DeepCC: Bridging the Gap Between Congestion Control and Applications via Multi-Objective Optimization

Lei Zhang, Yong Cui, *Member, IEEE,* Mowei Wang, Kewei Zhu, Yibo Zhu, *Member, IEEE,* and Yong Jiang, *Member, IEEE,*

Abstract—The increasingly complicated and diverse applications have distinct network performance demands, e.g., some desire high throughput while others require low latency. Traditional congestion controls (CC) have no perception of these demands. Consequently, literatures have explored the objective-specific algorithms, which are based on *either* offline training or online learning, to adapt to certain application demands. However, once generated, such algorithms are tailored to a specific performance objective function. Newly emerged performance demands in a changeable network environment require either expensive retraining (in the case of offline training), or manually redesigning a new objective function (in the case of online learning). To address this problem, we propose a novel architecture, DeepCC. It generates a CC agent that is generically applicable to a wide range of application requirements and network conditions. The key idea of DeepCC is to leverage both offline deep reinforcement learning and online fine-tuning. In the offline phase, instead of training towards a specific objective function, DeepCC trains its deep neural network model using multi-objective optimization. With the trained model, DeepCC offers near Pareto optimal policies *w.r.t* different user-specified trade-offs between throughput, delay, and loss rate without any redesigning or retraining. In addition, a quick online fine-tuning phase further helps DeepCC achieve the application-specific demands under dynamic network conditions. The simulation and real-world experiments show that DeepCC outperforms state-of-the-art schemes in a wide range of settings. DeepCC gains a higher target completion ratio of application requirements up to 67.4% than that of other schemes, even in an untrained environment.

Index Terms—Congestion Control; Data-Driven Networking; Multi-Objective Learning; Online Learning

I. INTRODUCTION

THE emerging applications in modern networks have very different performance requirements. Delay-sensitive applications, such as Internet telephony or cloud gaming, require a low transmission delay as low as a few milliseconds [1]. These applications may not benefit from higher bandwidth. On the other hand, the video streaming or file sharing, i.e., throughput-sensitive applications, often require high bandwidth for better performance [2]. In addition, some applications may provide the different specified demands of bandwidth and delay to satisfy users' quality of experience, such as some WebRTC-based applications [3]. Therefore, the transport layer should adapt to not only volatile network conditions, but also different application demands [4].

For the last thirty years, traditional TCP congestion controls (CC) have been dedicated to solving how to adapt to network conditions. However, they do not work well to satisfy various performance requirements due to their unaware of application demands. For example, Cubic [5] as the default CC algorithm in Linux kernel, uses a hardwired rule to regulate congestion windows (*cwnds*) and has no perception of application requirements. Many recently proposed CC algorithms, including Copa [2] and BBR [6], are designed based on their own understanding of throughput/latency trade-off and share the same limitation as Cubic. Some other CC algorithms [7]–[11], such as Sprout [7] and Verus [8], serve for specific applications or network conditions. They fall short in terms of generalization to adapt to various performance requirements.

With the diverse requirements of emerging applications, the learning-based CC becomes a research hotspot recently. These learning methods are well-suited to learning control policies without relying on inaccurate assumptions. Rather than using hardwired rules, these schemes define *one* objective function representing a single user-specified trade-off of requirement, and learn cwnds or sending rate by optimizing the specified function. For example, Remy [4] generates a decision tree to optimize an objective function of throughput and delay by *offline* learning. PCC [12] performs *online* exploring for the objective optimization of throughput and loss. However, they are limited to optimize only one objective function with fixed parameters while do not consider the different application-specific demands. Whenever any new application requirement emerges or network environment changes, the learning-based algorithms require carefully redesigning and retuning.

To address this problem, we propose a novel architecture, *DeepCC*, where the multi-objective congestion control for *various* performance goals is generated through machine learning methods. When running, it automatically adapts to application-specific demands and network conditions *without* redesigning or retraining efforts. DeepCC is not merely built upon off-the-shelf learning methods. Instead, DeepCC realizes this with two key ideas.

