Reinforcement Learning Based Congestion Control in a Real Environment

Lei Zhang, Kewei Zhu, Junchen Pan, Hang Shi, Yong Jiang, Yong Cui*
Department of Computer Science and Technology, Tsinghua University, Beijing, China
leizhang16@mails.tsinghua.edu.cn, zkw18@mails.tsinghua.edu.cn, panjc17@mails.tsinghua.edu.cn,
shi-h15@mails.tsinghua.edu.cn, jiangy@mail.sz.tsinghua.edu.cn, cuiyong@tsinghua.edu.cn

***Abstract*—Congestion control plays an important role in the Internet to handle real-world network traffic. It has been dominated by hand-crafted heuristics for decades. Recently, reinforcement learning shows great potentials to automatically learn optimal or near-optimal control policies to enhance the performance of congestion control. However, existing solutions train agents in either simulators or emulators, which cannot fully reflect the real-world environment and degrade the performance of network communication. In order to eliminate the performance degradation caused by training in the simulated environment, we first highlight the necessity and challenges to train a learning-based agent in real-world networks. Then we propose a framework, ARC, for learning congestion control policies in a real environment based on asynchronous execution and demonstrate its effectiveness in accelerating the training. We evaluate our scheme on the real testbed and compare it with state-of-the-art congestion control schemes. Experimental results demonstrate that our schemes can achieve higher throughput and lower latency in comparison with existing schemes.**

***Index Terms*—Reinforcement learning, network system, congestion control**

I. Introduction

Congestion control (CC) remains a cornerstone issue in the networking field, attracting much attention from both academia and industry [1] [2] [3]. The goal of congestion control is to dynamically regulate the sending data at each sender to maximize the total throughput, minimize the queueing delay, and minimize the packet loss [4].

So far, the research on congestion control can be roughly divided into three phases [5]. In the first phase, general-purpose congestion control schemes such as Reno [6] and Cubic [7] were proposed. These schemes treat all data flows and users fairly and became the default deployment method. Subsequently, researchers tried to develop special-purpose schemes [8] [9] to improve congestion control and studied how these new schemes coexist with the default ones. In the latest phase, researchers made no assumptions about what schemes are used by others, and designed schemes to help flows survive well with other traffic [10] [11] [12]. In the first two phases, the developed schemes deal with issues such as the complexity of network topology, the difference in the number of flows, and the traffic demand/dynamics, which are already very complicated. In the third phase, it becomes considerably more complicated due to the ignorance of the behavior of other co-existing traffics. The traditional congestion control schemes mainly focus on the problems faced by the first two phases, not being able to solve the problems in the latest phase.

Recently, machine learning technologies are developing rapidly and can solve complex problems, bring new opportunities to enhance congestion control. Deep reinforcement learning (RL) [13], one of the latest breakthrough techniques in machine learning filed, has been demonstrated as a powerful approach to sequential decision-making problems [14]. Integrating deep RL into the network system is emerging as a new interdisciplinary research topic, which has attracted a significant amount of research attention. It has been demonstrated that reinforcement learning can improve the performance of networking, including but not limited to congestion control [10] [15], video streaming [16] [17], network topology [18] and routing [19].

To the best of our knowledge, almost all of the existing RL-based approaches for congestion control are designed based on simulated environments. For example, Remy [10] generates congestion control rules for TCP based on simulations in NS-2 [20]. Aurora [15] and Custard [21] use deep RL to generate policies that map the observed network statistics to the sending rate based on their simulators. Indigo [12] learns the best congestion windows (cwnds) offline based on the emulator, i.e., Mahimahi [22]. Although these schemes use reinforcement learning methods to cope with varying network conditions, the trained models could not be directly applied in the real network system. Because either these emulators or simulators are based on numerical calculations and cannot actually send packets, or the package-level simulators cannot truly reflect the real-world network. According to the study presented in [12], the performance difference between the simulation environment and the real system can be more than 17%. Moreover, simply deploying the agent trained from the simulation environment in the real-world systems will encounter several practical issues such as the inference cost, the real-time decision-making issue, the generalization problem.

In this paper, we move one step further to address the issues faced by simulation-based RL congestion control schemes, aiming to design a learning-based congestion control framework that enables network operators to train an RL agent for congestion control in real environments instead of simulators. First, we highlight the key challenges of training an RL agent for congestion control in the real world. To solve the issues of

*Yong Cui is the corresponding author.

978-1-7281-6607-0/20/$31.00 ©2020 IEEE

congestion control in the latest phase, we show that congestion control is viewed as a learning task. And the RL agent can be trained to seek optimal or near-optimal control policies based on out-of-sync information provided by the sender and the receiver under diverse network conditions. To address the RL training challenges in the real network environment, we present a framework called *ARC* for training an RL agent in a real environment rather than a simulator. Specifically, we leverage an asynchronous learning algorithm to train a deep reinforcement learning model for congestion control based on the environment where the data packets are actually sent instead of numerical calculations in simulators.

We implement ARC in user space for congestion control. The sender and receiver are implemented based on the UDP transmission skeleton. We deploy our trained agent in the sender, which infers the best sending rate by leveraging both historical trajectories and the current network state. We conduct experiments to compare the performance of our schemes with state-of-the-art congestion control schemes such as Cubic [7], BBR [11], and Reno [6]. Extensive evaluations show that our scheme achieves high average throughput similar to that of BBR, and reduces the latency by 43% than that of BBR. Due to its insensitive to stochastic packet loss, ARC achieves higher throughput than that of Cubic and Reno. In addition, we test the RL agent under various parameters, such as the neural network architecture and the decision interval, and present the evaluation results, which shed light on the practical concerns about using the RL-generated model for congestion control in real networks.

