neural network that can output the best IW decision under various network environments. As such, we can avoid adaptive learning. Instead, our agent can make the best decision using the well-trained model. Note that here we can re-design a classification model instead of using the same model as in adaptive learning. Moreover, we can adopt several kinds of methods to improve the efficiency of making decision via neural policy, such as compressing model, running on CNN-friendly hardware etc. [16]. Several issues on supervised learning will be discussed in Section VII-C.

B. Mitigating Forward Dependency

Recall that to update the parameters of the model, we need training samples each of which consists of three elements: the system state s_t, the action a_t (which is IW) and the flow completion time d_t.

Ideally, d_t only depends on s_t and a_t, and the training algorithm will adjust the network parameters to penalize or encourage a_t based on whether d_t is large or small. However, d_t does not only depend on s_t and a_t, it also possibly depends on future actions a_{t+1}, a_{t+2}, *etc.* This is because the flow sent at time t does not complete immediately, it might be completed several time steps later, when action $a_{t+1} \ldots a_{t+k}$ have been taken. Those future actions might cause congestion in the network and affect the flow sent at time t, influencing d_t, thus tricking the training algorithm to incorrectly penalize or encourage a_t. During experiment, we found that it creates instability and hinders training.

To mitigate this forward dependency issue, we let an action persist for a period of time, so that its effect over FCT can be stably observed. Specifically, we reuse an IW decision for multiple flows, and update the IW decision periodically. This strategy is based on the assumption that flows in a short interval have similar system states.

C. Updating Policy Timely

The FCT's are not only delayed, but their delays are different. Such difference will bias the training, which impedes the model performance. Based on the analysis in the above section, after an action a_t is taken, one has to wait for its corresponding FCT d_t, and the time of waiting is exactly d_t. If d_t is large, such an adverse sample (which discourages a_t) will only arrive after many supportive samples (which encourage a_t) have been seen. As the training algorithm does not make use of adverse samples timely, it tends to be over-optimistic and thus slow down the convergence of training.

In order to address this issue, we propose a mechanism of *early feedback*, that is we estimate FCT for some flow before it finishes transmission. More specifically, when a flow sender starts transmission, it also starts a timer. Once the timer expires, but the transmission has not finished yet, then we estimate its FCT. Although there are many ways of estimation, we use a constant value for simplicity. If the value is large, then we penalize congestion much; while if the value is small, then we penalize congestion little.

Here, we have two hyperparameters in our algorithm: the timer and the estimation of early feedback. In our simulation, we set the timer to 100 ms (half of min RTO) and the estimation of FCT to 300 ms.

IV. Policy Function

We use neural network to represent the policy function, with system state s_t as the input and IW a_t as the output. Here, three important issues arise: 1) which factors are involved in the system state, and how to represent them; 2) how to formulate the IW output; 3) which architecture should be used for the neural network. We discuss each of them in below.

A. System State

We have two heuristic principles for selecting factors to represent the system state: 1) each factor is obtainable from the edge server alone; 2) each factor should relate to the congestion problem.

First, we find four factors describing our system: flow completion time, inter-arrival time, inter-departure time, and response size. Our system serves response flows by providing data transmission (*i.e.* service). A service starts when the edge server is about to send the first packet of a response; it finishes when the edge server receives the last ACK for the transmission. The service time in our problem is actually the flow completion time. In addition, the time between successive arrivals of responses, called inter-arrival time by convention, is also widely used in describing network status, and thus selected. Inspired by inter-arrival time, we think inter-departure time, that is the time between successive departures of responses, may be also useful. Lastly, the response size affects FCT (*i.e.* service time), and thus we include the size of the transmitted flows.

Second, we look at the factors expressing system performance. Besides the FCT mentioned above, RTT and throughput can also be measured by the edge server, and thus we include them as part of the system state.

For each of the above factors, we take the latest K samples and construct a histogram to describe their patterns over a period of time. The set of these histograms forms the system state s_t. Note that we use histogram instead of raw samples because it is more compact and histograms of different factors are easily comparable without normalization. Using histogram discards temporal information in the sample sequence, that is not a concern because these samples arrive in a short time during which the system load does not change much, so they can be safely treated as a snapshot of the system status. We assume that for a relatively stable system status, there is a corresponding optimal IW setting. As such, our model outputs an IW based on a snapshot, without considering temporal information between samples which reflect fluctuations in the system status. To compute the histogram for a factor, we first generate a sequence of bin edges. A simple way is to collect a set of samples in advance, compute a set of percentiles from it, and then make bin edges. Let $\{p_1, p_2 \ldots p_m\}$ denote a set of m percentiles, then we generate a set of $m+1$ bins as $\{(0, p_1), [p_1, p_2), \ldots, [p_{m-1}, p_m), [p_m, +\infty)\}$. In our simulation, we use i^{th} percentile where $i \in \{0, 10, 20 \ldots 100\}$ to make 12 bins. Recall that we have six variables, thus our histogram input forms an array of 12×6.

B. Policy Output

For the output, we can either output a value as IW or output a set of values as the probability distribution over a discrete set of IW. The former explores a continuous action space, while the later explores a discrete action space. We use the latter representation as the solution space is smaller, making the policy network easier to train.

Next, we discuss how we select this candidate set. For each response, IW is a positive integer bounded by the response size, which can be represented by counting its segments or bytes. Either way, the space of candidate IW is large. The size of action space has a significant effect on computational complexity. To reduce the complexity, we select 2^i segments as the candidate set. The i starts from 4 and increases to a value causing obvious congestion which can be found by brute force searching. In our simulation, we use

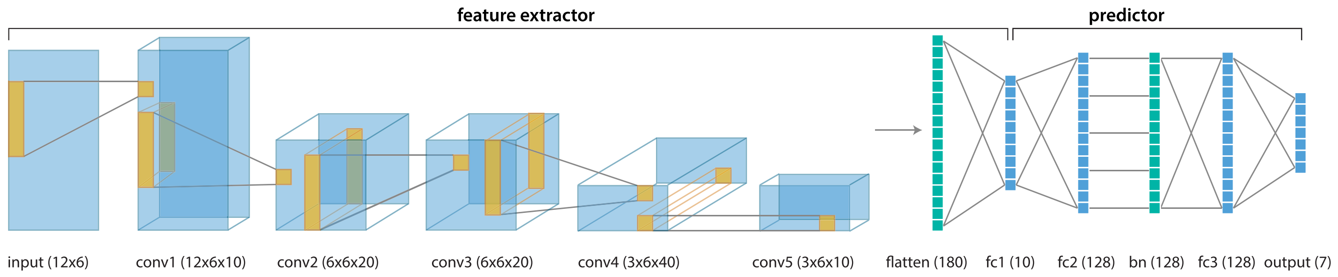


Fig. 3. The neural network used for policy function. The cubes in yellow represent kernels. "conv" represents convolutional layer, "fc" represents fully connected layer, and "bn" represents batch normalization. The number in each pair of brackets is the size of that layer.

$[10, 16, 32, 64, 128, 256, 512]$ as the set of candidate IW, where 10 is included as a standard setting proposed in [6].

Our design mimics the way traditional congestion control algorithms adjust a congestion window, which increases exponentially in the slow start phase. In a simple congestion control model without considering other factors causing packet drops, when no congestion happens, the difference of the two FCT's caused by two neighboring IW candidates 2^i and 2^{i+1} is one RTT. Thus, if the true optimal IW falls within $[2^i,\ 2^{i+1}]$, the optimal RTT and our computed RTT (with quantized IW) differs by one at most.

C. Neural Network Architecture

A neural network receives an input at one end, transforms it through a series of hidden layers, and produces a result at the other end. The output of a hidden layer is called activation. The activation of a layer is the input of the following layer. There are several types of hidden layer: fully connected layer, convolutional layer, and recurrent layer *etc.*

We design a neural network architecture as illustrated in Fig. 3. It consists of two parts: feature extractor and predictor. The feature extractor is responsible for extracting a feature from an input. It is composed of four 1D convolutional layers of 5×1 kernel, a 1D convolutional layer of 1×1 kernel, and a fully connected layer. Each layer has the following structure: 1) 10 kernels with a stride of one; 2) 20 kernels with a stride of two; 3) 20 kernels with a stride of one; 4) 40 kernels with a stride of two; 5) 10 kernels with a stride of one. Then, the output is flattened into a vector of 180 neurons, and is transformed to a vector of 10 neurons by a fully connected layer. The output here is the feature extracted from our input.

