ABS: Adaptive Buffer Sizing via Augmented Programmability with Machine Learning

Jiaxin Tang[1‡], Sen Liu[1‡], Yang Xu[12*], Zehua Guo[3], Junjie Zhang[4],
Peixuan Gao[5], Yang Chen[1], Xin Wang[1], H. Jonathan Chao[5]

[1]School of Computer Science, Fudan University, Shanghai, China
[2]Peng Cheng Laboratory, Shenzhen, China
[3]Beijing Institute of Technology, Beijing, China
[4]Fortinet, Inc., Sunnyvale, CA, USA
[5]New York University, USA
Email: {jxtang18, senliu, xuy, chenyang, xinw}@fudan.edu.cn, guolizihao@hotmail.com, {junjie.zhang, pg1540, chao}@nyu.edu

***Abstract*—Programmable switches have been proposed in today's network to enable flexible reconfiguration of devices and reduce time-to-deployment. Buffer sizing, an important factor for network performance, however, has not received enough attention in programmable network. The state-of-the-art buffer sizing solutions usually employ either fixed buffer size or adjust the buffer size heuristically. Without programmability, they suffer from either massive packet drops or large queueing delay in dynamic environment. In this paper, we propose Adaptive Buffer Sizing (ABS), a low-cost and deploy-friendly framework compatible with programmable network. By decoupling the data plane and control plane, ABS-capable switches only need to react to the actions from controller, optimizing network performance in run-time under dynamic traffic. Meanwhile, actions can be programmed by particular Machine Learning (ML) models in the controller to meet different network requirements. In this paper, we address two specific ML models for different scenarios, a reinforcement learning model for relatively stable network with user specific quality requirements, and a supervised learning model for highly dynamic network condition. We implement the ABS framework by integrating the prevalent network simulator NS-2 with ML module. The experiment shows that ABS outperforms state-of-the-art buffer sizing solutions by up to 38.23x under various network environments.**

***Index Terms*—adaptive buffer sizing, reinforcement learning, supervised learning, standing queue, quality of experience, NS-2**

I. Introduction

Programmable switches have been proposed in today's network to enable flexible reconfiguration of devices and reduce time-to-deployment [1]–[3]. Programmability has been introduced in packet header parsing, match-action tables [1] and packet scheduling [2], [3], which simplifies network management and promotes innovation to enhance application experience. Buffer size, an important factor for network performance, however, has been configured as a fixed value, in today's programmable switches. Buffer plays an important role in network devices (e.g., routers or switches). During congestion or bursty traffic, it is used to temporarily store packets that cannot be instantly forwarded by the device. The switch buffer size significantly affects network performance. For example, large buffers can improve link utilization by reducing the chance of packet drops, while at the same time may introduce the *buffer bloat* problem [4], where packets suffer from the significant queueing delays. On the contrary, small buffers may cause massive packet loss and retransmission, even during light congestion or bursts, eventually reducing network throughput. Setting buffers with proper size can effectively alleviate network congestion while still making full use of link bandwidth. Therefore, choosing a proper value for buffer size is essential for the network performance.

Some works are proposed to improve network performance by carefully selecting a fixed buffer size for specific network scenarios or under certain conditions [5], [6]. Over recent years, however, new network scenarios (e.g., data center networks, WiFi and cellular networks) emerge. In these network scenarios, the bottleneck link capacity is changing, and flows may have different Round Trip Times (RTTs) and different user specific requirements (i.e., Quality of Experience, QoE), all of which make fixed buffer sizing challenging. Thus, a promising solution is to adaptively adjust buffer size for different network scenarios and it has been recently attracted attentions from industry (e.g., Netflix [7], Facebook [8]) and academia [9]. However, these works lack programmability and only heuristically change the buffer size following a fixed logic. For instances, they would still run mistakenly according to the network states in the past rather than possible future network changes, when the traffic pattern alters. As consequences, the state-of-the-art solutions would hardly meet various network performance requirement in varying environments with different congestion control schemes.

In this paper, we propose an Adaptive Buffer Sizing (ABS) framework for programmable switches to dynamically configure or customize the buffer size behavior of a switch. By decoupling the data plane and control plane, ABS framework features high programmability of buffer sizing behaviors. Specifically, ABS-capable switches are able to meet various

‡ Co-first authors.
* Corresponding author.

network requirements under different dynamic environments (e.g., links with different latency and capacity, different congestion control schemes coexisting) by simply reacting to the the actions from the remote control plane. Moreover, the remote control plane can leverage different Machine Learning (ML) models to perform more complex buffer sizing actions based on the prediction of possible network changes according to the periodically collected in-network features from switches, outperforming other state-of-the-art solutions in both programmability and performances. Additionally, the ABS framework is feasible and can be easily implemented with low cost in today's programmability-enabled network with the assistance of high-performance programmable switches and the architecture of Software-Defined Networking (SDN).

The contributions of the work are summarized as follows:

- To the best of our knowledge, this is the first paper that works on the programmability of buffer sizing and introduces ML into buffer sizing issues. The proposed ABS framework defines the interface and actions between the data plane and control plane, enabling buffer size tuning based on real-time network states and their predictions.
- To verify the programmability of ABS under different network environments, two specific ML models (i.e., ABSRL and ABSSL) are addressed as typical use cases. According to our observations, analysis, and experiments, ABSRL is suitable for relatively stable network with specific user requirements (e.g., data center networks), while the ABSSL works quite well when the network is highly dynamic and noisy (e.g., wireless networks).
- We reconstruct the prevalent NS-2 simulator platform by integrating it with the ML models to evaluate the performance of ABS. The compatibility and synchronization problem between event-driven mechanism of NS-2 and message passing with the ML-based controller is resolved. Therefore, our platform can also be used for other ML-related NS-2 simulation besides buffer sizing[1].
- ABS and its programmability are thoroughly evaluated via massive experiments under various ML models, user specific requirements, network environments, and congestion control schemes. The results show that ABS can remarkably boost the network performance by up to 38.23x, compared with cutting-edge buffer sizing schemes.

The paper is organized as follows. Section II lists some related work. In Section III, we introduce the ABS framework and two ML models. Section IV is the experiment setting and evaluations of ABS. Finally Section V concludes the paper.

II. Related work

A. Buffer Sizing

Buffer sizing is a classic topic for network research, and many existing papers study how to select a fixed buffer size for good network performance [4]–[6], [10], [11]. However, as various new networks and applications come into sight, the previous fixed buffer size does not always work well [7], [12]. Westphal et al. [13] consider the wide-area network and propose to trim packets to reduce buffer sizes and improve RTTs. Facebook changes the buffer sizes of their data center and backbone routers. Their results show the buffer change could lead to tolerable degradation of packet drop rates and significant enhancements in latency [8]. CoDel [14] drops packets based on the minimum queueing delay which avoids the challenge of buffer sizing. Cocoa [9] automatically changes the buffer size by detecting the minimum queue length in the past. It has a similar goal as ABSSL but employs a heuristic algorithm with no prediction of network changes as ABS does. So the performance will degrade in dynamic and noisy network environments with multiple flows. Random Early Detection (RED) [15], an active queue management (AQM) algorithm, uses four parameters to control the drop behavior in the buffer. However, the parameters of RED are fixed and need to be manually decided and configured.