II. Motivation and Challenges

In this section, we first give our motivation for using a trained reinforcement learning agent for congestion control in real-world networks. Then we present the key challenges to design such an RL-based congestion control scheme in the real environment.

A. Motivation

Traditional congestion control schemes usually use metrics such as packet loss and round-trip time (RTT) as decision signals to adjust the sending rate or cwnds. Existing rule-based methods elaborately make use of these signals but achieve poor throughput when stochastic packet loss or network jitter occurs frequently. In fact, the performance of a congestion control scheme can be affected by many factors, including traffic patterns, link failure, dynamic latency, packet loss, application requirements and so on. It is difficult to design optimal or even near-optimal control policies from complex network environment with predefined static rules. As reinforcement learning is able to generate proper actions in highly dynamic and complicated environments such as the real-world networks, we can train a reinforcement learning agent to generate models for congestion control by leveraging both the historical trajectories and the current network state to dynamically adjust the sending rate or the cwnds at the sender.

Recently, many RL-based congestion control schemes have been proposed. The RL training process allows agents to learn packet-sending rules from enormous observations and feedback in the specific environment, thereby showing great promise in handling problems in complicated networks. However, existing approaches usually train agents in simulated environments to speed up the training process or easily control the training environment. Network simulators such as NS-2 [20] and real-time emulators such as Mahimahi [22] and Emulab [23] have been used to train RL agents, since these simulators are simple to use and can be easily controlled to simplify RL training.

However, simulation-based agent training has many disadvantages. For example, packet-level simulators such as NS-2 can simulate the process of sending data but suffer low acceleration ratio, resulting in longer training time in comparison with training based on the real network system. Different from the packet-level simulators, emulators such as Mahimahi can provide numerical calculations about the network behaviors. Although such emulators can provide a certain acceleration ratio, how to properly set the parameters to emulate a target network still remains as an open problem [12]. Moreover, the simulated environment could be blocked by the senders when they are waiting for the actions from the RL agent. The blockage can take a few milliseconds, which will negatively impact the transmission performance in the real networks [24].

To the best of our knowledge, there are no RL-based solutions for congestion control that have been implemented in real-world production systems. The preliminary results presented in [12] [25] and [26] show that directly applying the RL agents trained on the simulation or emulation platforms to the real-world systems without modification will lead to the under-utilization of bandwidth and long delay because such platforms cannot fully represent the real-world systems.

B. Key Challenges

To use a reinforcement learning agent for congestion control in the real-world systems, the following challenges must be addressed.

1) Training RL agent challenges: The first challenge is that there are many differences in the training mechanisms between a simulated environment and the real-world networks. First, when an RL agent is trained in a simulated environment, the learning model can be updated after each step of training [10] [15], since the simulation environment can be easily blocked to enable such updates. However, in the real world, the environment (i.e., the network sender) cannot be blocked. Hence, the learning model cannot be updated after each step of training, which will seriously degrade the training efficiency. Second, the exploration operation in the traditional RL training scheme, such as exploring the sending rate or cwnds, is generally added to the training process of the agent. However, in the real world, the model used by the sender is fixed during each episode of training because of the non-blocking environment. Therefore, the exploration mechanism cannot be directly added to the agent.

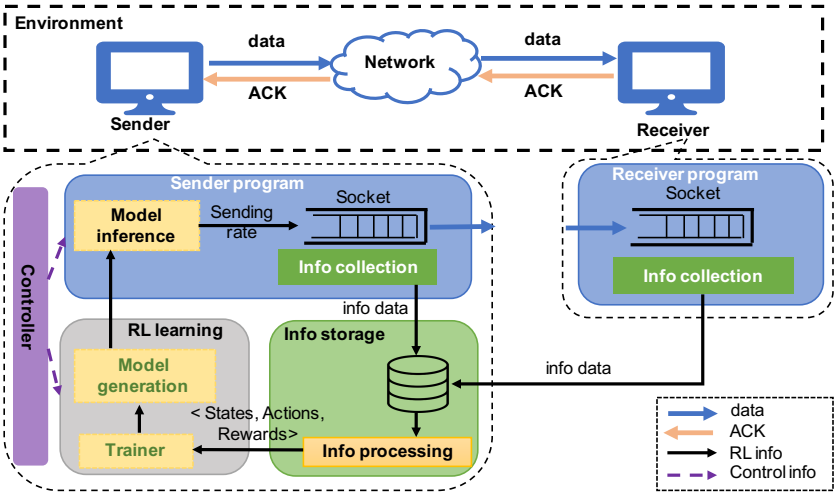

Fig. 1. The framework of ARC.

2) Real environment challenges: The second challenge is that training an RL agent in the real world is much more difficult than that in a simulated environment, such as blocking environment, out-of-sync information, inference cost. Firstly, the current simulated environments for RL-based CC adopt the synchronous RL approach, e.g., using the OpenAI Gym [27] to block the simulation environment for model training, which is controlled by the RL agent. The real network system could not be blocked by the sender and the RL agent is driven by the system. Hence, the RL agent for congestion control deployed in the real world will affect the real network throughput. Recently, several schemes using remote procedure calls (RPC) to communicate between the real system and the agent have been proposed [25] [26]. Even though these schemes provide a standard request-response design pattern, they still suffer from the drawbacks of delaying action and blocking the environment.