L. Zhang, Y. Cui, M. Wang, K. Zhu, and Y. Jiang are with the Department of Computer Science and Technology, Tsinghua University, Beijing 10084, China (e-mail: leizhang16@mails.tsinghua.edu.cn; cuiyong@tsinghua.edu.cn; wang.mowei@outlook.com;zkw18@mails.tsinghua.edu.cn; jiangy@sz.tsinghua.edu.cn).
Y. Zhu is with ByteDance Co., Ltd., Beijing 10080, China (e-mail: zhuyibo@bytedance.com).

First, DeepCC does not have a single fixed objective function during the training phase. The inflexibility of Remy and PCC is due to the fact that they only optimize one objective function during offline training or online learning. In contrast, DeepCC learns about how it should react to congestion with various objective functions and in various networks. DeepCC uses deep neural networks as its core model, which is well-known to be capable of handling multi-dimensional inputs and outputs. In DeepCC, these include the performance targets of throughput, delay, and packet loss, as well as the network conditions. DeepCC provides the largest possible flexibility to runtime – different applications can specify their own performance requirements during runtime, and even change them on demand. DeepCC will still work well without retraining.

Second, unlike the existing learning-based CC schemes, DeepCC leverages both offline deep reinforcement learning (DRL) and online fine-tuning. Offline and online approaches have their trade-offs. For example, offline approaches allow more flexible forms of objective functions but are weaker in adapting to network conditions. However, though being more adaptive to network conditions, online approaches must use special forms of objective functions [12] because they will impact the online learning speed and results. DeepCC aims to take the advantages of both approaches. It learns most of its knowledge through offline training, which is more focused on supporting flexible objective functions, while still includes an online fine-tuning phase that handles dynamic network conditions. This online fine-tuning is much more efficient than a purely online design because the results from offline training provide a good starting point and largely narrow down the search space. The main difference between DeepCC and other schemes are summarized in Table I.

To the best of our knowledge, DeepCC is the first congestion control that can optimize the multiple objective functions. It is also the first CC to combine both offline learning and online fine-tuning. Though this approach is less seen in the networking community, it is in fact popular in the machine learning research and has achieved many state-of-the-art results [13], [14]. The main contributions are listed as follows:

- We present a novel architecture named DeepCC, which can satisfy different performance requirements without redesigning or retraining efforts. It fully leverages the power of offline and online learning techniques that improve the generalization ability for both application requirements and network conditions (§III).
- We propose a multi-objective DRL algorithm to learn Pareto optimal (or near optimal) control policies for different performance trade-offs. Our solution efficiently explores the wide optimization objective space and offline learn an optimal (or near optimal) strategy (§IV-A and §IV-B).
- We design an online tuning algorithm for various application-specific demands and network conditions. It can dynamically choose one of the learned Pareto optimal policies to adapt to the real-time network conditions under the guidance of the specific demand. It greatly facilitates DeepCC to meet the explicit performance requirements under different network environments (§IV-C).

TABLE I
DIFFERENCES BETWEEN DEEPCC AND OTHER SCHEMES

Solution	Optimization objective	Offline/Online	Diverse requirements	TCR(%)
Cubic [5]	/	/	×	0.0
BBR [6]	/	/	×	0.0
Remy [4]	Single	Offline	×	44.20
PCC [12]	Single	Online	×	0.0
Vivace [16]	Single	Online	×	0.0
Indigo [17]	Single	Offline	×	3.20
DeepCC	Multiple	Offline&Online	√	67.40

We implement and evaluate DeepCC in the Mahimahi [15] emulator and real-world network. Compared against state-of-the-art schemes, DeepCC achieves a wide range of performance and gains a higher target completion ratio (TCR) of application requirements up to 67.4% than that of other schemes, even in an untrained environment (§V and §VI).

II. MOTIVATION AND CHALLENGE

Comparing with traditional congestion control schemes, learning-based approaches can adapt to network conditions, such as Remy [4], Indigo [17], PCC [12], and Vivace [16]. Nevertheless, poor generalization ability limits them to work with changing application-specific requirements and network conditions. In the following, we first summarize the limitations of the existing schemes. Then we detail the key challenges that are tackled by DeepCC's design.