B. ML in Networks

ML has been widely used to improve network performance [16]–[34]. For instances, Reinforcement learning (RL) is used to optimize the performance of Traffic Engineering (TE) [16]–[20]. Sun et al. [21] use RL to accelerate the transmission of coflows. Jiang et al. [23] propose a similar architecture named data-driven network. Ref. [24], [25] leverage RL model for routing traffic. Ref. [26]–[28] use RL to adjust TCP congestion windows (CWnd) adaptively. Wang et al. [29] deploy an RL-based initial video segment selector on edge CDN server for optimal QoE. Heuristic algorithms can hardly solve the challenges in these related work due to complicate state and solution space. RL is effective by optimizing a definite reward function. However, RL models are hard to be trained online and converge in highly dynamic environment. Xie et al. [31] design a supervised learning (SL) algorithm for partial RL model online update. Valadarsky et al. [32] attempt to leverage SL to generate better routing configuration. Berger [33] models optimal caching decisions and uses GBDT to optimize CDN caching. Yan et al. [34] develop ABR algorithm which combines classic control with an SL predictor for video streaming.

C. Network Programmablility

Traditional network devices (e.g., switches and routers) usually have proprietary data plane implementation that cannot be modified by users. This greatly hinders the innovation of network industry. In the past decade, much effort has been put to enable programmability and softwarization in network devices, such as OpenFlow [35], [36], P4 [1], programmable scheduler [2], and Network Function Virtualization (NFV) [37]. These new techniques greatly improve the programmbility of network devices and make it possible to use ML to control and manage today's network devices for automated management and better performance.

III. Adaptive Buffer Sizing

In this section, we first introduce the framework of ABS in Section III-A, then we describe two specific ML models using

[1]The code is available at https://github.com/jiaxintang/abs/tree/master.

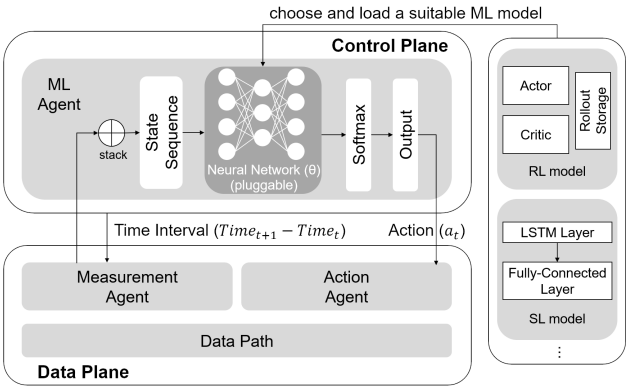


Fig. 1: The framework of ABS.

the ABS framework, ABSRL and ABSSL, in Section III-B and III-C, respectively. Finally, we discuss the feasibility of ABS in Section III-D.

A. Overview of ABS Framework

Our work focuses on programmability of the buffer size for each switch with the assistance of pluggable ML module. Fig. 1 shows the design of ABS framework, which consists of a pluggable ML module in the control plane and network configuration operations in the data plane.

In the control plane, the ML agent firstly choose and load a suitable ML model according to the network environment and user requirements. Then, the ML agent leverages the neural network model to generate action a_t, which is the current recommended buffer size adaptive to the varying network environment, and the latest a_t is sent back to the data plane periodically. Specific ML models used in ABS and their details are addressed in the Section III-B and III-C.

The data plane has two agents: measurement agent and action agent. The measurement agent is responsible for continuously collecting the network state at the switch and sending the state s_t to the ML agent in a given time interval. Specifically, the agent tracks several available variables (for specific states features, please refer to Section III-D) in a switch. The state features s_t are the average values from the last data transmission to the control plane. The action agent sets buffer size according to the latest received action a_t. The interface between the data plane and control plane includes action a_t, state s_t and time interval between the two transmissions of state. The time interval is managed by the control plane.

In conclusion, at each time step t, the ML agent receives an observation s_t from the measurement agent; it then uses the given ML model to calculate the best buffer size a_t based on the previous observations $s_t, s_{t-1}..., s_{t-w+1}$ where w denotes the observation window size. a_t is then sent to the action agent and the time interval between $Time_t$ and $Time_{t+1}$ is sent to measurement agent. Therefore, with the latest actions from the control plane, the ABS-capable switches are able to perform programmed buffer sizing logic, optimizing network performance under dynamic traffic.

B. Adaptive Buffer Sizing with RL

We design an RL model to serve as the ML agent in ABS framework, namely ABSRL, to meet different user specific requirements (i.e., QoE) in different network scenarios. Some scenarios may need extremely low delay, while the others pay more attention to throughput. Different QoE functions can serve as the rewards of the RL model. ABSRL can improve the performance of buffer sizing in a targeted manner. Note that, we do not focus on the design of network QoE functions in this paper and we use a representative one based on throughput and delay to serve as the reward and measure the performance of our model.

1) Design of RL Model: We utilize the Actor Advantage Critic (A2C) model [38] as the RL model for buffer size calculation and Proximal Policy Optimization (PPO) method [39] for model optimization.

A2C: A2C model takes advantage from both value-based and policy-based methods. Value-based methods are sample-efficient and steady, while policy-based methods can converge fast for continuous environments. In A2C, the maximum future reward $Q(s,a)$ is divided into the state value function $V(s)$ and the advantage function $A(s,a)$. $V(s)$ is the base reward of a given state no matter what action is taken. $A(s,a)$ represents the extra reward of an action compared with other actions. Then the policy is designed to maximize A, which can effectively reduce variance and increase stability. The A2C model consists of an actor and a critic. The actor learns the policy to take an action based on a given state which can maximize the estimate accumulated advantage function $\hat{A}$.

$$\hat{A}_t = \delta_t + (\gamma)\delta_{t+1} + ... + \gamma^{T-t+1}\delta_{T-1} \tag{1}$$

where

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2}$$

where $\hat{A}_t$ is the estimate advantage function at time t. The critic tries to make an estimate of the state value function $V(s)$. The two modules interact with each other to help find the best actions, i.e. the buffer size. We use a neural network to work as the actor and critic.