Secondly, the information needed for RL training, such as the sending rate, the number of sent packets, the receiving rate or loss rate, comes not only from the sender but also the receiver. In a simulation or emulation platform, such information can be easily obtained. The $\{state, action, reward\}$ tuple can be easily synchronized because the RL agent is allowed to block both the sender and receiver. In the real world, both the receiver and the sender cannot be blocked. Hence, it is challenging for the RL agent to obtain synchronized state, action and reward in real network environments.

Thirdly, since the RL agent needs to run in a real-time environment, the sender should get the control policy from the trained model. The inference cost will affect the gain in many metrics such as throughput and latency. The inference time should be faster than the data transmitted over the real network so that it will not impact the transmission performance. Therefore the overhead of the RL model should be carefully handled. In the congestion control problem, if the inference time is too long, the sender will wait a long time to get a decision, thereby degrading the network performance.

III. The Framework of ARC

In this section, we present the framework of *ARC*, a reinforcement learning based congestion control scheme designed based on real network environments. Figure. 1 depicts the framework of ARC based on a general environment with one connection between one sender and receiver through the network links. ARC has five key modules: the *Info Collection* module in both the sender program and receiver program gathers the information required for training. The collected raw information is stored in the *Info Storage* module and will be processed to output the $\{state, action, reward\}$ tuples that will be used by the *RL learning* module to train the agent. The *Model inference* module enables the trained agent to make decisions on adjusting the sending rate. A *Controller* is used to control the decision interval, probabilistic exploration, and the parameters for training. In the following, we first describe the workflow of ARC and then describe the detail of the functions provided by each module.

A. Workflow of ARC

ARC follows an environment-driven design for congestion control. Before training an RL agent in the system, the sender loads the RL-based model which can be updated periodically. To get the action, the sender takes the decision form the current trained RL agent in the sender program. The agent makes decision-making on the sender every decision interval, i.e., the agent follows a policy of the sending rate for congestion control. Specifically, each time a connection is established, the agent will synchronize its policies with the trainer to obtain the current learned policies. In addition, the training of the RL agent and the execution of sending data are asynchronous. The information required by training an RL agent is collected to the information storage module from the sender and receiver. Meanwhile, the RL learning module then generates the agent through step by step training from the historical trajectories.

B. Information Collection

When a connection is established between a sender and one receiver, the sender transmits the data to the receiver according

to the control policy from the trained agent. The training data of the agent comes from the raw information providing by the sender and receiver. When receiving an ACK, the sender obtains the current RTT and the received sequence number of packets. At an interval t, the sender computes statistics based on ACKs such as the sent bytes, ack bytes, average RTT, average packet sent interval. When the receiver receives the packets, it also counts received bytes with the same interval. The information is collected from the sender and the receiver respectively and stored in the *Info storage*.

C. Information Storage & Processing

To deal with the out-of-sync information problem, ARC stores the raw information collected by both the sender and the receiver at the storage module located at the sender side respectively. The raw data is then processed based on the RL formulation (See in §IV-A). In ARC, congestion control is formulated as a sequential decision-making problem under the RL framework. The states of RL are the network statistics from the sender, the action is the sending rate for the sender, and the reward depends on the statistics of throughput, delay, and loss rate at an interval t. The collected data is processed to generate the {*state, action, reward*} tuples to be used for training the agent. In particular, to match the delayed action with the corresponding state and the reward, the calibrate timer in ARC at the sender starts up when the first packet is sent and a similar timer at the receiver starts up when the first packet is received.

D. RL Leaning

To tackle the non-blocking environment problem, ARC adopts an asynchronous RL training with off-policy correction. The trainer takes sampled historical data from the information storage and uses them to learn the optimal or near-optimal control policies. Compared with the conventional RL algorithm, the exploration mechanism and trigger event in ARC are different. The exploration mechanism is added to the system environment (See in §IV-B). Moreover, ARC follows environment-driven design with an asynchronous RL agent, not an agent-driven design. We detail the asynchronous RL algorithm for congestion control in the following section (See in §IV-C).

E. Model Inference

To obtain the output of the model, the sender should load the model, feed inputs, and retrieve inference output. The inference module is located at the sender program. In the initial connection, the sender loads the default model. During the transmission processing, the sender continually get a decision from the model inference on adjusting the sending rate. Meanwhile, the RL learning module continually updates the model based on the training data at the sender.

F. Controller

To deal with the inference cost issue, a controller is used in ARC to control the decision interval and the parameters for RL training. A decision interval is defined as the amount of time to perform one control loop to solve the real-time decision of the congestion control problem, i.e., the decision interval when the agent is invoked. The controller also controls the probability exploration rate and maximum exploration scope. In addition, ARC decouples the training algorithm and execution algorithm. The controller controls the agent learning and data transmission respectively. The data sent will not be blocked during agent learning. The sender in the real-world network executes the agent for the next action. Once a full trajectory such as 64 consecutive interval information is obtained, this information is trained to update the agent. All interactions are asynchronous and will not block the sender.

IV. RL Algorithm for Congestion Control

In this section, we present the detailed design of the RL agent for congestion control. We first describe the formulation of state, action, and the reward function for congestion control. Then we present the RL exploration strategy and the asynchronous training algorithm.