The second part of the neural network is a predictor with the feature as input. It is responsible for predicting action. It consists of two fully connected layers of 128 neurons and an output layer. The output is transformed to a probability distribution by using a softmax function.

We choose to use convolutional neural network (CNN) because histogram has spatial structure, which can be exploited by the CNN. We will show its advantage over a simple fully connected network in simulation. Moreover, CNN layers are pretty lightweight in terms of parameter count, all the convolutional layers combined only contributes 30% to the total parameter count. In addition, model compression techniques targeted at CNN structure can improve efficiency [16].

Reinforcement learning is a trial and error approach, so trying every choice without bias at the beginning of training is very important. Otherwise, we may fail to find the optimal choice due to lack of exploration. Thus, we use the technique of *batch normalization* in the policy function. This approximately forces all input to the prediction network to have zero mean and unit variance just after initialization, thus have a higher chance to lead to uniform output. It is sufficient to apply batch normalization to the second fully connected layer, as shown in Fig. 3.

V. Online Learning Algorithm

We have formulated the problem as an MDP. We use the classic asynchronous advantage actor-critic (A3C) algorithm [17] to compute the best policy.

Given a policy with a set of parameters θ, we can generate a trajectory $\{s_0, a_0, s_1, a_1, \ldots, s_t, a_t, \ldots\}$ and a corresponding set of FCT $\{d_0, d_1, \ldots, d_t, \ldots\}$. Recall that our problem is to find the best policy parameters θ such that the expected accumulated FCT is minimized at each timestep t, that is $\min_\theta \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k d_{t+k}\right]$. The discounted accumulated FCT is called *return* by convention. Let R_t denote it, that is $R_t = \sum_{k=0}^{\infty} \gamma^k d_{t+k}$.

Let $\pi_\theta(a|s)$ denote the policy function having parameters θ. It is the probability of taking action a under state s. We select action according to it. Let $V_\omega(s)$ denote the value function having parameters ω. For the value function, we adopt the similar neural network architecture as for the policy function. The only difference is that the last layer of the value function outputs a value instead of a probability distribution. Besides, we let the two functions share all the parameters of the feature extractor part. The A3C algorithm iteratively updates θ and ω until they converge.

We adopt the parallel learning architecture in A3C for stabilization. As shown in Fig. 4, it consists of a central agent, several subagents, and several environment instances. The central agent is responsible for maintaining the latest parameters θ and ω; each subagent is responsible for making IW decisions according to a policy function and computing the updates for θ and ω; all environment instances are the similar, which means they have similar system states. The parallel learning is possible in the real environment. First, according to a research on 3G/LTE mobile traffic [9] many cellular towers share the same traffic pattern related to their geographical locations. We believe this feature will be true

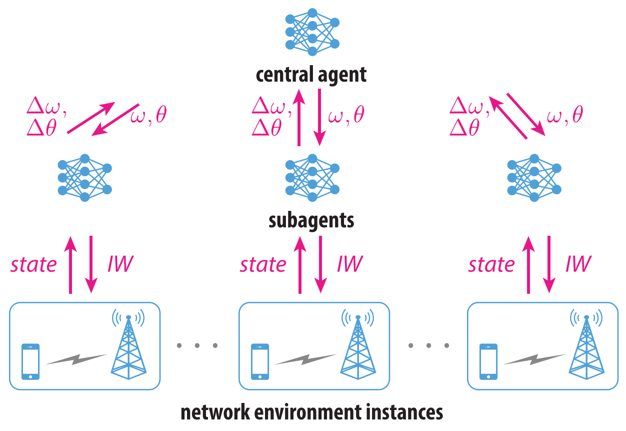


Fig. 4. The architecture of parallel training.

in 5G cellular networks as well. Second, collaborative MEC is a workable framework [2]. Each of the subagents can run on an edge server located with a cellular tower, and the central agent can run on any one of edge servers or a specialized server.

In this parallel architecture, when an edge server in an environment instance query the best IW for current situation, it sends the state to its associated subagent. The subagent then computes the best IW using its policy function, and returns the result immediately. After some steps, the subagent computes increments $\Delta\theta$ and $\Delta\omega$ from the past trajectory. The subagent then sends those updates to the central agent, who then applies those updates on the latest parameters θ and ω. This updating is asynchronous.

The A3C algorithm for each subagent starts from $t = 0$. At each iteration a subagent runs through the following steps:

1) reset increments: $\Delta\theta \leftarrow 0$ and $\Delta\omega \leftarrow 0$;
2) synchronize the subagent parameters (θ' and ω') from the central agent: $\theta' \leftarrow \theta$ and $\omega' \leftarrow \omega$;
3) interact with the environment using the current policy and collect a trajectory $\{s_t, a_t, \ldots, s_{t+T-1}, a_{t+T-1}\}$, and a corresponding set of FCT $\{d_t, \ldots, d_{t+T-1}\}$, where T is a hyperparameter larger than n (the parameter for n-step return);
4) compute n-step return $\bar{R}_{t+i}$ for all $i \in [0, T-1-n]$:

$$\bar{R}_{t+i} = \sum_{k=0}^{n-1} \gamma^k d_{t+i+k} + \gamma^n V_{\omega'}(s_{t+i+n}); \tag{2}$$

5) compute advantage A_{t+i} for all $i \in [0, T-1-n]$:

$$A_{t+i} = \bar{R}_{t+i} - V_{\omega'}(s_{t+i}); \tag{3}$$

6) compute the cumulative increment

$$\Delta\theta \leftarrow -\sum_{i=0}^{T-1-n} \nabla_{\theta'} \log \pi_{\theta'}(a_{t+i}|s_{t+i}) A_{t+i}; \tag{4}$$

7) compute the cumulative increment

$$\Delta\omega \leftarrow \sum_{i=0}^{T-1-n} A_{t+i} \nabla_{\omega'} V_{\omega'}(s_{t+i}); \tag{5}$$

8) perform asynchronous updates to those global parameters θ and ω: $\theta \leftarrow \theta + \eta_1 \Delta\theta$ and $\omega \leftarrow \omega + \eta_2 \Delta\omega$, where η_1 and η_2 are learning rate;
9) set $t \leftarrow t + T$;
10) repeat the above steps until reaching maximum iteration number.

In the above algorithm, we use n-step return to approximate R_t. As a result, the tail part of a trajectory is dropped in each iteration. To avoid this waste of training data, we can cache this sub-trajectory and use it in the next iteration. The above algorithm can be easily extended to implement this idea.

VI. Adaptive Learning Algorithm

As mentioned above, our algorithm starts with a uniform distribution over the action space using batch normalization, it converges as the trial and error process goes. The network environment may change afterward, and the performance of the policy model trained in old environment will likely degrade. To address this issue, we propose an adaptive algorithm. It detects the change in system environment and restarts learning on demand. To detect the change of environment, we monitor the input to the neural network, which is histogram. If a notable difference between consecutive histograms is observed, the environment is likely undergoing a significant change.

First, we use the change of the neural network input (histogram information) to detect the change of environment. We use cosine similarity to evaluate the similarity between two inputs, and define the difference between input $\mathbf{s}$ and old input $\mathbf{s}'$ as

$$\Delta\mathbf{s} = 1 - \frac{\mathbf{s} \cdot \mathbf{s}'}{\|\mathbf{s}\|\|\mathbf{s}'\|}. \tag{6}$$

The greater the value is, the larger the change is. To stabilize the algorithm, we use moving average over a window to capture the old state $\mathbf{s}'$.

The second task of our algorithm is to restart learning. A naive way is to re-initialize all the parameters in neural models, but this destroys too much information learned before. Since our purpose is to make the policy model to output an uniform distribution so that each action can be explored again, it is sufficient to re-initialize the parameters of the last two layers.

The adaptive algorithm runs at the central agent. At each iteration our algorithm proceeds as follows:

1) obtain the average state $\mathbf{s}$ in this iteration and the moving average state over past L iterations denoted by $\mathbf{s}'$;
2) compute $\Delta\mathbf{s}$ as above;
3) Once $\Delta\mathbf{s}$ is larger than a threshold, then we re-initialize the last two layers of the policy function and the value function.