PPO: For training the model, we leverage PPO which deals with A2C in an off-policy manner. Compared with simple policy gradient (PG), PPO is more sample efficient and easy to converge. The objective function to be maximized combines the value function error $criticLoss$, which is the loss of the critic, and policy surrogate $actorGain$ which is the calculated advantage of the actor. Additionally, an entropy bonus is introduced to make the policy distribution more uniform and ensure sufficient exploration.

$$L(\theta) = actorGain(\theta) - criticLoss(\theta) + entropy \tag{3}$$

$$criticLoss(\theta) = (V_\theta(s_t) - (r_t + V_\theta(s_{t+1})))^2 \tag{4}$$

$$entropy = H(\pi_\theta(a_t|s_t)) \tag{5}$$

$$actorGain(\theta) = E_t[min(pr_t(\theta)\hat{A}_t), clip(pr_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{6}$$

where $pr_t(\theta)$ denotes the probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ and θ_{old} is the parameters of policy before the update. $clip$ means

clipping the value pr_t into the interval $[1-\epsilon, 1+\epsilon]$ to avoid the model stepping so far and the performance collapsing.

2) RL formulation: **State space:** The state describes the network state from the last report. State s_t of time t is $(qDelay_t,\ inPackets_t,\ throughput_t,\ pDelay_t,\ capacity_t)$, where $qDelay$ denotes the average queueing delay, $inPackets$ is the average size of enqueued packets, $pDelay$ is the average propagation delay of flows, and $capacity$ is the bottleneck link capacity. Note that we split the RTT into queueing delay and propagation delay. Queueing delay is dominated by the queue behavior in the switch while the propagation delay in turn can affect the queue behavior. Since the network environment keeps changing and our recommended buffer size should be adaptive to the network environment after it is configured, we use Long Short-Term Memory (LSTM) [40] to obtain the latent future network state $fState_t$ based on the state sequence sq_t over a past observation window with size w.

Action space: The action space is the recommended buffer sizes according to the observations. The action a_t of time t is the buffer size measured by the number of packets.

Reward function: The reward is a score of network performance. Larger reward represents better performance of the network configuration. Since network requirements of different scenarios may be different, different QoE functions can serve as the reward function. We only use a representative one here to quantitatively measure the joint impact of throughput and queueing delay. The QoE to evaluate action a_t can be calculated based on the next state s_{t+1}. In detail, the QoE is the product of $QoE_{throughput}$ and QoE_{delay}.

$$QoE(Reward) = QoE^{\alpha}_{throughput} * QoE^{\beta}_{delay} \tag{7}$$

where α and β are the *importance weights* to trade off between the throughput and the queueing delay. $QoE_{throughput}$ is calculated based on the utilization of the link capacity:

$$QoE_{throughput} = throughput/linkCapacity \tag{8}$$

QoE_{delay} follows the results of [41], where QoE_{delay} decreases along a sigmoid-like curve as the delay increased as shown in Fig. 2 which can be divided into three ranges. When the delay is very small, increasing the delay slightly degrades QoE_{delay}. However, if the delay is larger than a certain threshold, QoE_{delay} will drop rapidly even if the delay increases a little [41]. And when QoE_{delay} is very small, larger delay is hard to make the experience worse. QoE_{delay} used in our paper is given in Eq. 9.

$$QoE_{delay} = \begin{cases} a_1 * x + b_1 & \text{if } x \le t_1 \\ a_2 * x + b_2 & \text{if } t_1 < x \le t_2 \\ a_3 * e^{-b_3 * x} + c_3 & \text{if } x > t_2 \end{cases} \tag{9}$$

where x denotes the average queueing delay in seconds. The specific parameters are designed based on environment. The throughput and queueing delay will increase simultaneously when the buffer size increases; consequently, $QoE_{throughput}$ will be larger, and QoE_{delay} will be smaller. The RL model needs to maximize the QoE to make a trade-off between them.

3) Algorithm: Algorithm 1 summarizes the algorithm for ABSRL in the training phase. The algorithm for each epoch can be divided into two parts. The first part (Line 2 - Line 12) is to calculate the best buffer size and interact with the measurement and action agent. Each time a state s_t is received from the measurement agent, the agent will compute the reward $reward_{t-1}$ of the last action and push the tuple into RolloutStorage for the off-policy model update. Then it calculates the current action a_t based on the state sequence sq_t. To synchronize the time in the data plane and control plane, another output is the time interval from time t to the next observation transmission time $t+1$. a_t and the time interval between t and $t+1$ are sent to the data plane. The measurement and action agent will pause at the sync time. When the measurement agent sends the observation to the RL agent, the action agent will pause until it receives a command containing the recommended buffer size. The second part (Line 13 - Line 20) is to update the model. Since we utilize PPO for policy optimization, we update for several times based on the sampled data from RolloutStorage. Additionally, for the inference phase, the parameter θ is well trained. Only Line 8 - Line 9 need to be used for buffer size inference.

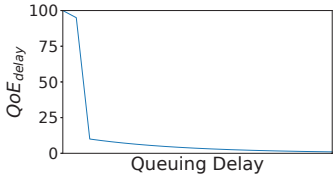


Fig. 2: An example of QoE_{delay} curve.

Algorithm 1 The training algorithm for ABSRL

Input: Observation s_t of switch
Output: Recommended Buffer size a_t, time interval between t and $t+1$

1: **while** $Epoch \le$ EPOCHNUM **do**
2: **while** $currenttime \le$ SIMULATIONTIME **do**
3: Fetch observation state s_t
4: Push s_t extracted from the observation into sq_t
5: **if** $sq_t.length() \ge$ WINDOWSIZE **then**
6: $reward_{t-1} \leftarrow QoE(qDelay, throughput)$
7: Push $(s_{t-1}, a_{t-1}, v_{t-1}, reward_{t-1})$ into the RolloutStorage
8: $fState_t \leftarrow LSTM(sq_t)$
9: $a_t, v_t \leftarrow Actor_\theta(fState_t), Critic_\theta(fState_t)$
10: **end if**
11: Pass message with a_t and time interval
12: **end while**
13: **for** $(s_t, a_t, v_t, reward_t)$ in RolloutStorage **do**
14: $r_t \leftarrow reward_t + \gamma * v_{t+1}$
15: **end for**
16: **while** $\exists$ Data in RolloutStorage not be used for optimizing **do**
17: fetch a batch from RolloutStorage
18: $\Delta L_\theta \leftarrow actorGain_\theta - criticLoss_\theta + entropy$
19: Update parameter θ of the model:

$$\theta \leftarrow \theta + \alpha \Delta L_\theta$$

20: **end while**
21: **end while**

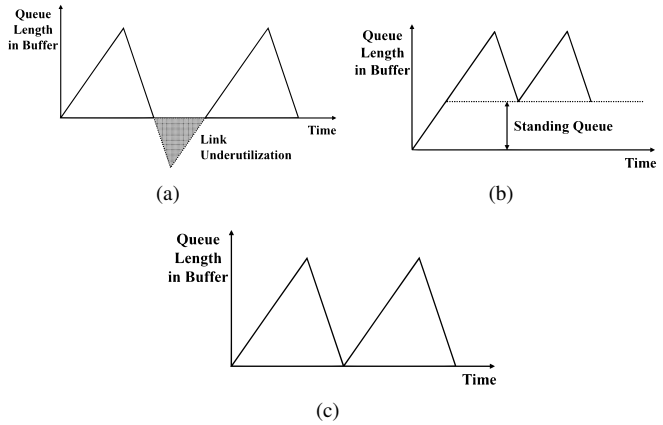


Fig. 3: The queue length in buffer with different buffer size ((a) small buffer; (b) large buffer; (c) optimal buffer).