A. RL Formulation

The congestion control problem can be modeled as reinforcement learning task where the learning agent provides dynamic *policies* to map the feedback from the network and the receiver (i.e., *reward*) to the selection of the sending rate (i.e., *action*) based on current observations (i.e., *state*).

State: The state in the RL agent is a snapshot of the environment at the decision interval that can be observed by the agent after the sender selects the rate. Let s_t be the state at time step t. s_t is defined as a vector that includes: (i) the averaged sent packet interval, (ii) the packet loss, (iii) the average delay, (iv) the sent bytes, and (v) the last action. The statistics can be easily obtained from the sender by tracing the ACKs signals and the decision intervals.

Action: The action indicates how the agent responds to the observed state of the environment. In our formulation, the action a_t is defined as the sending rate of the sender, which is a continuous variable. The agent takes an action at the end of each *decision interval*.

Policy: The policy is a set of rules that map from the observed states of the environment to actions to be taken when in those states. The policy learning is the core of a reinforcement learning agent. In congestion control, the action is selected by a policy $\mu(s_t)$. The policy is represented by neural networks with a manageable number of adjustable parameters of neural networks. In ARC, we leverage the deterministic policy gradient algorithm [28] to generate the policy which can output the deterministic action from the continuous action space.

Reward: A reward is the overall benefit of an RL agent when it follows a policy. In each decision interval, the sender updates the observed state and executes an action to adjust the sending rate, which results in an instantaneous reward r_t. Generally, the instantaneous reward of congestion control only depends on the current state and action, and it is independent of the

previous states and actions. Hence, we adopt a linear function that rewards throughput while penalizing loss and delay to achieve high throughput, low latency, and low congestion loss rate. The reward r_t is defined as:

$$r_t = \frac{throughput_t}{throughput_{max}} - \frac{avg\_delay_t}{delay_{max}} - (loss\_rate_t - \alpha) \quad (1)$$

where $throughput_t$ is the instantaneous observed throughput of the receiver and the $throughput_{max}$ is the maximum value among all the history throughput; $delay_{max}$ is the maximum delay of the current connection; avg_delay_t is the average delay; $loss\_rate_t$ is the observed packet loss rate from the sender and receiver. Parameter α represents an acceptable network packet loss rate. In ARC, α is set to 0.05. Generally, the RL agent selects an action to maximize the expected cumulative reward $\epsilon[\sigma_{t=0}^{T}\gamma^t r_t]$, where $\gamma \in (0, 1)$ is a discount factor and T is the total learning steps.

B. Exploration Strategy

Exploration is essential and important in the training process of reinforcement learning. Through exploration, the agent could obtain sufficient transition samples to gain experience and attempt to achieve the optimal and near-optimal policy. In traditional RL training algorithms, the RL exploration adds some random noise into the decision-making mechanism of an action. However, since the RL trainer is independent of the environment, the exploration strategy cannot be directly added to the RL trainer. Therefore, in our approach we deploy the exploration mechanism to the environment, that is, the sender first uses random sampling with normal distribution to obtain a random noise, then adds the noise to the action obtained from the agent. Finally, the sender executes the updated action to explore possible strategies.

C. Asynchronous Training

Reinforcement learning usually takes a long time to train in simulators and training in the real world will be more difficult. To speed up the training, ARC uses an asynchronous training mechanism in which network communication and agent training is executed asynchronously. ARC can start up multiple environments. Each environment is configured to experience a different set of network conditions. However, these agents continually send their {*states, actions, rewards*} tuples to the learning agent. For each sequence of tuples that it receives, the learning agent uses the DDPG [28] algorithm. DDPG uses the actor-critic algorithms with off-policy to compute a gradient and perform a gradient descent step. The actor-networks are responsible for choosing the proper action. The critic networks estimate the value of an action and conduct to update the parameters of actor and critic networks. The learning agent then updates the actor-network and the environments load the new model when they are initialized. Algorithm 1 shows the pseudocode of the asynchronous learning to speed up training. Note that this can happen asynchronously among the learning and multiple environments, i.e., there is no blocking between the learning agent and environments.

Algorithm 1 The asynchronous training of ARC

```
Initialize NumEnv //The maximum number of environments
Parallel do
    for i in 1...NumEnv:
        env.run() // Environment running parallelly
    end for
    agent.learn() // The agent learning parallelly
End parallel do
function ENV.RUN()
    while Training is not end do
        current_model = get_model()
        run_receiver()
        run_sender(current_model)
        if Sending is end then
            Store information from sender and receiver
        end if
    end while
end function
function AGENT.LEARN()
    while Training is not end do
        load experience_replay
        current_model = DDPG.learning()
    end while
end function
```

V. Implementation

We implement a user-space prototype of ARC for congestion control. The sender and receiver are implemented in the user space by adapting the UDP-based transmission skeleton. ARC replaces the TCP sender-side rate control with the model inference module, data collection and storage module, and rate control module. ARC also provides a controller to set the decision interval, the training parameters, and so on. The RL agent of ARC is implemented with PyTorch [29].

The prototype of ARC provides the interfaces to implement the RL-based agent congestion control in the real network environment. The detailed interfaces are as follows. The *parameter_config* function uses to set the parameters for decision interval and training. Both the *send* and the *receive* functions are running in non-blocking mode. The *send* function sends data according to the action returned by the *choose_action* function. The output of *choose_action* function is the sending rate of sender based on the input of *state*. The *receive* function is responsible for receiving data. When each training episode ends, the *store_information* function stores the information from the sender and the receiver to the data storage server. In ARC, we adopt Redis [30], which is an open-source (BSD licensed) and in-memory data structure store, as the storage server.