Retraining is not a common practice in learning based algorithms, as it discards valuable information learned before, so additional justification is required. Here we justify the use of retraining. First, in our problem, training samples arrive as learning process goes, and we have no control over their order, so we cannot randomize them like what is typically done in offline training. Second, the system state usually remain stable

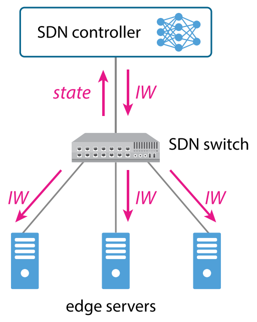


Fig. 5. An implementation of our method in a SDN-based MEC system.

for quite a while before significant change, as suggested in [18] traffic changes in the scale of hour. As such, our learning algorithm feeds lots of similar samples to the model in a long duration and results in overfitting. When new kinds of samples arrive later, the overfitted model locked in a local minimum has difficulty to forget the past and learn again. Last, but most importantly, overfitting has a severe impact on exploration when system state changes. In our learning, we explore IW candidates according to the probability $\pi_\theta(a|s)$. Then, an overfitted model, which has tiny or even zero entropy, always chooses the outdated best IW, which leads to poor performance in new state, and seldom chooses the other IW candidates. Due to low exploration, the adaptation to new system state is either very slow or impossible, basically depending on how small the entropy is. Overall, we need to manually introduce entropy into the model when necessary, and we found that retraining the last few layers of the model strikes a good balance among simplicity, robustness and effectiveness.

VII. Implementation and Discussion

In this section, we discuss how to implement our method in SDN-based mobile networks and in a realtime manner. In addition, we discuss some issues on supervised learning.

A. SDN-Based Implementation

We have challenges in implementing the method in MEC systems. The first is how to obtain system state as a global view of whole MEC system. For a MEC system having a single edge server, the system state observed by that server is the global view. However, in the practical scenario with multiple servers, the load is distributed among them, and then the system state observed by each server only represents a local view. Those local views may not agree with each other about the system load due to load imbalance. The second challenge is how to enable neighboring MEC systems to collaborate in parallel learning.

We propose to adopt SDN to address the above challenges and make the implementation easier. SDN separates the control from data plane and enables intelligent network management and service orchestration. Researchers have proposed to utilize SDN to solve the complexities in MEC systems [3], [4]. In SDN-based MEC systems, we can easily implement our method. As shown in Fig. 5, edge servers are connected to a SDN switch, and the switch are connected to a SDN controller. The switch is responsible for collecting system states and rewards, and the controller is responsible for learning the neural policy by the adaptive online algorithm. The switch collects data and periodically sends it to the controller; the controller computes the best IW according to the collected state and pushes it back; the switch then publishes the latest IW to all edge servers connected; each edge server then transmits flows using the new IW. The controller also periodically updates the neural policy using the collected data.

SDN can also enable collaborative learning among multiple MEC systems. Several types of SDN multicontroller architectures have been proposed, as introduced in a recent survey [19]. To implement parallel learning, we can adopt a hierarchical design with two layers. The lower layer contains local controllers, each of which controls a MEC system, while the upper layer contains the root controller, which manages all the lower layer. We let each local controller run a subagent and the root controller run the central agent. As discussed in the parallel learning, each subagent at each local controller performs asynchronous updates to the central agent at the root controller.

B. Realtime Implementation

Our algorithm can be implemented efficiently to run in realtime. The system consists of three parts which can operate concurrently: 1) learning and updating model parameters; 2) inferring IW based on system state; 3) responding to IW query. The first thread is responsible for collecting trajectory and computing local model update increments, sending those increments to the central agent, receiving latest model parameters from the central agent and updates the local model. The second thread feeds the latest system state into the model and computes the IW, and then caches the computed IW. Each iteration takes about 1ms, measured in our simulation. These two threads run in the background and they run as fast and as frequently as they can. The third thread runs in the foreground and replies to IW query instantly with the cached IW. In this way, the delay experienced by the client is negligible. Note that although the IW is only updated by the second thread periodically, it is fast enough as the system state is unlikely to undergo any significant change in 1ms.

In our online learning algorithm, the edge server needs to query IW frequently so that sufficient data is collected for learning. However, once we obtain a static model by supervised learning, then the edge server does not need to query IW frequently any more, since mobile traffic changes at the scale of hour as suggested in [18].

C. Discussion on Supervised Learning

We mentioned supervised learning in the overview of our method. There are several issues worthy of more discussion. First, the dataset may include some data having a bad label, *i.e.* action output, resulting in poor FCT, for instance, those data collected at the start of learning before convergence. We can correct those labels in a simple manner. As mentioned in the adaptive learning algorithm, by monitoring state difference we can divide a duration into several intervals, each having a

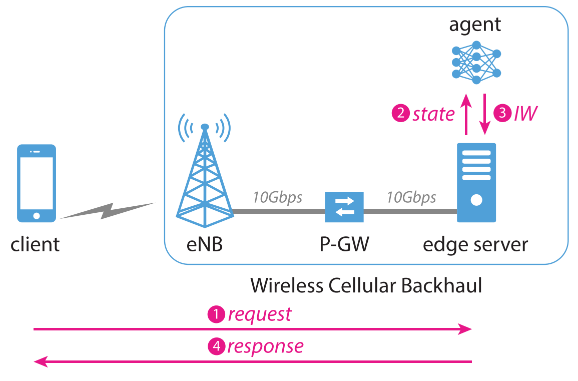


Fig. 6. The architecture of 5G mmWave MEC simulator and the process flow.

steady network environment. For each interval, we select the most frequently used IW (which must be the best) as the label for all data within that interval.

Second, some similar inputs may correspond to two or more different action outputs, since all of them are almost the optimal solution and result in the same performance. It is better to assign a single label to those similar inputs. Thus, we propose a simple way to do this. We evaluate the average state of each interval and compare with each other. For those intervals having a similar state, which can be evaluated by using state difference and some threshold, we assign a single label to the data within those intervals by selecting the action leading to the lowest FCT. As such, the dataset is prepared well for supervised learning.

VIII. Simulation Setup

We simulate and evaluate our method extensively. We compare our method with a naive method where IW is fixed to 10 segments (IW-10 for short) and also with an existing dynamic IW setting algorithm called SmartIW [7], [8] used in search engines.

Here, we briefly introduce the SmartIW method. Its core idea is using discounted Upper Confidence Bound (UCB) algorithm to dynamically find the best IW so that total reward is maximized. The reward is defined as a weighted sum of throughput ratio and RTT ratio. According to their work, this method periodically restarts the algorithm at the timescale of minutes so as to follow the dynamic traffic. In addition, we use the same IW candidates as in our method for the purpose of comparison. The other parameters are set according to [7], [8]. Besides, it is unclear how to set the discount factor in their work, and we set it to 0.99 in our simulation.

We build a simulator as shown in Fig. 6. It is mainly composed of three components: a client, an edge server, and an agent. The client and the edge server communicate through a 5G mmWave cellular network. We measure that its downlink throughput can reach 2.8 Gbps under continuous data transfer, which is consistent to the result provided in [20]. The edge server is connected to the P-GW (packet gateway) in the backhaul network of a cellular system without disturbing the existing system as suggested in [21]. The bandwidth between the P-GW and the server is 10 Gb/s and the delay is 50 μs. Thus, bandwidth bottleneck locates at wireless links, and congestion may happen at eNB. In our simulator, the agent runs on the edge server. The simulator does not use the SDN-based architecture introduced in Section VII-A for simplicity, but this does not affect performance evaluation. We build the simulator based on the network simulator ns3, and use the module of 5G mmWave network in [20]. We implement our agent by using TensorFlow in python.

In our simulator system, the client sends a request of 1 KB to the edge server. Once the server receives the request, it sends a response back to the client. The size of the responses are uniformly distributed between 2 KB and 1024 KB. Before the server is about to send a response, it asks the agent which IW is the best with some state information transmitted as input. The agent then computes the best IW according to those information and returns the value back. We simulate the arrival of the client requests as a Poisson process. In other words, the inter-arrival time of client requests follows exponential distribution. We vary its mean to generate different bursts and traffic loads.

In our simulation, the eNB locates at coordinate (0, 0) m and is at the height of 30 m. The client initially locates at a random position within a square area of 600 m^2 centered at (0, 0) m, is at the height of 1.5 m, and moves at 25 m/s, a typical vehicle speed, with a random direction within the area.

In addition, we set ns3 simulator as follows:

1) use TCP Cubic as transport protocol;
2) set minimum retransmission timeout (RTO) to 200 ms as in Linux;
3) set the TCP maximum segment size to 1024 B;
4) set TCP maximum transmit/receive buffer size to 1 MB;
5) set queue size of each NetDevice to 256 kB.