C. Adaptive Buffer Sizing with SL

Since the drastic change of link capacity of the network may introduce noise into the rewards, RL can hardly converge. We also introduce SL into buffer sizing in the noisy environment and propose ABSSL.

1) Optimization Goal: Inspired by CoDel [14], we clarify a general optimization goal for a good buffer management, that is to minimize the standing queue and spare ratio of the bottleneck link simultaneously. The standing queue is the queue always existing in the buffer which is caused by the mismatch between the sending window of senders and the buffer size. Taking TCP Reno as example, the CWnd of the sender continuously grows until a packet is dropped, which represents the fullness of buffer. Then, CWnd cuts down and the sending rate slows down abruptly, which results in the decrease of the number of packets in the buffer. As the sending rate grows up, the queue in the buffer becomes longer again. The cyclic process is shown in Fig. 3. If the buffer size is too small (as Fig. 3(a)), the buffer will be empty and the bottleneck link will be underutilized when the sending rate decreases. However, if the buffer is too large (as Fig. 3(b)), there will always exist a queue in the buffer, resulting in excess delay. If standing queue and spare ratio of the bottleneck link are both 0 as shown in Fig. 3(c), the system will achieve the minimum queueing delay while keeping the maximum link utilization.

2) Design of SL Model: Based on the above optimization goal, we design an SL model, i.e., ABSSL, which utilizes LSTM to predict the standing queue in the next time interval which is the size of buffer to be changed. The features are the same as the state space of ABSRL. The predicted queue length can be a positive value which means that our current buffer needs to shrink. On the other hand, the predicted result can also be a negative value which means that the current buffer size is too small to fully utilize the bandwidth in the next interval, so the buffer size need to be increased. Here we limit the largest value of change because drastic and frequent fluctuation of buffer size may cause large number of packet drop and then greatly damage the performance. We measure the length of standing queue rather than the queueing delay used in CoDel, because the ground truth is easier to acquire in the next interval based on the measured state features. For example, if the output of the model at time t is $\hat{y}_t$, which means that the predicted standing queue at time $t+1$ is $\hat{y}_t$. Then, in order to keep both the standing queue and spare ratio as 0 at time $t+1$, the buffer size decreases by $|\hat{y}_t|$ packets (if $\hat{y}_t$ is positive), or increases by $|\hat{y}_t|$ (if $\hat{y}_t$ is negative). When the measurement agent obtains the actual standing queue Y_{t+1} and spare ratio sr_{t+1} next time, we can calculate the ground truth y_t as Eq. 10. Then we leverage Adam [42] to train the SL model. Since we can evaluate the effectiveness of the model in real time and use the collected data for model update, the training is in place.

$$y_t = \begin{cases} \hat{y}_t + Y_{t+1} & \text{if } Y_{t+1} > 0 \\ \hat{y}_t - linkCapacity_{t+1} * sr_{t+1} & \text{if } Y_{t+1} = 0, sr > 0 \\ \hat{y}_t, & \text{if } Y_{t+1} = 0, sr = 0 \end{cases} \tag{10}$$

D. Feasibility

Both the measurement agent and action agent of ABS can be easily implemented on today's programmable switches (e.g., barefoot's tofino) using languages such as P4. The communication between the ML agent and measurement/action agent can be realized by protocols such as OpenFlow or P4 runtime. The data transmitted from the measurement agent to the ML agent is the network state features. As we mentioned in section III-B, the state features used in ML models include ($qDelay$, $inPackets$, $throughput$, $pDelay$, $capacity$), the total size of which is 40 bytes (i.e., 8 bytes each). Among them, $qDelay$, $inPackets$, $throughput$ and $capacity$ of the egress link are directly available in the view of the switch. As for $pDelay$, it can be calculated as $RTT - qDelay$. Here, we leverage the TCP timestamp mechanism used in Linux, which is opened by default, to obtain the RTT of end-hosts at the switch. The timestamp when the data packet is sent is stored in the option field of TCP header. RTT can be calculated by the difference between two timestamps from the sequential packets with an ACK packet between them. In order to mitigate the computation overhead of RTT at the switch, we periodically sample the packets in flows and extract the timestamps for RTT calculation. Additionally, the average of RTT from all flows is used to obtain the value of $pDelay$.

For the ML agent, the training phase is efficiency-tolerable which can be deployed on a high-performance server. As for ABSRL, the training module need to be implemented under the simulation environment. The pattern of the practical environment can be stable for relatively long time, and the model will be retrained based on the pattern at a regular basis. To avoid the sudden failure of our trained model caused by accidental change of environment, we periodically disable our RL model and set the buffer size to be the value calculated by Bandwidth-Delay-Product (BDP). Then the performance of ABS and BDP-based buffer sizing will be compared. If the performance of ABS is poor, the model will be retrained. For ABSSL, the training process is much simpler. Since its ground truth can be obtained in real time, we do not need a simulation environment and we can use the data collected recently from

the real environment for training. Moreover, we can calculate the deviation between ground truth and the predicted value, and then check if the SL model fails. In case of failure, the model will be retrained.

As for the inference module of ML agent, one design is to deploy on each single node, that is each high-performance switch. We test it on a low-performance server whose CPU is Intel Xeon with a single core. The time for each inference of ABS is less than 2ms which is negligible compared with the time interval between two modification operations to the buffer size. The CPU utilization is less than 0.5% and the memory size of the trained model is smaller than 124 MB. Another type of deployment is to resort to the architecture of SDN. In this case, the ML inference module is deployed in the controller and the value of buffer size is transmitted to the switch periodically. The size of one state transmitted from the switch to the controller is 40 bytes. The data transmitted from the controller to the switch include the buffer size and the time interval, which is 12 bytes in total. If the time interval between two rounds of communication is set as 0.1s, the bandwidth consumption of communication between the controller and a switch is only 4.16 Kbps.

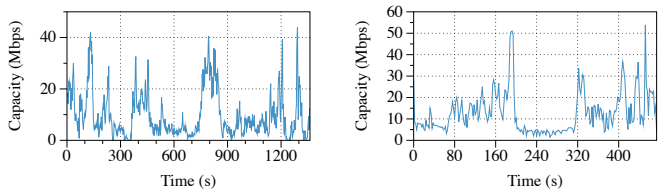
IV. Evaluation

A. Experiment Setting

We design a simulation platform for evaluating the performance of ABS. The platform consists of the NS-2 simulator, the ML module, and the interfaces for their interactions (as Fig. 1). NS-2 is well known as an event driven simulator. Therefore, it maintains its own timeline during its simulation. There is huge time drifting compared to the timeline in the ML module, resulting in severe synchronization issues. To this end, we modify the queue management module (e.g., Drop-tail) in NS-2, and add timers to trigger the feature transmission operation when the ML module exactly requires, ending with time synchronization between NS-2 and the ML module by force. Then, we provide interfaces in queue management module of NS-2 for both fetching features to the ML module and deploying actions obtained from the ML module. As a result, our aforementioned modifications make NS-2 compatible with ML module in SDN controllers.