A. Limitations of the existing schemes

We conduct experiments to evaluate the performance of existing state-of-the-art schemes[1] in cellular and Wi-Fi links. We plot the distribution of throughput and 95th percentile queueing delay of them in Fig. 1. We then highlight the limitations with illustrative examples.

Limitation 1: Existing schemes can not generalize for diverse application requirements.

Manual policies, such as Cubic [5], allow the senders to deliver data based on heuristics rather than performance requirements. Application-specific optimization schemes, such as Remy and PCC, often do not have general applicability for diverse performance requirements. A fine-tuned scheme for one application works poorly for other applications if performance requirements change.

In principle, the learning-based schemes can support application requirements by designing their objective functions. However, the existing schemes, such as Remy or Vivace, at least in their current form, only provide three choices of objective functions. As a direct consequence, they work in some isolated operating points. As shown in Fig. 1, the family of Vivace can only operate on the high-throughput area (the left-top) and cannot achieve low latency, while that of Remy fails to escape from the scope of low-latency area (the right-bottom). None of them could cover the Pareto front [18].

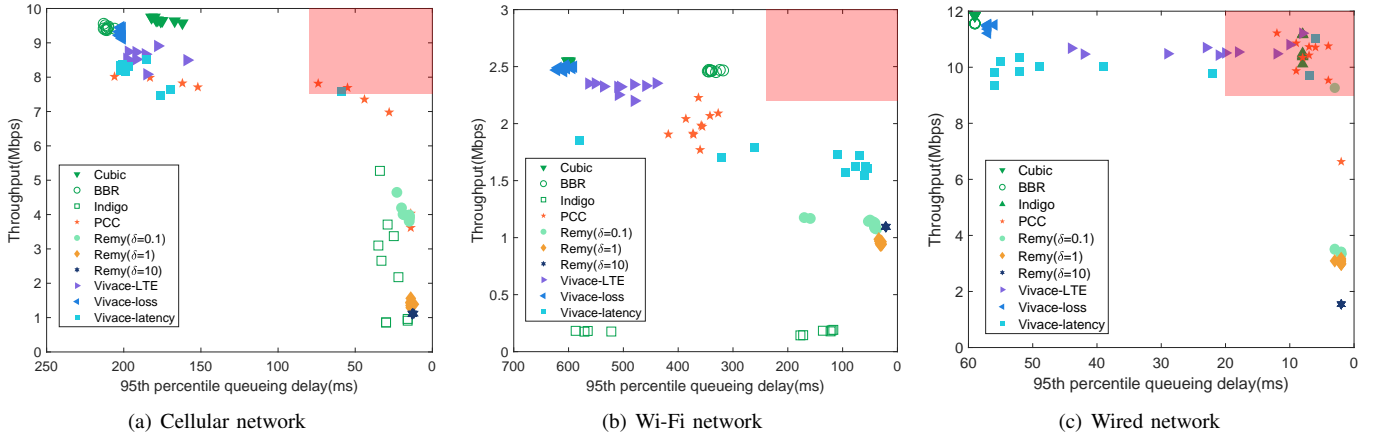[1]The source codes are provided by Pantheon [17].

Fig. 1. Throughput vs. 95th percentile queueing delay achieved by the existing CC with their objective functions under three network links. All schemes are repeatedly tested 10 times and each test lasts for 50 seconds.

Although Remy and Vivace can behave on different trade-offs with their three objective-specific functions, the performance range is quite limited so that they can not satisfy diverse requirements.

Even if there are infinite objective functions available that could achieve any trade-offs, it is still non-trivial for the applications to use such a learning-based scheme. The main reason is that none of these schemes provide a systematic approach that could guide the CC to select the suitable objective function to meet a specific quality of service (QoS) requirement (e.g., 8 Mbps throughput while keeping the delay within 70 ms and the loss rate within 1%, the shaded area as shown in Fig. 1(a)). Moreover, the average queueing delay (calculated as the difference between the observed round-trip time, i.e., RTT, and the minimum RTT) and throughput of a single scheme with one objective function can be highly variable under a single network scenario in Fig. 1(a), such as PCC or Indigo. Hence, in the view of the application providers, the existing learning-based schemes may be still far from satisfactory.