In our simulation, if not stated otherwise, the candidate set of IW is (10, 16, 32, 64, 128, 256, 512) and hyperparameters are set as follows:

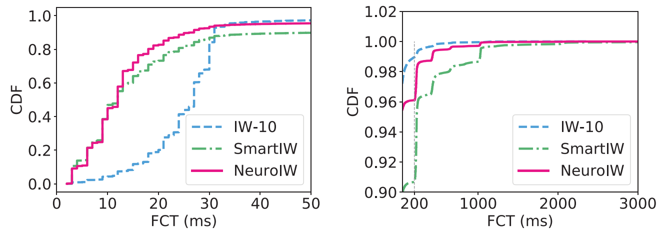
1) the timer of early feedback (introduced in Section III-C) is set to 100 ms, half of min RTO;
2) the approximation of early feedback is set to 300 ms;
3) the number of samples K for computing histogram input (introduced in Section IV-A) is set to 50;
4) the discount factor in the objective function is set to 0.9, that is we consider an action approximately affects the performance of 50 flows in future at most;
5) the reuse factor is set to 10 (introduced in Section III-B);
6) step size η_1 and η_2 are set to 0.0008 and 0.002 respectively. Our experience is first to find a suitable η_2 so that the advantage $\bar{R}_t - V_\omega(s_t)$ reduces fast and fluctuates near zero, and then to set η_1 roughly half.

IX. Performance Evaluation

In this section, we demonstrate through simulation that our adaptive online initial window decision method can simultaneously reduce flow completion time and limit congestion effectively under dynamic traffic. In the following simulations, our method uses 8 subagents to learn in parallel. For other methods, we run simulations for 8 times with different random seeds and take average results.

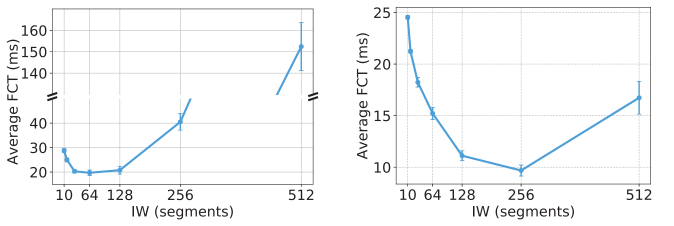
TABLE I
FCT (ms) FOR DIFFERENT METHODS. THE PERCENTAGES IN EACH BRACKET ARE THE REDUCTION RATIO COMPARED TO IW-10

Load level		IW-10	SmartIW	NeuroIW
High	Mean	29.48	54.14 (-84%)	26.61 (10%)
	Median	27.00	12.00 (56%)	12.00 (56%)
Low	Mean	24.56	14.89 (39%)	11.50 (53%)
	Median	27.00	10.00 (63%)	9.00 (67%)



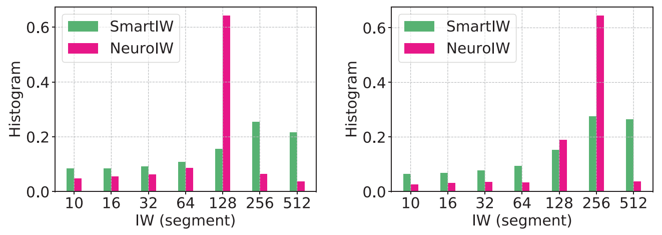
(a) (b)

Fig. 7. The empirical CDF of FCT under a high load. (a) less than 50 ms; (b) greater than 50 ms.



(a) (b)

Fig. 8. Average FCT for different IW settings, where the bars around the symbols represent the 95 percent confidence interval. (a) high load; (b) low load.



(a) (b)

Fig. 9. The histogram of initial congestion window. (a) high load; (b) low load.

A. Reduction of Flow Completion Time

We first investigate the effectiveness of our method under different levels of static load.

1) High Load: We set the mean inter-arrival time of client requests as 5 ms, and forms a high load for the simulation system. First, we use FCT as a metric to compare our method with IW-10 and SmartIW. We show the results in Table I. It is seen that our method (NeuroIW) reduces average FCT by 10% and median FCT by 56% respectively compared to IW-10 method. This suggests that our method can reduce FCT significantly. In comparison, SmartIW method degrades the mean FCT severely.

Moreover, we show the empirical cumulative distribution function (CDF) of FCT in Fig. 7. To illustrate it clearly, we divide its x-axis into two parts: the part less than 50 ms and the part greater than 50 ms. In Fig. 7a, it is shown that our method performs slightly better than SmartIW for the flows whose FCT is small, and both methods are significantly better than IW-10. From Fig. 7b, it is observed that 4% of the flows experience congestion in our method, whose FCT is beyond 200 ms (min RTO). It is slightly larger than 1 percent when IW is 10. However, near 10% of the flows experience congestion in SmartIW, which is greatly larger than that in the other methods. This explains why SmartIW has the same median with our method but has a poor mean performance.

Next, we analyze why our method is better than others. First, we find the best IW by a brute force iteration through all candidate IW's. As shown in Fig. 8a, FCT is a convex function of IW. Each of three candidates (32, 64, and 128) achieves the lowest FCT approximately, while 256 and 512 result in high FCT.

Then, we show the histogram of IW for various approaches in Fig. 9a. It is observed that for our method over 80% of the flows use any one of the best IW candidates. In comparison, for SmartIW the distribution is more even and the IW choices of 256 and 512 occupy over 40 percent in total. As Fig. 8a shows, these two candidates result in high FCT. This suggests that SmartIW cannot avoid aggressive choices, which is the reason for severe congestion and high average FCT.

Finally, we analyze where the benefit comes from. The flows are binned into 11 groups based on flow size. The flow size of the first group is less than 10 kB, the second group is between 10 kB and 100 kB, the third group is between 100 kB and 200 kB, etc. We show the mean FCT per class in Fig. 10a. By comparing our method and IW-10, it is observed the flows with medium size (between 100 kB and 900 kB) experience FCT reduction. The reduction ranges from 2.4 ms to 7.6 ms. The shorter flows (less than 100 kB) are affected by this and suffers longer transmission time.

2) Low Load: We also evaluate the performance under a low load. We set the mean inter-arrival time of client requests as 50 ms. The results are shown in Table I. It is seen that our method achieves the best result again. It reduces mean FCT by 53% and reduces median FCT by 67% compared to IW-10 method. In comparison, the reduction ratio of SmartIW is lower. Moreover, we illustrate the empirical CDF of FCT in Fig. 11. In this case, our method performs slightly better than SmartIW both in the tail and the median performance.

We also analyze why our method is better than others. We simualte each IW candidate and find that 256 is the best choice in this case, as shown in Fig. 8b. Then, we show the histogram of IW for these methods in Fig. 9b. It is observed that when using our method over 80% of the flows use IW of either 128 or 256 (the best two candidates). In comparison, when using SmartIW only 40% of the flows use those best candidates, while the other 60% of the flows use the other candidates resulting in high FCT.

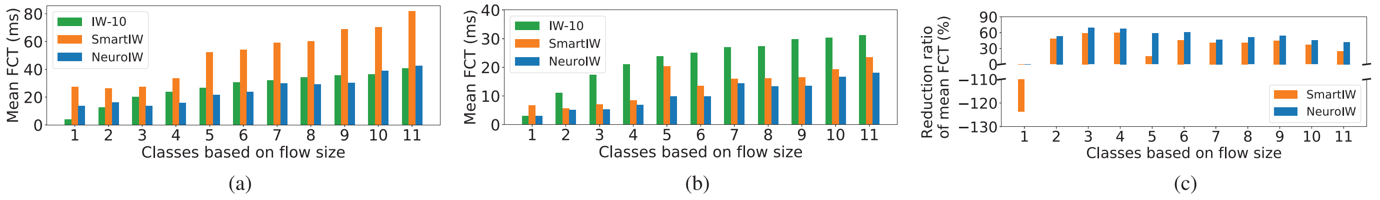


Fig. 10. Performance per flow class based on flow size. (a) FCT under a high load; (b) FCT under a low load; (c) FCT reduction ratio under a low load.

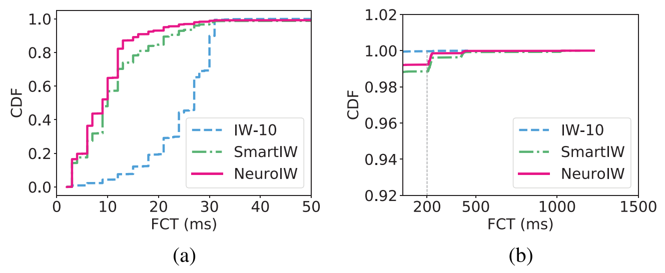


Fig. 11. The empirical CDF of FCT under a low load. (a) less than 50 ms; (b) greater than 50 ms.