Our testbed topology is a dumbbell topology consisting of several pairs of senders and receivers, and two programmable switches on the link deployed with our modified Drop-tail module. To test the impact of buffer size on different RTTs and bottleneck link capacity, we design two experimental environments:

Env 1: The bottleneck link capacity is relatively stable while the RTTs of the connections are diverse. The detailed topology consists of 100 senders and 100 receivers. In order to produce the congestion link, the bandwidth of the links between end-hosts (i.e., both senders and receivers) and switches is 10Gbps, whereas the one between two switches is only 1Gbps. To provide RTT variance, the latency between senders and switches is classified into ten groups according to its value, with the latency varies from 0.1 ms to 7.8 ms. Moreover, the latency between receivers and switches is completely symmetry to the one between senders and switches.



(a) (b)

Fig. 4: The capacity of wireless network. The left one is denoted as capacity trace 1 and the right one is trace 2.

In order to test the performance of ABS thoroughly, we conduct the experiments with three different distributions using representative parameters to mimic the realistic network environments. First, the arrival of flows (i.e., the beginning time of flows) obeys the Poisson distribution with the average arrival ratio $\lambda = 0.02$. Next, the transmission duration of flows follows exponential distribution with average transmission duration $k = 5$. Third, the senders and receivers are randomly picked up with normal distribution with the expectation of location μ. Here, μ increases by 5 every one second and it decreases by 5 every one second when it grows to 100. After dropping to 0, μ repeats increasing with the same step of 5 every one second. Therefore, the RTTs of flows could vary from each other once their arrival time is different. The whole simulation lasts for 300 seconds. We split the simulation into two parts. The data provided in the first 200 seconds are used for training and those in the last 100 seconds are regarded as the test set.

Env 2: This environment is setup to mimic the wireless communication. Different from Env 1, RTTs between all the senders and receivers are identical (i.e., 100 ms). Since wireless has smaller bandwidth than the previous environment, we change the bandwidth of the links between end-hosts and switches to around 160Mbps. Due to the reduced capacity of the bottleneck link, the experiments are conducted with only 2 senders and 2 receivers. The variation of the bottleneck link's capacity follows the trace captured from the real wireless communication [43], in which the capacity is quickly changing as shown in Fig. 4. All the flows arrive simultaneously and they last till the end of the simulation in this environment. Other network parameters are the same as Env 1.

For the parameters of the QoE function, we empirically summarize the latency of the data center and wireless network and design two QoE_{delay} for Env 1 and Env 2, respectively.

B. Simulation Evaluation

1) Baselines: **Fixed Buffer Sizing**: The buffer size is set as two fixed values in different orders of magnitude, namely 10 packets and 100 packets[2].

BDP-based scheme: The buffer size is adaptively set as the recommendation proposed by [5] without any ML framework

[2]In this paper, the buffer size in packets means how many MTU-sized packets the buffer can accommodate.

involved, which is the product of bandwidth and RTT divided by the square root of the number of flows. Since queueing delay is related to buffer size, only propagation delay is regarded as the RTT. We use the RTT and bandwidth collected in the current time interval to calculate and set the buffer size in the next time interval.

Cocoa: Cocoa is a heuristic method to change the buffer size based on the standing queue and spare ratio in the past with no prediction.

Besides the buffer sizing schemes above, we also compare the performance of our proposed ABS framework with two classic congestion control methods.

RED: RED is an AQM method for congestion avoidance. Compared with Drop-tail, it drops packets before the buffer becomes completely full.

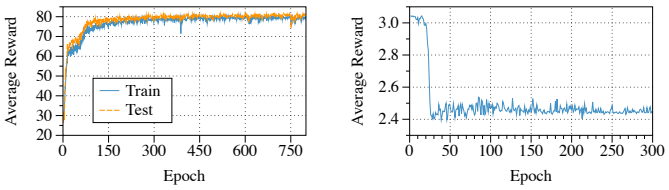
CoDel: CoDel controls the network congestion by dropping packets based on the minimum queueing delay.

2) Flows with different RTTs (Env 1): As for ABSRL and ABSSL, the size of the buffer is updated every 1 second according to the result of the ML module. Here we use 1 second because the environment is relatively stable. We also tried other experiments with time interval set as 0.1 second, and obtained similar results. The adjusting frequency in BDP-based scheme is the same as ABSRL and ABSSL. The parameters of the QoE function are set as follows:

$$QoE_{delay}1 = \begin{cases} -5.0 \times 10^4 * x + 100.0 & \text{if } x \leq 10^{-4} \\ -8.5 \times 10^5 * x + 180.0 & \text{if } 10^{-4} < x \leq 2 \times 10^{-4} \\ 22.0 * e^{-3988.6*x} + 0.1 & \text{if } x > 2 \times 10^{-4} \end{cases} \tag{11}$$

We set the importance weights of throughput and delay as $\alpha = 1$ and $\beta = 1$. Then, we get the QoE function $QoE1$.

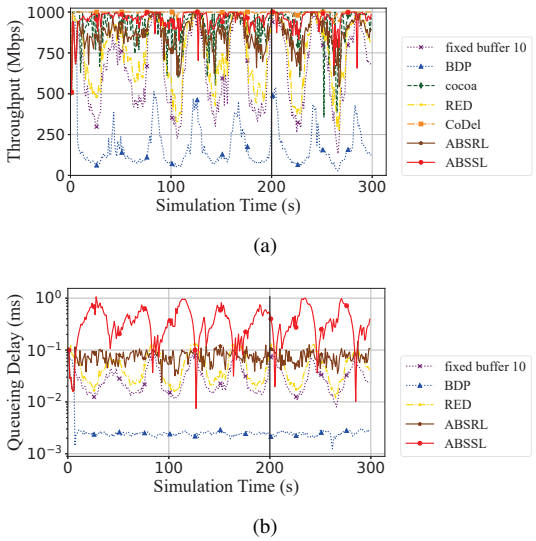
Performance Comparison We conduct the experiments when TCP New Reno, TCP Cubic and mixed congestion control algorithms (60% New Reno, 30% Cubic, 10% BBR) are used as the transmission protocol between all senders and receivers. The performance of ABS and the baselines are shown in Table I where THP means throughput. Generally, the average QoE of ABS significantly outperforms the baselines by up to 38.23x. An interesting observation is that the average buffer size recommended by ABSRL is 22.25 when using New Reno, while the value is 11.31 when using Cubic. However, the average throughput of Cubic is larger than that of New Reno, which shows that the requirement of buffer size when using Cubic is smaller than that when using New Reno under the same traffic pattern. ABS framework can achieve better performance whatever congestion control algorithm is used. The average QoE of ABSRL is 24.1% better than fixed buffer sizing schemes. ABSSL does not perform best in this scenario because its optimization goal pays more attention to the utilization of the link bandwidth, while the QoE function has more stringent requirements on queueing delay. However, it still outperforms both CoDel and buffers with inappropriate fixed size. CoDel performs well on throughput, but it suffers greater delay because of the setting of fixed queueing delay target. As for cocoa, due to its design does not consider multiple flows, the queueing delay is extremely unstable.