Limitation 2: Existing learning-based schemes cannot generalize well across a wide range of network conditions.

Except the application-specific demands, congestion control needs to adapt to a wide variety of heterogeneous network conditions. The existing online or offline schemes can benefit from their own properties but still face great limitations.

Schemes relying on offline learning inherently face the generalization problem. The model or agent (i.e., the learned control rules) can only be trained with limited data that cannot cover all the network conditions and thus may overfit the dataset, at least to some extent. For example, Indigo could attain low latency within its design scope in Fig.1(c). However, its performance can be degraded when the actual network conditions mismatch the training assumptions [19] as shown in Fig.1(b) .

Although the online schemes perform at least acceptably when facing new network condition, the convergence time is a fundamental problem for them. We compare the convergence behavior of the above-mentioned schemes as shown in Fig. 15. It is similar to the results that Vivace claims - PCC converges slowest [16]. In general, the offline schemes are more stable

than online schemes [20]. This is because the offline schemes can leverage prior knowledge of networks obtained during training.

B. Challenge

As many have observed, learning-based congestion control schemes have emerged in support to adapt to complicated network conditions. Although these schemes deliver a satisfactory performance of a single specific requirement, they still have limitations and fail to achieve different trade-offs of performance requirements. Along this direction, DeepCC is designed to tackle the following key challenges.

(1) Guaranteeing the application-specific demands. It is important to guarantee the specified demands for the applications. However, achieving the specified performance is non-trivial because CC schemes achieve the performance unpredictably under complicated network conditions. Further, learning-based schemes could provide a desired trade-off for one or a class of applications by defining a fine-tuned objective function, but it is difficult to achieve the deterministic performance demands defined by applications or users.

(2) Huge space of optimization objectives. Each metric of application requirements could span on a large scale. The diverse requirements with multiple dimensions that fall into different trade-offs of the performance metrics further make it harder to deal with. The huge space of optimization objectives poses a great challenge, especially for reinforcement learning, which must "explore" the action space in training to learn a good policy for each optimization objective.

(3) Poor generalization for learning-based schemes. Online learning has the ability to adapt to the environment quickly, whereas the convergence time is too long. By contrast, the learned agent or model through offline learning, which is provided a good starting point by offline learning, can perform well in the scenarios that are similar to the training environments. But it could fail to adapt to unseen network conditions.

III. DESIGN OVERVIEW

In this work, we seek to close the gap between the congestion control and different applications' requirements
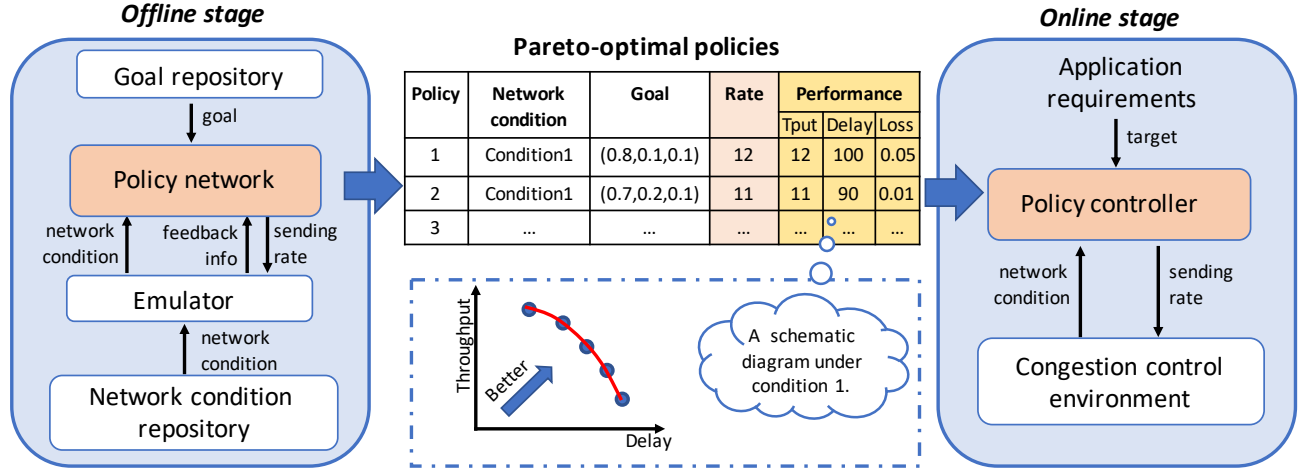
Fig. 2. The DeepCC architecture.