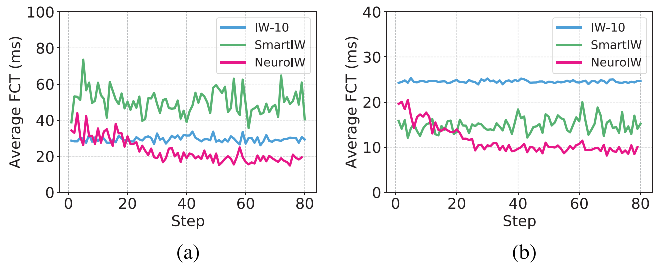


Fig. 13. The average FCT as a training process goes. (a) high load; (b) low load.

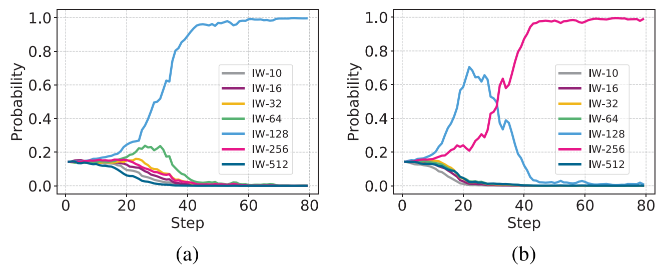


Fig. 12. The probability of selecting each IW choice as a training process of our method goes. (a) high load; (b) low load.

Moreover, by comparing the histogram of IW for SmartIW method under two levels of load in Fig. 9, we find that there is no much difference between the two distributions. This suggests that SmartIW does not adapt itself to different traffic. In comparison, we observe that our method has very different IW distributions under two levels of load, which suggests that our method has an adaptation capability.

We also show the mean FCT per class in Fig. 10b. By comparing our method and IW-10, it is observed that all flows larger than 10 kB experience FCT reduction, 18 ms at most. Then, we illustrate the reduction ratio in Fig. 10c. It is observed that the reduction ratio is significant, 69 percentage at most.

B. Convergence of Training Algorithm

Next, we illustrate how our method converges to the best IW as a training process goes. In Fig. 12, we show how the probability of selecting each IW choice changes as a training process goes. Each step corresponds to 8 subagents and a batch size (100 flows) per subagent. The value of each step is averaged over total 800 flows.

For a high load, as shown in Fig. 12a, initially we have a uniform probability. As the training process goes, the probability of selecting 128 (one of the best candiates) increases gradually and reaches 1 after about 40 steps; while the probability of selecting any other candidate decreases and reaches 0 finally. The similar phenomenon appears under a low load as shown in Fig. 12b. In this case, our algorithm converges to 256, which is the best candidate under a low load.

We also show the average FCT as a training process goes in Fig. 13. It is observed that after around 30 steps, our method converges to the lowest FCT. For high load, it takes around 15 seconds; for low load, it takes around 150 seconds. No matter which case, it is sufficiently short. However, for any other method the performance fluctuates without convergence.

C. Adaptation to Dynamic Load

Next, we simulate a dynamic load where the mean interarrival time of client requests alternates between 50 ms (low load) and 5 ms (high load), and demonstrate that our algorithm can adapt to the dynamic load. In our simulation, each type of load lasts for 8000 requests, and two types of load alternate twice. For our algorithm, we set the threshold of the state difference to 0.05.

1) Effectiveness of Our Algorithm: First, we evaluate the performance using FCT as shown in Fig. 14a. A point at each step in the figure is the average value over 8 subagents and 100 flows per subagent. It is shown that the FCT drastically changes around every 80 steps. At the 80th step and the 240th step, the system transforms from a low load to a high load. At those points, the FCT increases drastically, worse than 30 ms. However, our algorithm captures those changes, and then the FCT reduces gradually and reaches a low value after a short time (around 30 steps). Moreover, it is observed that our adaptive algorithm obtains the best performance under each type of load. We get 10 ms under a low load and 20 ms under a high load.

Second, we analyze how our algorithm adjusts IW as the dynamic load goes. As shown in Fig. 14b, it is observed that

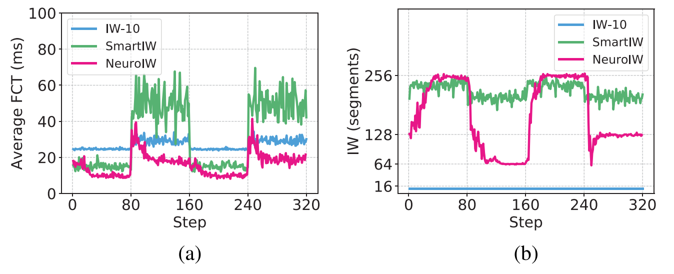


Fig. 14. Average performance under a dynamic load. (a) FCT (ms); (b) IW (segments).

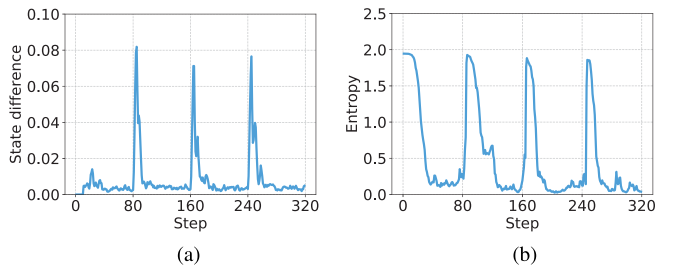


Fig. 15. Interesting variables under a dynamic load. (a) the state difference $\Delta\mathbf{s}$; (b) average entropy.

IW changes closely following the dynamic load. In the first phase the IW increases and reaches 256 (the optimal solution under a low load); in the second phase it reduces quickly to 64 (an optimal solution under a high load); in the third phase it increases and reaches 256 again; in the final phase, it drops rapidly and reaches 128 (another optimal solution under a high load).

Recall that we detect the change of environment by monitoring the difference between consecutive histograms as discussed in Section VI. We evaluate how the state difference changes during simulation, and show the result in Fig. 15a. It is observed that three spikes appear in the simulation with an interval of near 80 steps, which is consistent with the load changes. Furthermore, we observe that the peak values are larger than the threshold set previously, and thus it activates the re-learning process. Moreover, we illustrate the average entropy in Fig. 15b. It is shown that the entropy is close to 2 (the entropy of uniform probability) at the beginning, and then it drops gradually as the learning process goes. At the beginning of each new interval, it rockets to the starting value since the re-learning is activated by our adaptive method, and then it drops again.

2) Comparison With Other Methods: In this section, we compare our algorithm with the other methods (IW-10 and SmartIW). We let SmartIW restart itself every 2000 requests (shorter than the load interval of 8000 requests) so that it has chances to adjust itself under a new load.

As shown in Fig. 14a, it is obvious that our method achieves the lowest FCT during the whole four load intervals. Specifically, for SmartIW, it performs slightly worse than our method under a low load, and performs very bad under a high load (in the second and the fourth load intervals). As shown in Fig. 14b, it is observed that SmartIW adjusts IW in the

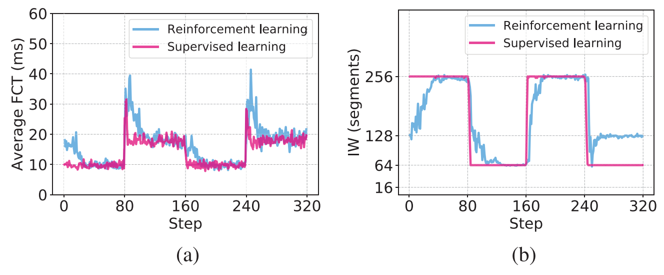


Fig. 16. Performance comparison of a supervised-learned model and the adaptive online learning algorithm. (a) FCT (ms); (b) IW (segments).

similar way under both types of load. This demonstrates that SmartIW cannot perceive the load change and neither adapt itself.

D. Supervised Learning

As mentioned in our method, we can improve the effectiveness and efficiency by collecting data during online learning and run a supervised learning later on. In this simulation, we collect data from the above simulation in Section IX-C and obtain 2.5×10^5 samples in four phases. There are bad labels in each phase. Besides, the second and the fourth phases have similar state, but have different actions. Thus, we process the dataset as introduced in Section VII-C. As a result, the samples in the first and the third phases are labeled with 256; while the samples in the other phases are labeled with 64.