(a) (b)

Fig. 5: The average reward of ABSRL (a) and the loss of ABSSL (b) during the training epochs in Env 1.

Then we take TCP New Reno as example and dive into the results. The average reward of ABSRL and the loss of ABSSL during the training epochs are shown in Fig. 5. We can find that ABSRL gradually converges after 200 epochs, while ABSSL converges much more quickly. As shown in Fig. 6, we compare the throughput and delay in every second with other baselines. The left part of the black vertical line (i.e., simulation time $\leq$ 200) is used for training, and the right part is the test set. According to our experiment result, the performance on training set and test set is similar. Compared with the fixed buffer sizing schemes, ABSRL can flexibly adjust the buffer with the change of RTT to achieve relatively stable throughput. As for the BDP-based scheme, since the queueing delay in the next time interval cannot be foreseen, it underestimates the RTT and the buffer size is relatively small, resulting in underutilization of bandwidth. Therefore, the performance is relatively poor. Additionally, we show the change of buffer size during the simulation in Fig. 7(a). Both ABSRL and ABSSL are able to automatically adjust the buffer size according to the network condition. The buffer size grows as RTT increases, while buffer size reduces when RTT decreases. In all the cases, ABSRL can make a best trade-off between throughput and delay which meets the requirements of the QoE, and ABSSL has acceptable performance in general scenarios.



(a)

(b)

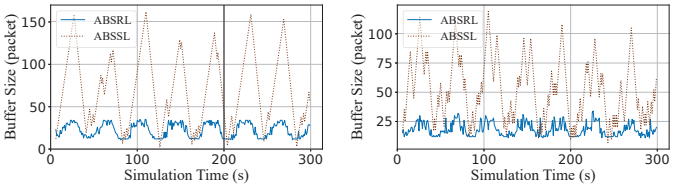
Fig. 6: Comparison of throughput and delay in Env 1. (We do not show the queueing delay of cocoa and CoDel in this figure due to its extremely large value.)

TABLE I: Performance of ABS and baselines in Env 1.

Method	TCP New Reno				TCP Cubic				Mixed CC			
	Avg THP (Mbps)	Avg Delay (ms)	95th %ile Delay (ms)	Avg $QoE1$	Avg THP (Mbps)	Avg Delay (ms)	95th %ile Delay (ms)	Avg $QoE1$	Avg THP (Mbps)	Avg Delay (ms)	95th %ile Delay (ms)	Avg $QoE1$
Fix(10)	685.31	0.033	0.073	65.62	893.46	0.052	0.010	84.68	825.83	0.0434	0.1013	78.40
Fix(100)	968.38	0.712	0.985	2.13	994.64	0.948	1.11	1.05	974.27	0.7903	1.0858	4.566
BDP	177.16	0.0025	0.0028	17.28	358.28	0.0023	0.0034	34.94	973.74	9.202	26.525	2.792
cocoa	932.51	22.03	104.3	11.85	944.31	2.30	4.17	35.77	901.40	24.62	123.70	24.74
RED	774.69	0.057	0.121	71.26	943.35	0.085	0.115	78.35	878.50	0.0686	0.1436	75.04
CoDel	994.96	6.95	10.58	0.09	997.31	7.46	112	0.090	992.89	6.852	9.910	0.0896
ABSSL	956.21	0.388	0.938	17.23	976.75	14.3	0.248	55.20	705.24	0.0400	0.1505	61.91
ABSRL	**877.43**	**0.075**	**0.107**	**81.44**	**903.89**	**0.050**	**0.033**	**85.18**	**856.54**	**0.0638**	**0.1133**	**80.11**

TABLE II: Performance of ABS and baselines in Env 2.

Method	TCP New Reno				TCP Cubic				Mixed CC			
	Avg THP (Mbps)	Avg Delay (ms)	95th %ile Delay (ms)	Avg $QoE2$	Avg THP (Mbps)	Avg Delay (ms)	95th %ile Delay (ms)	Avg $QoE2$	Avg THP (Mbps)	Avg Delay (ms)	95th %ile Delay (ms)	Avg $QoE2$
Fix(10)	1.84	32.6	82.8	51.05	2.28	33.5	90.2	55.11	1.94	32.36	84.02	41.10
Fix(100)	5.73	185.2	512.2	49.78	6.73	230.8	632.0	46.12	6.62	210.47	604.42	39.60
BDP	2.17	30.13	69.49	49.88	3.20	34.2	85.7	51.01	2.31	34.76	94.13	40.52
cocoa	8.06	2264.4	7503.4	15.40	8.06	6495	18151	12.15	4.63	2578.01	8122.34	4.233
RED	2.30	37.39	101.6	54.88	2.81	41.9	113.6	59.11	2.44	39.51	108.12	44.47
CoDel	4.40	38.75	110.3	64.62	4.87	40.6	120.6	66.66	4.63	38.24	116.07	47.03
ABSSL	**6.10**	**61.17**	**173.4**	**65.31**	**6.42**	**44.7**	**97.3**	**70.29**	**6.27**	**58.46**	**154.01**	**47.14**



(a) (b)

Fig. 7: The change of buffer size. (a) shows the buffer size in Env 1. (b) shows the buffer size with a new traffic pattern.

TABLE III: The effect of different QoE functions in Env 1.

Method	$\alpha = 2$ $\beta = 1$	$\alpha = 1$ $\beta = 2$	QoE ($QoE_{delay}2$)
Fix(10)	49.26	64.36	66.49
Fix(100)	1.88	0.18	21.30
BDP	4.41	17.26	17.29
Cocoa	11.25	10.28	23.21
RED	57.56	67.61	74.83
CoDel	8.73e-02	8.31e-05	0.19
ABSSL	15.94	10.90	64.11
ABSRL	69.29	77.16	85.09

Effect of different QoE functions We conduct experiments to test the effect of different QoE functions. We change the parameters in $QoE1$ into $\alpha = 2, \beta = 1$ and $\alpha = 1, \beta = 2$. We also test new $QoE_{delay}2$, in which the parameters are set as follows: $a_1 = -1.67, b_1 = 100.0, a_2 = -2.83 \times 10^5, b_2 = 180.0, a_3 = 18.0, b_3 = -999.1, c_3 = 0.1, t_1 = 3 \times 10^{-4}, t_2 = 6 \times 10^{-4}$. $t1$ and $t2$ in $QoE_{delay}2$ are larger than those in $QoE_{delay}1$, which means that $QoE_{delay}2$ has more relaxed requirements on delay. The results are shown in Table III. Since the QoE functions serve as the reward function in the training process of ABSRL, it can cater to different network requirements of applications and achieve better performance than other baselines. However, ABSSL cannot adapt to specific requirements. The performance of ABSSL is better when the QoE function pays more attention to throughput or has less strict requirements on queueing delay. So, when α, $t1$ and $t2$ is much larger, the QoE performance of ABSSL can be better.