Model Inference: To implement an RL agent for the real-world networks, the sender usually loads the trained model, feeds the preprocessed state, and retrieves the action. Thus, the learning-based design should consider the inference cost

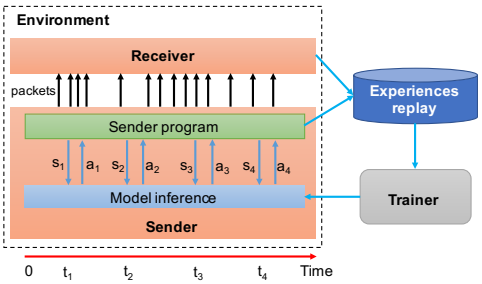

Fig. 2. The asynchronous execution of ARC.

and the heterogeneity of the implementation platforms, which can impact the gain on throughput and delay for congestion control. To deal with this problem, we transform the RL agent into the ONNX (open neural network exchange) [31] format, which provides a standard for representing machine learning models and enables models to be deployed in a variety of frameworks and runtimes. The sender program uses ONNX Runtime [32] for inference in ARC. The inference time of ARC takes approximately 0.8 ms (See in §VI-A).

Asynchronous Execution: Existing RL-based congestion control environments such as Aurora [15] and Indigo [12] follow the synchronous RL approach e.g., using the Gym [27] to block the sender for model execution. In the synchronous mode, packets could not be sent until getting the action. This makes it infeasible for deployment because a sender waiting for the decision-making from the RL agent for a few milliseconds, which will negatively impact throughput [24]. To address this problem, the processes of model inference and sending packets are asynchronous in our implementation so that the model inference process can not affect the running of congestion control. Figure. 2 illustrates the asynchronous execution process. First, the trained model is converted into the ONNX model and loaded in the sender program. The sender invokes the model inference for the action according to the current state in every decision interval. Meanwhile, the sender is not blocked and still sends the packets according to the last action until it gets the new action.

VI. Evaluation

In this section, we evaluate the performance of ARC through experiments on a real testbed and compare it with several congestion control schemes deployed in the Linux kernel. We also evaluate the impact of various parameters such as the decision interval and neural network architecture on the performance of ARC.

A. Setup

To evaluate ARC, we use tc [33] to regulate the bandwidth, minimum RTT, and stochastic packet loss in the bottleneck links. To generate the control policies for congestion control, ARC's agent uses four fully connected layers in which there are 2 hidden layers with 128 neurons. The actor-network takes the processed features from the state and outputs the sending rate with the "tanh" activation function. The critic-network takes the inputs and uses the same architecture as the actor-network to conduct feature extraction. The final output of the critic network is a linear neuron without activation function. During the training stage, we empirically set the discount factor $\gamma = 0.9$, which implies that the current action will be influenced by 10 future steps. The learning rates of the actor and critic networks are configured to be 10^{-5} and 10^{-4}, respectively.

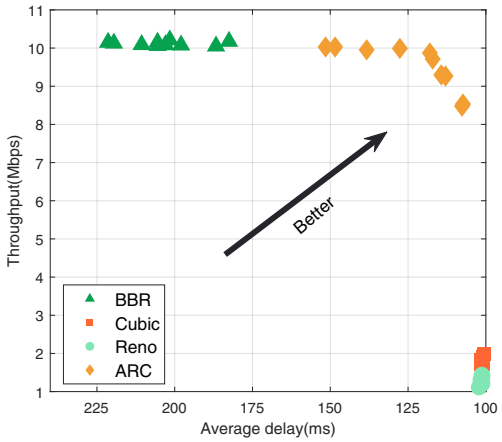

Fig. 3. The performance of congestion control in ARC.

B. The Performance of ARC

To validate the effectiveness of ARC, we repeatedly run ARC under different network conditions. In this section, we first evaluate the performance of throughput and delay of ARC. Then we describe the training performance of ARC under different network conditions.

1) The performance of throughput and delay: To evaluate the performance of congestion control of ARC, we set up a dynamic link with an average bandwidth of 10Mbps, 100 ms minimum RTT, and 1% stochastic packet loss. The in-network buffer is set to 100 KB for the bottleneck link. We test ARC against Cubic [7], Reno [34], and BBR [11] over the same link. ALL schemes are repeatedly tested 10 times and each test lasts for 30 seconds.

The results are shown in Figure. 3. As expected, these schemes have different trade-off points between throughput and delay. Due to the stochastic packet loss on the bottleneck link, Cubic and Reno fail to achieve high throughput. Both ARC and BBR achieve nearly 10 Mbps on throughput since they are not sensitive to stochastic packet loss. However, BBR has significantly higher delay than ARC due to its inaccuracy estimation about minimum RTT and bandwidth. On average, the delay of ARC is 43% lower than that of BBR. This is because the RL agent of ARC takes the maximum throughput, minimum RTT and minimum packet loss as its target during the training process, ARC tries its best to achieve higher throughput and lower delay without packet congestion loss under the current network conditions.