We use the same neural network architecture as shown in Fig. 3. We obtain a model with accuracy above 99.4% after training for one epoch. Then, we test its performance under the same dynamic traffic as in the above simulation. We run the test eight times and get the average results. Fig. 16a and Fig. 16b show the change of FCT and the change of IW respectively, and also show the comparison with the result achieved by the adaptive online learning algorithm. It is observed that the supervised-learned model makes correct IW decisions under dynamic traffic. Moreover, its adaptation is more responsive.

E. Evaluation With LTE Mobile Traffic Distribution

In this section, we evaluate our method with an empirical distribution of TCP flow sizes in LTE networks as shown in Fig. 17, which is studied from a real-world LTE packet trace in [9]. This distribution exhibits strong heavy-tail characteristics, 95% of flows are less than 85.9 kB and the top 0.6% of flows ranked by payload sizes, each with over 1 MB payload, account for 61.7% of the total downlink bytes. Next, we set inter-arrival time of client requests following exponential distribution with mean as 2 ms, 1 ms, and 500 μs. As such, we can generate three kinds of traffic. In Fig. 18, we illustrate the estimated throughput (tested per interval of 10 ms) when we set mean inter-arrival time to 2 ms, 1 ms, and 500 μs in turn. As such, we can mimic the mobile traffic patterns, characterized by peak-valley feature, observed in 3G/LTE cellular networks [18]. We test and show the concurrency of flows and the CDF of instant throughput in Fig. 19.

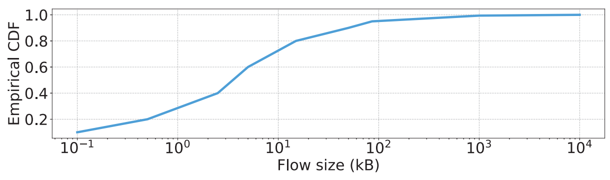


Fig. 17. An empirical distribution of TCP flow sizes in LTE network.

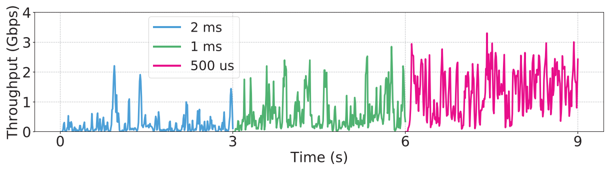


Fig. 18. The throughput vs time. The throughput estimated when we set mean inter-arrival time to 2 ms, 1 ms, and 500 μs in turn.

In our simulation, we set each type of traffic last for ten thousands requests. We use [10,16,32,64,128,256] as IW candidates and set the threshold of the state difference to 0.02. We show the change of average FCT in Fig. 20a. The same as above, our method achieves the lowest FCT during all three load intervals. SmartIW performs bad under medium load and high load (the last two load intervals). From Fig. 20b, it is observed that SmartIW adjusts IW in the same way no matter under which type of load. However, our method adapts to load. As load becomes high, our method reduces IW adaptively.

Next, we analyze the performance under each type of load. The same as above, we also classify flows based on flow size and analyze FCT per class. Here, we have 12 groups. Except the first 11 groups similar as before, we add a new including the flows larger than 1 MB. We show mean FCT per flow class in Fig. 21 and reduction ratio per flow class in Fig. 22. First, we observe that for all types of loads, all flows with medium size (between 10 kB and 1 MB) experience FCT reduction. The reduction is over 10 ms for a majority of groups and reaches 20 ms for some. The reduction ratio is over 30% for a majority of groups and reaches 50% for some, which is very significant. For the short flows less than 10 kB, their FCT exhibit increasing in different levels. It is because these flows are affected by other flows with large IW.

Lastly, we analyze the performance evaluated over all flows. Under low load, NeuroIW and SmartIW achieve comparable reduction of 22% over IW-10. As shown in Fig. 23a, we observe that both methods use very similar IW distribution during this phase. Under medium load, our method is better than SmartIW by around 7%, and better than IW-10 by around 9%. As shown in Fig. 23b, our method uses lower IW than SmartIW. Under high load, our method is significantly better than SmartIW by around 45% and comparable with IW-10. As shown in Fig. 23c, our method uses much lower IW than SmartIW.

F. Necessity of Adaptive Algorithm

In our method, we propose an adaptive learning algorithm to detect the change of environment and restart learning accordingly. Here, we demonstrate its effectiveness. We use the same settings as in Section IX-C and [16, 64, 256, 512] as IW candidates. We show results in Fig. 24. It is observed that without adaptive algorithm entropy drops quickly as the training process goes and reaches zero at the 80th step. As a result, the RL algorithm loses exploration ability, and IW is stuck at 256, which is a bad choice for the following high load. In comparison, by using adaptive algorithm, the learning is restarted when system traffic changes. The lower FCT is achieved after re-learning.

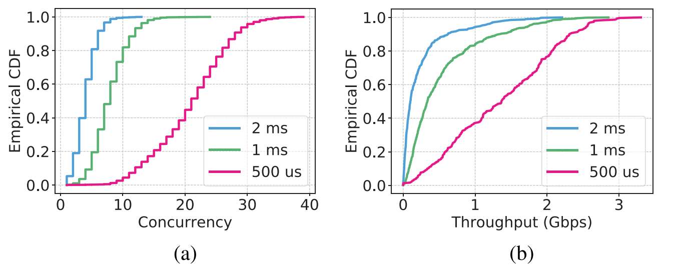


Fig. 19. Traffic characteristics. (a) The concurrency of flows; (b) The CDF of instant throughput.

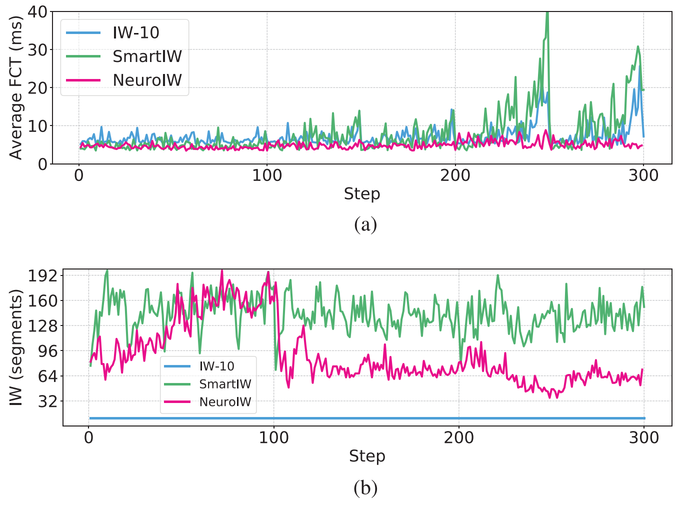


Fig. 20. Average performance under LTE mobile traffic. (a) FCT (ms); (b) IW (segments).

G. Evaluation of Policy Function

Here, we verify that the design of policy function is reasonable. In our design, we take the latest K samples and construct a histogram as the input of a 1D CNN neural model. First, we verify whether 1D CNN performs better than a simple neural network (SNN). We let network input and output unchanged, but use only one 128-neuron hidden layer with batch normalization used. We evaluate this method in the same environment as above (Section IX-F), and show comparison results in Fig. 25. Note that, since we change the policy architecture, we have to set the learning rate differently for SNN: actor learning rate is 0.001, and critic learning rate is 0.0026. From Fig. 25a, we observe that our method (histogram CNN) obtains lower FCT than SNN. From Fig. 25b, we observe that the algorithm with SNN does not converge to the best IW. We also show the RL advantage value (introduced in Section V) in Fig. 25c, which should reduce first

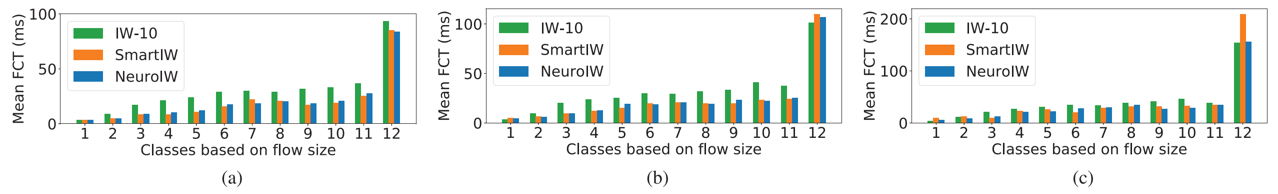


Fig. 21. Mean FCT per flow class based on flow size under LTE mobile traffic. (a) low load; (b) medium load; (c) high load.