Generalization Since the network is dynamic, the pattern of traffic may change. To test the generalization of the trained model, we change the pattern of the traffic with average flow arrival ratio $\lambda = 0.04$ and average flow transmission duration $k = 10$. Then we use the model trained in the previous scenario (i.e., the model trained with $\lambda = 0.02$ and $k = 5$) to decide the buffer size. The buffer size can still been automatically adjusted as shown in Fig. 7(b). The performance of ABSRL and ABSSL are still acceptable (Avg $QoE1$ of ABSRL=76.00, Avg $QoE1$ of ABSSL=18.35) in case of the different traffic patterns.

3) Links with dynamic capacity (Env 2): Similar to the experiment above, we conduct experiments to study whether ABS can adapt to the network with dynamic capacity. In this environment, we set the parameters of QoE_{delay} as follows:

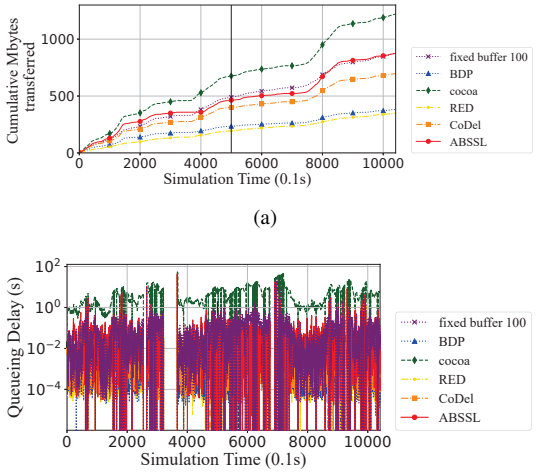
$a_1 = -62.5, b_1 = 100.0, a_2 = -1214.3, b_2 = 192.1, a_3 = 27.4, b_3 = -6.8, c_3 = 0.07, t_1 = 0.08, t_2 = 0.15$. α and β are set as 1. The QoE function is denoted as $QoE2$. We take capacity trace 1 in Fig. 4 as example. The first 500 seconds is used as the training set and the remaining trace is regarded as the test set. Since the trace is changing almost all the time, the configuration time interval is set as 0.1 second.

Performance Comparison We examine the performance of ABS in Env2 when TCP New Reno, TCP Cubic and mixed congestion control algorithms (one flow uses New Reno and the other flow uses Cubic) are used by the senders. By conducting several times of experiments, we find that ABSRL can hardly converge when the capacity is highly dynamic and much noisy is introduced into the reward, while ABSSL is much more robust in such complicate environment. The results shown in Table II demonstrate the effectiveness and efficiency of ABSSL. ABSSL outperforms all the baselines and its performance using Cubic is much better than it using New Reno because of the improvement of throughput, while the performance of CoDel is similar. So ABSSL performs better than CoDel when using Cubic.

For better illustration, we take TCP New Reno as example and plot the cumulative bytes transferred and queueing delay of different schemes in Fig. 8. The cumulative bytes transferred can be seen as the performance of average throughput. The left part of the black vertical line in the figure is the training phase, and the right part is the test phase. ABSSL can achieve better balance between throughput and delay by flexibly changing the buffer size (as shown in Fig. 9(a)), outperforming fixed buffer sizing schemes. On one hand, when the fixed buffer size is small, the throughput is small (namely, the link capacity cannot not fully utilized). On the other hand, the large fixed buffer size results in excess queueing delay. BDP-based scheme suffers from the low throughput due to the smaller buffer size, because this scheme is not able to acquire the queueing delay in the future and it calculates its buffer size by only considering the current propagation delay. Cocoa has a good performance on throughput, but the delay of it is intolerable and may cause timeout. Compared with buffer sizing schemes, CoDel performs much better. However, the strict limitation on the queueing delay results in a relatively poor performance on throughput.

Generalization Since users may move to areas covered by different wireless networks, we examine the effectiveness of the model trained in the environment with capacity trace 1 in Fig. 4 when it is used in a new environment with capacity trace 2. ABSSL can still adjust the buffer size based on the new capacity trace (shown in Fig. 9(b)) and the performance of ABSSL keeps stable (Avg $QoE2 = 70.96$) which demonstrates the generalization and robustness of the trained model of ABSSL. Moreover, ABSSL can be trained in place and the failure of ABSSL model can be quickly found and updated to adapt to a completely new environment.

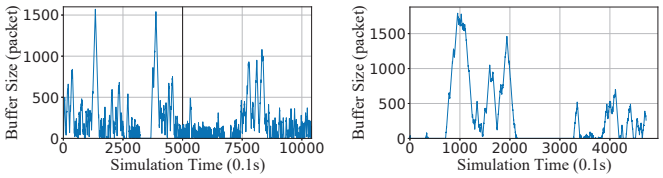
In conclusion, we test our proposed ABS framework in two environments. The overall performance of ABS shown in Table I and II demonstrates that ABS outperforms other buffer sizing schemes when RTT or link capacity is dynamic. We also compare it with other classic congestion control algorithms which do not base on buffer size. CoDel performs well in the environment with stable RTTs, however, the performance is relatively poor when RTTs of flows are highly dynamic. RED has too much parameters which are hard to configure in different environments. So, ABS framework is much more efficient and easily-configured in dynamic environments.



(a)

(b)

Fig. 8: Comparison of throughput and delay in Env 2.



(a) (b)

Fig. 9: The change of buffer size. (a) shows the buffer size in Env 2. (b) shows the buffer size of capacity trace 2 (model trained with capacity trace 1).

V. Conclusion

In this paper, we propose a novel programmable buffer sizing framework, ABS, to meet the user requirement in dynamic network traffic. Two specific ML models, namely ABSRL and ABSSL, are addressed as typical use cases of programmability for different dynamic network environments. The experiments on the reconstructed dedicated simulation platform show that our ABS remarkably improve the performance by up to 38.23x on QoE functions compared with the state-of-the-art buffer sizing solutions.

Acknowledgment

This work is sponsored by Key-Area Research and Development Program of Guangdong Province (2021B0101400001), National Natural Science Foundation of China (62150610497, 62172108, 61971145, 62002066, 62002019), China Postdoctoral Science Foundation (2021M690705), the Beijing Institute of Technology Research Fund Program for Young Scholars, and Shanghai Pujiang Program (2020PJD005).

REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[2] A. Sivaraman, S. Subramanian, M. Alizadeh *et al.*, "Programmable packet scheduling at line rate," in *SIGCOMM*, 2016, pp. 44–57.

[3] N. Sharma, C. Zhao, M. Liu *et al.*, "Programmable Calendar Queues for High-speed Packet Scheduling," in *NSDI*, 2020, pp. 685–699.

[4] J. Gettys, "Bufferbloat: Dark Buffers in the Internet," *IEEE Internet Computing*, vol. 15, no. 3, pp. 96–96, 2011.

[5] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," *ACM SIGCOMM CCR*, vol. 34, no. 4, p. 281–292, 2004.

[6] N. Beheshti, E. Burmeister, Y. Ganjali *et al.*, "Optical packet buffers for backbone internet routers," *ToN*, vol. 18, no. 5, pp. 1599–1609, 2010.

[7] B. Spang, B. Walsh, T. Huang *et al.*, "Buffer sizing and video qoe measurements at netflix," *Stanford University, Workshop on Buffer Sizing*, 2019.

[8] N. Beheshti, P. Lapukhov, and Y. Ganjali, "Buffer sizing experiments at facebook," *Stanford University, Workshop on Buffer Sizing*, 2019.

[9] M. Bachl, J. Fabini, and T. Zseby, "Cocoa: Congestion Control Aware Queuing," *Stanford University, Workshop on Buffer Sizing*, 2019.

[10] M. Enachescu, Y. Ganjali, A. Goel, N. McKeown, and T. Roughgarden, "Routers with very small buffers." in *INFOCOM*, 2006.

[11] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental study of router buffer sizing," in *IMC*, 2008, pp. 197–210.

[12] N. McKeown, G. Appenzeller, and I. Keslassy, "Sizing router buffers (redux)," *ACM SIGCOMM CCR*, vol. 49, no. 5, pp. 69–74, 2019.

[13] C. Westphal, K. Makhijani, and R. Li, "Packet Trimming to Reduce Buffer Sizes and Improve Round-Trip Times," *Stanford University, Workshop on Buffer Sizing*, 2019.

[14] K. Nichols and V. Jacobson, "Controlling Queue Delay," *Commun. ACM*, vol. 55, no. 7, p. 42–50, Jul. 2012.

[15] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[16] P. Sun, Z. Guo, J. Li, Y. Xu, J. Lan, and Y. Hu, "Enabling scalable routing in software-defined networks with deep reinforcement learning on critical nodes," *IEEE/ACM Transactions on Networking*, 2021.

[17] M. Ye, J. Zhang, Z. Guo, and H. J. Chao, "Federated traffic engineering with supervised learning in multi-region networks," in *IEEE ICNP*, 2021, pp. 1–12.

[18] ——, "Date: Disturbance-aware traffic engineering with reinforcement learning in software-defined networks," in *IEEE/ACM IWQOS*, 2021, pp. 1–10.

[19] J. Zhang, M. Ye, Z. Guo, C.-Y. Yen, and H. J. Chao, "Cfr-rl: Traffic engineering with reinforcement learning in sdn," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2249–2259, 2020.

[20] Z. Xu, J. Tang, J. Meng *et al.*, "Experience-driven networking: A deep reinforcement learning based approach," in *INFOCOM*, 2018, pp. 1871–1879.

[21] P. Sun, Z. Guo, J. Wang, J. Li, J. Lan, and Y. Hu, "Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling," in *IJCAI*, 2021, pp. 3314–3320.

[22] A. Mestres, A. Rodriguez-Natal, J. Carner *et al.*, "Knowledge-defined networking," *ACM SIGCOMM CCR*, vol. 47, no. 3, pp. 2–10, 2017.

[23] J. Jiang, V. Sekar, I. Stoica, and H. Zhang, "Unleashing the potential of data-driven networking," in *COMSNETS*, 2017, pp. 110–126.

[24] L. Chen, J. Lingys, K. Chen *et al.*, "Auto: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *SIGCOMM*, 2018, pp. 191–205.

[25] P. Sun, Y. Hu, J. Lan, L. Tian, and M. Chen, "TIDE: Time-relevant deep reinforcement learning for routing optimization," *Future Generation Computer Systems*, vol. 99, pp. 401–409, 2019.

[26] W. Li, H. Zhang, S. Gao, C. Xue, X. Wang, and S. Lu, "Smartcc: A reinforcement learning approach for multipath tcp congestion control in heterogeneous networks," *JSAC*, vol. 37, no. 11, pp. 2621–2633, 2019.

[27] Z. Xu, J. Tang, C. Yin *et al.*, "Experience-driven congestion control: When multi-path TCP meets deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1325–1336, 2019.

[28] X. Nie, Y. Zhao, Z. Li, G. Chen, K. Sui, J. Zhang, Z. Ye, and D. Pei, "Dynamic TCP Initial Windows and Congestion Control Schemes Through Reinforcement Learning," *JSAC*, vol. 37, no. 6, pp. 1231–1247, 2019.

[29] H. Wang, K. Wu, J. Wang, and G. Tang, "Rldish: Edge-Assisted QoE Optimization of HTTP Live Streaming with Reinforcement Learning," in *INFOCOM*, 2020, pp. 706–715.

[30] Y. Li, Z. Han, Q. Zhang, Z. Li, and H. Tan, "Automating Cloud Deployment for Deep Learning Inference of Real-time Online Services," in *INFOCOM*, 2020, pp. 1668–1677.

[31] R. Xie, X. Jia, and K. Wu, "Adaptive Online Decision Method for Initial Congestion Window in 5G Mobile Edge Computing Using Deep Reinforcement Learning," *JSAC*, vol. 38, no. 2, pp. 389–403, 2020.

[32] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to Route," in *HotNets*, 2017, p. 185–191.

[33] D. Berger, "Towards Lightweight and Robust Machine Learning for CDN Caching," in *HotNets*, 2018, p. 134–140.

[34] F. Yan, H. Ayers, C. Zhu *et al.*, "Learning in situ: a randomized experiment in video streaming," in *NSDI*, 2020, pp. 495–511.

[35] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[36] D. Erickson, "The beacon openflow controller," in *HotSDN*. ACM, 2013, pp. 13–18.

[37] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[38] V. Mnih, A. Badia, M. Mirza *et al.*, "Asynchronous Methods for Deep Reinforcement Learning," in *ICML*, 2016, pp. 1928–1937.

[39] J. Schulman, F. Wolski, P. Dhariwal *et al.*, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[40] F. Gers, "Learning to forget: continual prediction with LSTM," *IET Conference Proceedings*, pp. 850–855(5), January 1999.

[41] X. Zhang, S. Sen, D. Kurniawan *et al.*, "E2e: embracing user heterogeneity to improve quality of experience on the web," in *SIGCOMM*, 2019, pp. 289–302.

[42] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.

[43] "Mahimahi," http://mahimahi.mit.edu, 2015, [Online; accessed 25-June-2021].