2) The performance of training: To evaluate the training performance in the real networks, we compare the training

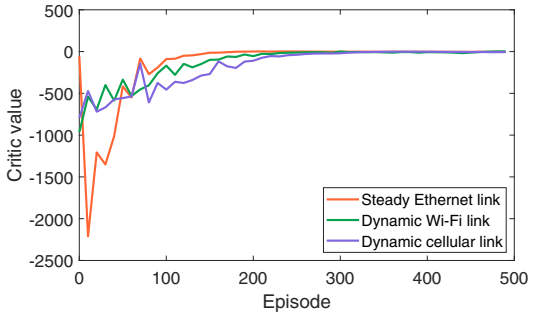

Fig. 4. The critic-network values under different network conditions.

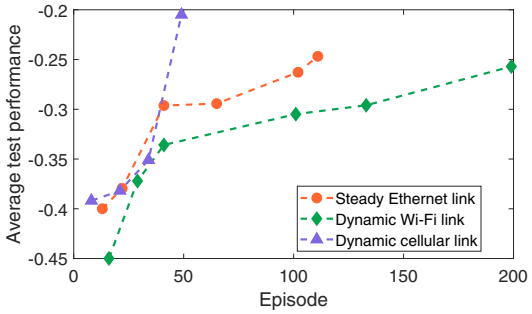

Fig. 5. The testing performance of different models in the training process.

results in the following three scenarios with different network links which refer to [12].

- *Steady Ethernet link*: 10 Mbps bandwidth, 100 ms minimum RTT, 0% loss rate, and 125KB buffer.
- *Dynamic Wi-Fi link*: 10 Mbps average bandwidth, 100 ms minimum RTT, 1% packet loss, and 100KB buffer.
- *Dynamic cellular link*: 3 Mbps average bandwidth, 200 ms minimum RTT, 5% packet loss, 75KB buffer.

We use the same parameters to train RL agents separately in the above scenarios. Figure. 4 shows the critic-network value under these three network scenarios which represent the expected cumulative reward. When the value of the critic network no longer grows, we say the RL algorithm has converged and the model has been well trained. It can be seen that the critic value reaches a stable value at about 200 episodes in the above three networks.

We also evaluated the test performance of the models in the training process which is computed by the reward function. As shown in Figure. 5, the average test performance of the above models in the intermediate process is lower. Hence, the throughput is also low and the delay is high. For the well-trained model, the average test performance is the largest, thus the model can achieve higher throughput and lower delay. In contrast, the average test performance of the cellular network increases fast, whereas in the Ethernet link and Wi-Fi link grow slowly. This is because the average bandwidth of the cellular network scenario is smaller. The action space is smaller accordingly than that of the other two links, so the optimal or near-optimal policy can be found faster.

C. The Microbenchmarks of ARC

In this section, we describe the microbenchmarks that can provide a deeper understanding of the behavior of ARC and shed light on some practical concerns with the RL-based congestion control schemes. Firstly, we compare the effect with different decision intervals on throughput. We then analyze the impact on inference time with different neural network architectures on inference time.

1) Decision interval: The decision interval is one of the important parameters in ARC, and improper setting on decision interval can affect the performance of ARC. On the one hand, if the decision interval is too small, the number of model inference will increase, thereby resulting in long the inference time. On the other hand, if the decision interval is too large,

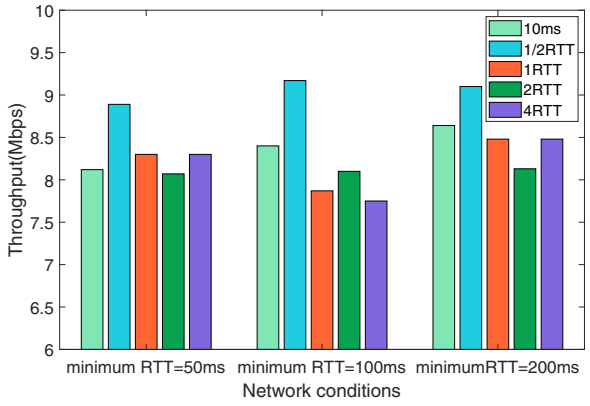

Fig. 6. The performance of different decision interval.

it is impossible to track the changes in network conditions in time, thereby failing to react to the changes in network states.

To test the effect of different decision intervals on network performance, we evaluated the throughput under different decision intervals with different RTTs. Considering the recent studies of congestion control, the interval used varies from 10 ms [25] to 4 RTT [35]. We conduct experiments with decision intervals ranging from 10 ms to 4 minimum RTT under the bottleneck links with different RTTs. Figure. 6 shows the throughput under different decision intervals. It can be seen that the performance under the decision interval of 1/2 RTT is better than that of other decision intervals, even on the network conditions with different RTTs. Benefiting from 1/2 RTT, the state reflects the changes in network conditions in time. At the same time it may ensure sufficient time so that the action could be fully executed in the case of no congestion, that is, the data sent by the sender according to the decision-making can reach the receiver, thereby the estimation of reward is more accuracy.

2) Neural network architecture: ARC uses the learning architecture (See in §VI-A) to convert to ONNX format. To understand the impact of different neural network structures on the model inference, we swept a range of neural network parameters with different numbers of neurons and numbers of hidden layers to test the inference time. First, using a single fixed hidden layer, we varied the number of neurons in the hidden layers. Each set is tested 10 times and Figure. 7 shows the average inference time with different numbers of neurons in the hidden layer. As expected, the inference time gradually

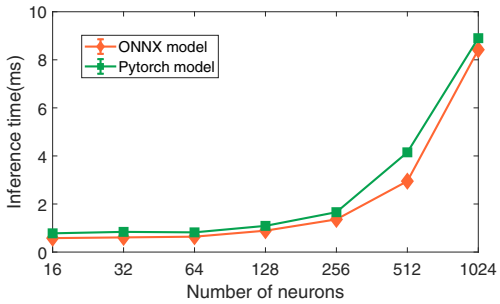

Fig. 7. The inference time with different number of neurons.