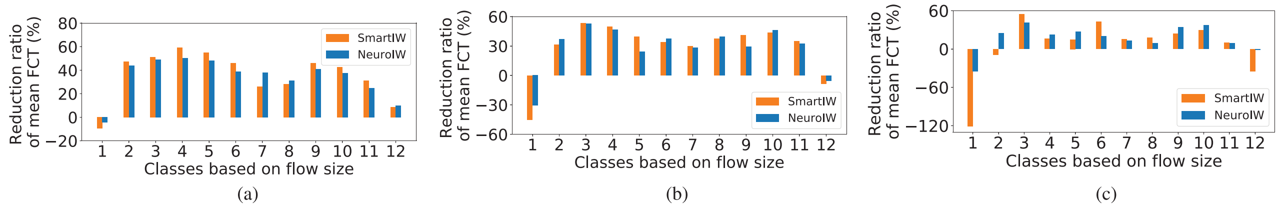


Fig. 22. The reduction ratio of mean FCT per flow class based on flow size under LTE mobile traffic. (a) low load; (b) medium load; (c) high load.

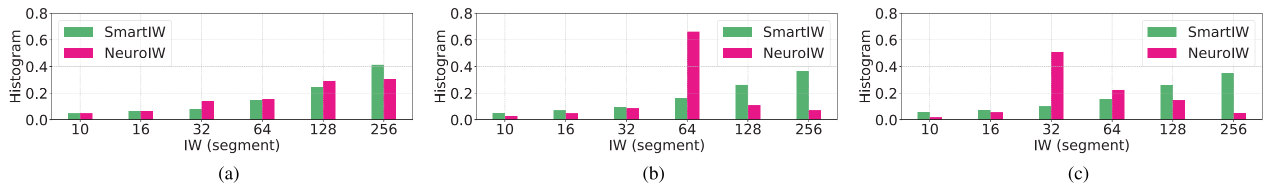


Fig. 23. The histogram of initial congestion window under LTE mobile traffic. (a) low load; (b) medium load; (c) high load.

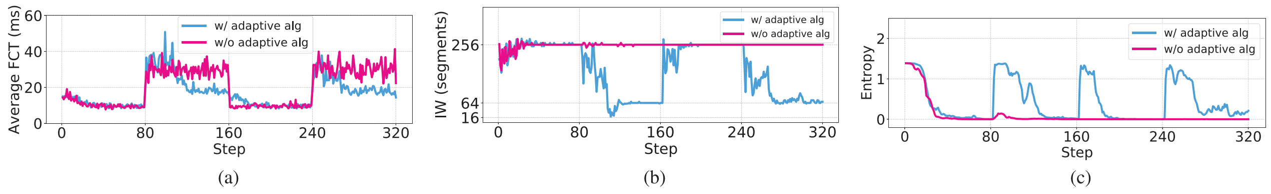


Fig. 24. Comparison with and without adaptive algorithm. (a) FCT; (b) IW; (c) entropy.

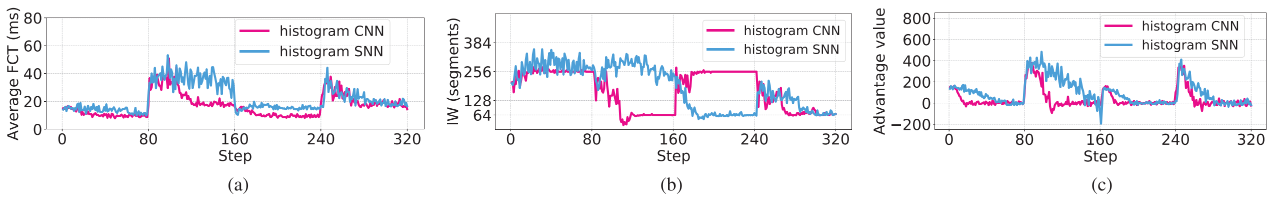


Fig. 25. Evaluation on various policy functions. (a) FCT; (b) IW; (c) advantage value.

and then fluctuate near zero. It is observed that in the situation of SNN the value drops as expected, which demonstrates that hyperparameters are set properly. Moreover, it is observed that our method is more stable than SNN case.

We also evaluate the policy of inputting the latest K samples into the above SNN. They achieve similar results as shown in Fig. 26. This shows that our framework is not very sensitive to the choice of the underlying neural net model. However, using raw samples as input has a major drawback: it couples the neural net architecture with the number of input samples K. The first layer of the SNN has $K \times D \times N$ parameters, where D is the dimension of a single raw sample and N is the number of neurons in the first hidden layer. As K increases, the parameter count of SNN expands quickly. In our case, the first layer of SNN has more than 38k parameters ($K = 50$, $D = 6$, $N = 128$), while the entire CNN has less than 30k parameters. In addition, using raw samples is inflexible because K has to be fixed before training, it is impossible to vary K during training. This hinders certain application, for example, using raw samples in a fixed time window as input. Also, it makes model reuse difficult as models trained with different K are incompatible. As such, we advocate the use of histogram instead of raw samples as input.

H. Evaluation of K

We also evaluate the impact of the number of samples used to generate histogram input, that is K. We set its value to 20, 50 (default value), 128 and compare their performance.

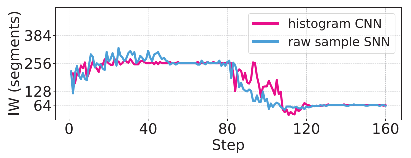
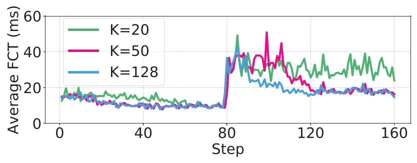
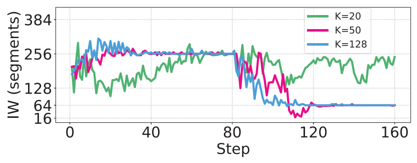


Fig. 26. Comparison of our method and SNN policy with input of raw sample.



(a)



(b)

Fig. 27. Evaluation on the impact of the number of samples for computing histogram input. (a) FCT; (b) IW.

The simulation environment is the same as above. The results are shown in Fig. 27. It is observed that with K being 128, the learning algorithm converges faster than default case; with K being 20, the learning algorithm converges slower than default case in the first phase and does not converge to optimum in the second phase. The reason is that when K becomes larger, histogram input becomes more stable and improves learning; while when K becomes very small, histogram input becomes unstable and degrades learning.

X. Related Works

We review the existing works related to our problem, and categorize them into five groups as follows.

A. MEC and SDN-Based MEC

Tran *et al.* in [2] surveyed three representative use-cases of MEC including mobile-edge orchestration, collaborative caching and processing, and multi-layer interference cancellation. They demonstrated the benefits and applicability of MEC in 5G networks. Baktir *et al.* in [3] made an extensive survey to demonstrate that SDN has capability to solve the complex design problems in MEC systems. The SDN-enabled edge Computing can provide intelligent network management including service discovery, service commissioning and migration, performance tuning and optimization, and user handover. Huang *et al.* in [4] proposed and implemented a SDN-based MEC framework, which achieved a significant latency reduction.

B. Increasing Initial Congestion Window

Some researchers have proposed to increase initial congestion window to reduce the latency of HTTP responses [5], [6] and the flow completion time of high performance computing traffic in software defined networks [22]. However, IW is still a fixed value in these works. Recently, Nie *et al.* in [7] proposed to dynamically set initial window with reinforcement learning to reduce web latency. Its core idea is using discounted Upper Confidence Bound (UCB) algorithm to dynamically find the best IW so that total reward is maximized.

C. Machine Learning Based Congestion Control

Some researchers proposed to use machine learning to address internet congestion control. Remy [23] is the first notable computer-generated congestion control method. In this approach, the designer specifies some assumptions about the network and a performance optimization objective, then Remy produces the congestion control rules. PCC [24] is another congestion control architecture in which each sender continuously observes the connection between its sending rates and empirically experienced performance, enabling it to adopt the ones leading to the largest utility. PCC Vivace [25] is a very recent work based on PCC. It employs provably optimal online optimization based on gradient ascent to achieve high utilization of network capacity, swift reaction to changes, and fast and stable convergence.

D. RL-Based Congestion Control

Research community have proposed to use reinforcement learning to address internet congestion control, which enables a network to automatically control itself by learning from experience. QTCP [26] adopts a Kanerva coding to approximate value functions. Their simulation results show that QTCP can achieve higher throughput while maintaining low transmission latency. Three other works [27]–[29] adopt deep neural network as the policy function for congestion control. Aurora [27] suggests that DRL-based protocol can distinguish non-congestion loss from congestion-induced loss, and adapt to variable network conditions. TCP-Drinc [28] adopts LSTM to handle the correlation in time series. DRL-CC [29] is a design for multi-path TCP congestion control. It utilizes a recurrent neural network for learning a representation for all active flows.