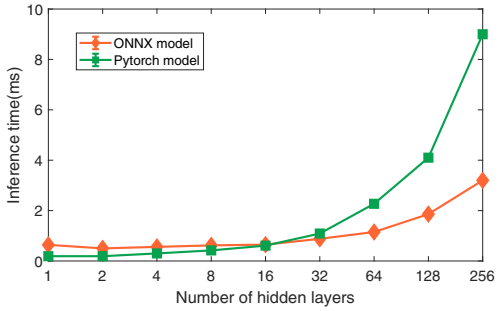

Fig. 8. The inference time with different hidden layers.

increases as the number of neurons increases. In addition, the inference time of the ONNX model is less than that of the PyTorch [29] model.

Next, we varied the number of hidden layers where the number of neurons in each layer is fixed to 128. Figure. 8 shows the inference time under different numbers of hidden layers. Interestingly, we find that the shallower neural networks of 2 hidden layers yield the best performance. The inference time steadily rises as we increase the number of hidden layers. Adjusting these parameters to construct deeper neural networks may improve performance. Meanwhile, these complex neural networks generally take a long time to train. Compared with the PyTorch model, when the hidden layer is a shallow network, the inference time of the ONNX model is slightly higher than that of the PyTorch model. When the number of hidden layers gradually increases, the inference time of the ONNX model is much lower than that of the PyTorch model. Therefore, if the neural network structures for RL-based CC are small, Both the PyTorch model and the ONNX model can meet the transmission requirements. If the neural network architecture is larger, using the ONNX model could improve the performance of throughput than that of the PyTorch model.

VII. Related work

Many efforts have been devoted to applying reinforcement learning to network transmission systems, including the studies that use reinforcement learning to congestion control. Moreover, some researches have focused on the learning-augmented computer system. We summarize the related work as follows.

RL for Internet congestion control Simulators and emulators, such as NS-2 [20] and Mahimahi [22], have been used by researchers to design novel congestion control schemes because they provide various parameters to simulate different network conditions including traffic pattern, queue management, and stochastic loss rate. However, properly setting these parameters to emulate a target network is still difficult. Custard [21] and Aurora [15] leverage deep RL to generate a policy that maps observed network statistics to a proper sending rate or cwnds for Internet congestion control. They used offline learning strategy because the simulated environment can be blocked by the RL agent. When evaluating in the real world, the offline-trained agents cannot block the environment. Moreover, the inference time usually impacts the performance in real networks.

Pantheon [12] provides a set of benchmark algorithms of congestion control and a shared evaluation platform. Indigo [12] is another method of learning-based CC scheme with the data gathered from Pantheon. Indigo learns to "imitate" the oracle rules offline, in which the oracle is constructed with ideal cwnds given by the emulated bottleneck's bandwidth-delay product. R3Net [26] is an RL-based recurrent network for Real-time communications (RTC), allowing rapid adjustment to complex and dynamic network conditions. It also uses a simulation environment. When evaluating R3Net in the real world, the R3Net agent is deployed into the ONNX format and uses ONNX Runtime for inference in the RTC system. Though the training in the simulation could speed up training, it might result in poor performance due to the huge gap is between the simulation environment and the real world.

Learning-augmented computer system. In [25], Park discussed some challenges in designing RL systems and developed a platform for researchers to experiment with RL for the computer systems. The platform consists of 12 real-world system-centric optimization problems, such as congestion control, job scheduling, SQL database query optimization, and so on. However, Park's congestion control environment adopts the user-space implementation together with RPC for environment agent communication. It takes a synchronous approach with a short step-time of 10 ms, thereby only suitable to very small models. AutoSys [36] is another work about the co-design of learning algorithms to the system. It provides two principles to make systems learnable and two principles to make learning manageable for the case of web search. However, it lacks the performance validation and does not address the challenges for learning algorithms to be implemented in real systems.

There are other specific applications such as video streaming, live streaming, which perform offline training in simulation and running in the real-world. For example, ABRL [37] adopts a training method that depends on simulator to cope with the changing network conditions and deploys the trained agent in the real world. L3VTP [38] is a live video transmission platform used to speed up the live ABR research loop. L3VTP trains agents on simulation platforms and only focuses on the low-latency video transmission. Different from these solutions that use the simulated environment to train agents, our solution ARC trains the agent in real-world networks and achieves better performance.

VIII. Conclusion

Combining reinforcement learning and congestion control is a promising approach to improve the performance of the real-world network systems. However, there are some challenges to be addressed before successfully deploying an RL agent in the real world. To eliminate the huge gap between training RL agents in simulation platforms and running in the real-world networks, We propose a framework, ARC, to solve congestion control problem with reinforcement learning in the real world with asynchronous policy. We have designed an RL-based congestion control algorithm and trained it in the real network environment. We implement ARC in the user space. Extensive evaluation results show that ARC can achieve high throughput and low delay than the default scheme of the Linux kernel in the real environments.

Acknowledgment

This work is supported in part by the National Key R&D Program of China under Grant 2018YFB1800303 and in part by NSFC Project under Grand 61872211.

References

[1] X. Zuo, M. Wang, T. Xiao, and X. Wang, "Low-latency networking: Architecture, techniques, and opportunities," *IEEE Internet Computing*, vol. 22, no. 5, pp. 56–63, 2018.

[2] H. Shi, Y. Cui, X. Wang, Y. Hu, M. Dai, F. Wang, and K. Zheng, "Stms: Improving mptcp throughput under heterogeneous networks," in *USENIX Annual Technical Conference*, 2018.

[3] W. K. Leong, Z. Wang, and B. Leong, "Tcp congestion control beyond bandwidth-delay product for mobile cellular networks," *International Conference on Emerging NETWORKING Experiments and Technologies*, pp. 167–179, 2017.

[4] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," *USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 329–342, 2018.

[5] L. Zhang, M. Wang, Z. Yang, Y. Jiang, and Y. Cui, "Machine learning for internet congestion control: Techniques and challenges," *IEEE Internet Comput.*, vol. 23, no. 5, pp. 59–64, 2019.

[6] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 314–329. [Online]. Available: http://doi.acm.org/10.1145/52324.52356

[7] L. XU, "Cubic : A new tcp-friendly high-speed tcp variant," *Proc. Workshop on Protocols for Fast Long Distance Networks,*, 2005.

[8] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound tcp approach for high-speed and long distance networks," *IEEE INFOCOM 2006*, 2006.

[9] Y. Zaki, T. Potsch, J. Chen, L. Subramanian, and C. Gorg, "Adaptive congestion control for unpredictable cellular networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 509–522, 2015.

[10] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *Sigcomm*, 2013.

[11] S. H. Yeganeh and S. H. Yeganeh, "Bbr: congestion-based congestion control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[12] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: The training ground for internet congestion-control research," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 731–743.

[13] J. Kober and J. Peters, "Reinforcement learning in robotics: A survey," *International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[14] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *IEEE Network*, vol. 32, no. 2, pp. 92–99, 2018.

[15] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 3050–3059.

[16] J. Jiang, V. Sekar, I. Stoica, and H. Zhang, "Unleashing the potential of data-driven networking," 09 2017, pp. 110–126.

[17] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 197–210.

[18] M. Wang, Y. Cui, S. Xiao, G. Wang, D. Yang, K. Chen, and J. Zhu, "Neural network meets dcn: Traffic-driven topology adaptation with deep learning," 06 2018, pp. 97–99.

[19] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, 2017, pp. 185–191.

[20] "Ns2," https://www.isi.edu/nsnam/ns/.

[21] N. Jay, N. Rotman, P. Godfrey, M. Schapira, and A. Tamar, "Internet congestion control via deep reinforcement learning." NeurIPS18, 10 2018.

[22] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate record-and-replay for http," in *USENIX Annual Technical Conference (USENIX ATC)*, 2015, pp. 417–429.

[23] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 255–270, 11 2002.

[24] V.Sivakumar, T.Rocktaschel, A.H.Miller, H.Kuttler, N.Nardelli, M.Rabbat, J.Pineau, and S.Riedel., "Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions," *arXiv preprint arXiv:1910.04054*, 2019.

[25] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, R. Addanki, M. K. Shirkoohi, S. He, V. Nathan, F. Cangialosi, S. B. Venkatakrishnan, W.-H. Weng, S. Han, T. Kraska, and D. Alizadeh, "Park: An open platform for learning-augmented computer systems," in *NeurIPS 2019*, 2019.

[26] J. Fang, M. Ellis, B. Li, S. Liu, Y. Hosseinkashi, M. Revow, A. Sadovnikov, and et al., "Reinforcement learning for bandwidth estimation and congestion control in real-time communications," *arXiv preprint arXiv:1912.02222*, 2019.

[27] G. Brockman, V. Cheung, L. Pettersson, J.Schneider, J. Schulamn, J. Tang, and W. Zaremba, "Copenai gym," *CoRR*, vol. abs/1606.01540, 2016.

[28] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[29] "Pytorch," https://pytorch.org.

[30] "Redis," https://redis.io.

[31] "Onnx:open neural network exchange," https://github.com/onnx/onnx.

[32] "Onnx runtime: cross-platform, high performance scoring engine for ml models." https://github.com/microsoft/onnxruntime.

[33] "tc: Linux advanced routing and traffic control." http://lartc.org/lartc.html.

[34] V. Jacobson, "Modified tcp congestion avoidance algorithm," *end2end-interest mailing list*, 1990.

[35] M. Dong, Q. Li, D. Zarchy, B. Godfrey, and M. Schapira, "Pcc: Re-architecting congestion control for consistent high performance," in *Usenix Conference on Networked Systems Design and Implementation(NSDI)*, 2015, pp. 395–408.

[36] M. C.-J. Liang, H. Xue, M. Yang, and L. Zhou, "The case for learning-and-system co-design," *ACM SIGOPS Operating Systems Review*, vol. 53, no. 1, pp. 68–74, July 2019. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-case-for-learning-and-system-co-design/

[37] H. Mao, D. D. Shannon Chen, S. Singh, D. Blaisdell, Y. Tian, M. Alizadeh, and E. Bakshy, "Real-world video adaptation with reinforcement learning." ICML, 2019.

[38] G. Yi, D. Yang, M. Wang, W. Li, Y. Li, and Y. Cui, "L3vtp: A low-latency live video transmission platform," 08 2019, pp. 138–140.