E. Deep RL-Based Resource Management

Some researchers have proposed to use deep reinforcement learning to address challenging resource management problems. Mao *et al.* in [12] proposed to use DRL to generate adaptive bitrate algorithms automatically to optimize user quality experience. Chinchali *et al.* in [14] proposed to use DRL to optimally schedule delay-tolerant IoT traffic in cellular networks. They demonstrated that their scheduler can enable mobile networks to carry more data with minimal impact on existing traffic. Li *et al.* in [15] proposed to use DRL to solve parameter tuning for storage performance optimization, which is slow and costly in practice. They evaluated in a file system and demonstrated a significant increase in I/O throughput. Mirhoseini *et al.* in [13] proposed to use DRL to optimize device placement for TensorFlow computational graphs, that is to predict which subsets of operations should run on which of the available devices.

XI. Conclusion

In this paper, we investigate the IW decision problem in mobile edge computing, that is to adaptively adjust initial congestion window such that flow completion time is optimized, while congestion is minimized. We propose an adaptive online decision method to solve the problem, which learns the best policy (a neural network function) using deep reinforcement learning. We propose several techniques to ensure our algorithm can converge stably and fast. To further improve the responsiveness and efficiency of IW decision, we propose an approach based on supervised learning, which first collects data during online learning, and then trains a policy with the processed data. We also propose a SDN-based implementation of our method. Our simulations in a 5G mmWave MEC simulator demonstrate that our algorithm can effectively reduce FCT with little congestion caused. Moreover, it can adapt IW to dynamic network conditions. We also demonstrate the effectiveness of our policy design and some parameter setting.

References

[1] B. Liang, *Mobile Edge Computing*. Cambridge, U.K.: Cambridge Univ. Press, 2017, pp. 76–91.

[2] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, "Collaborative mobile edge computing in 5G networks: New paradigms, scenarios, and challenges," *IEEE Commun. Mag.*, vol. 55, no. 4, pp. 54–61, Apr. 2017.

[3] A. C. Baktir, A. Ozgovde, and C. Ersoy, "How can edge computing benefit from software-defined networking: A survey, use cases, and future directions," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2359–2391, 4th Quart., 2017.

[4] A. Huang, N. Nikaein, T. Stenbock, A. Ksentini, and C. Bonnet, "Low latency MEC framework for SDN-based LTE/LTE-A networks," in *Proc. IEEE ICC*, May 2017, pp. 1–6.

[5] N. Dukkipati *et al.*, "An argument for increasing TCP's initial congestion window," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 3, pp. 26–33, Jun. 2010.

[6] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, *Increasing TCP's Initial Window*, document RFC 6928, Apr. 2013, pp. 1–24. [Online]. Available: https://tools.ietf.org/html/rfc6928

[7] X. Nie, Y. Zhao, D. Pei, G. Chen, K. Sui, and J. Zhang, "Reducing Web latency through dynamically setting TCP initial window with reinforcement learning," in *Proc. IEEE/ACM IWQoS*, Jun. 2018, pp. 1–10.

[8] X. Nie *et al.*, "Dynamic TCP initial windows and congestion control schemes through reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1231–1247, Jun. 2019.

[9] J. Huang *et al.*, "An in-depth study of LTE: Effect of network protocol and application behavior on performance," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 363–374, 2013.

[10] C.-X. Wang *et al.*, "Cellular architecture and key technologies for 5G wireless communication networks," *IEEE Commun. Mag.*, vol. 52, no. 2, pp. 122–130, Feb. 2014.

[11] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 59–62, Jan. 2006.

[12] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with Pensieve," in *Proc. ACM SIGCOMM*, New York, NY, USA, 2017, pp. 197–210.

[13] A. Mirhoseini *et al.*, "Device placement optimization with reinforcement learning," in *Proc. ICML*, 2017, pp. 2430–2439.

[14] S. Chinchali *et al.*, "Cellular network traffic scheduling with deep reinforcement learning," in *Proc. AAAI*, 2018, pp. 1–9.

[15] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E. Long, "CAPES: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2017, pp. 42:1–42:14.

[16] R. Xie, X. Jia, L. Wang, and K. Wu, "Energy efficiency enhancement for CNN-based deep mobile sensing," *IEEE Wireless Commun.*, vol. 26, no. 3, pp. 161–167, Jun. 2019.

[17] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, New York, NY, USA, vol. 48, Jun. 2016, pp. 1928–1937.

[18] H. Wang, F. Xu, Y. Li, P. Zhang, and D. Jin, "Understanding mobile traffic patterns of large scale cellular towers in urban environment," in *Proc. Internet Meas. Conf.*, 2015, pp. 225–238.

[19] Y. Zhang, L. Cui, W. Wang, and Y. Zhang, "A survey on software defined networking with multiple controllers," *J. Netw. Comput. Appl.*, vol. 103, pp. 101–118, Feb. 2018.

[20] M. Mezzavilla *et al.*, "End-to-end simulation of 5G mmWave networks," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 2237–2263, 3rd Quart., 2018.

[21] I. Hadžić, Y. Abe, and H. C. Woithe, "Edge computing in the ePC: A reality check," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, 2017, pp. 13:1–13:10.

[22] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali, "Rethinking end-to-end congestion control in software-defined networks," in *Proc. 11th ACM Workshop Hot Topics Netw.*, 2012, pp. 61–66.

[23] K. Winstein and H. Balakrishnan, "TCP ex Machina: Computer generated congestion control," in *Proc. ACM SIGCOMM*, 2013, pp. 123–134.

[24] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance," in *Proc. USENIX NSDI*, 2015, pp. 395–408.

[25] M. Dong *et al.*, "PCC Vivace: Online-learning congestion control," in *Proc. USENIX NSDI*, 2018, pp. 343–356.

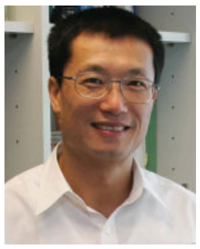
[26] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis, "QTCP: Adaptive congestion control with reinforcement learning," *IEEE Trans. Netw. Sci. Eng.*, vol. 6, no. 3, pp. 445–458, Jul. 2019.

[27] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on Internet congestion control," in *Proc. ICML*, vol. 97, Jun. 2019, pp. 3050–3059.

[28] K. Xiao, S. Mao, and J. K. Tugnait, "TCP-Drinc: Smart congestion control based on deep reinforcement learning," *IEEE Access*, vol. 7, pp. 11892–11904, 2019.

[29] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-driven congestion control: When multi-path TCP meets deep reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1325–1336, Jun. 2019.

Ruitao Xie received the B.Eng. degree from the Beijing University of Posts and Telecommunications in 2008 and the Ph.D. degree in computer science from the City University of Hong Kong in 2014. She is currently an Assistant Professor with the College of Computer Science and Software Engineering, Shenzhen University. Her research interests include AI networking and mobile computing, distributed systems, and cloud computing.



Xiaohua Jia (F'13) received the B.Sc. and M.Eng. degrees from the University of Science and Technology of China in 1984 and 1987, respectively, and the D.Sc. degree in information science from the University of Tokyo in 1991. He is currently the Chair Professor with the Department of Computer Science, City University of Hong Kong. His research interests include cloud computing and distributed systems, data security and privacy, computer networks, and mobile computing. He is also the General Chair of the ACM MobiHoc 2008, the TPC Co-Chair of the IEEE GlobeCom 2010-Ad Hoc and Sensor Networking Symposium, and the Area-Chair of the IEEE INFOCOM 2010 from 2015 to 2017. He is an Editor of the IEEE Internet of Things, the IEEE Transactions on Parallel and Distributed Systems from 2006 to 2009, *Wireless Networks*, the *Journal of World Wide Web*, and the *Journal of Combinatorial Optimization*.

Kaishun Wu received the Ph.D. degree in computer science and engineering from HKUST in 2011. After that, he worked as a Research Assistant Professor with HKUST. In 2013, he joined SZU as a Distinguished Professor. He has coauthored two books and published over 100 high quality research articles in international leading journals and primer conferences, such as IEEE TMC, IEEE TPDS, ACM MobiCom, and IEEE INFOCOM. He is also the inventor of 6 U.S. and over 90 Chinese pending patents. He is a fellow of IET. He received the 2012 Hong Kong Young Scientist Award and the 2014 Hong Kong ICT Awards: Best Innovation and 2014 IEEE ComSoc Asia–Pacific Outstanding Young Researcher Award.