by proposing DeepCC, a multi-objective CC with various optimization *goals* that can adapt to meet different *targets* of application-specific demands (See Table II for an explanation of the notation). DeepCC does not need to modify the TCP protocol and only obtains the QoS requirements from the upper layer applications. DeepCC is not merely built upon off-the-shelf reinforcement learning approaches. Instead, it leverages several ideas to solve the above-mentioned challenges.

In conventional learning-based congestion control, the agent can collect network states from environments and learn to improve its output action based on its fine-tuned optimization objective. However, this approach is unaware of the different application demands so that its achieved performance can be away from the actual application requirements. To cope with the problem, we propose an online algorithm to fulfill the application-specific demands (e.g., 10 Mbps throughput while keeping the delay within 100 ms and the loss rate within 1%). The online algorithm allows DeepCC to adaptively tune the policy according to the application demands.

The learning-based agent can explore diverse network environments to enrich its experience and try to meet all metrics of application requirements, including different throughput, delay, and loss rate. However, the huge space of optimization objectives makes it very difficult to train an optimal or near-optimal agent for massive application requirements. To handle this problem, DeepCC does not directly optimize for diverse performance requirements instead it optimizes for different trade-offs of performance metrics as an intermediate objective, using the weights to represent different trade-offs. Specifically, DeepCC leverages multi-objective optimization to offline generate a flexible agent that is able to learn the Pareto optimal or near-optimal policies and be tuned for different objectives after training.

It is well known that poor generalization is a key problem of learning-based schemes. To mitigate the generalization problem, we propose the two-stage learning architecture which leverages the benefit from both offline and online learning. In our cases, even if the same application requirements, DeepCC should adopt different policies in different network

TABLE II
NOTATION

Name	Description	Symbol
State	the last action	$s^{(1)}$
	an exponential weighted moving average of the current RTT	$s^{(2)}$
	an actual sending rate	$s^{(3)}$
	the average delay in each decision interval	$s^{(4)}$
Action	the sending rate	a
Reward	the compound of measurement and goal	r
Goal	the relative weight of throughput	$g^{(1)}$
	the relative weight of delay	$g^{(2)}$
	the relative weight of packets loss rate	$g^{(3)}$
Measurement	the normalized throughput	$m^{(1)}$
	the normalized delay	$m^{(2)}$
	the observed packet loss rate	$m^{(3)}$
Target	the requirement of minimum throughput	$T^{(1)}$
	the requirement of maximum delay	$T^{(2)}$
	the requirement of maximum loss rate	$T^{(3)}$

environments. In this way, DeepCC continuously adjusts the weights of the model obtained from offline training through online tuning algorithm to meet the application's target under different environments. Fig. 2 shows the high-level overview of DeepCC's design which contains two stages.

In the offline stage, DeepCC learns a set of Pareto optimal policies (i.e., the policy network with different goals as input condition in Fig. 2) under different network conditions. Considering the diverse performance requirements, we start by defining a group of weights, which expresses the different trade-offs (termed *goal*) of the relative preferences for throughput, delay, and loss rate (§IV-A). The policy network takes the goal and the network conditions (termed *state*) as input and outputs the sending rate. DeepCC trains the policy network to optimize for different goals through a large number of offline experiments from the emulator (§IV-B). The well-learned policy network can build a good relationship between the input conditions and the sending rate over all possible

performance trade-offs.

In the online stage, DeepCC matches the network condition and performance requirement (termed *target*) to the sending rate (termed *action*) using the learned Pareto optimal policies. At the start of the connection establishment, applications provide their requirements of bandwidth, delay, and packet loss to the policy controller (§IV-C). At runtime, the controller continuously detects the changes in the difference between current performance and requirements, and automatically chooses the most proper policy (represented by *goal*) that can best fit the requirements. Then the best sending rate is decided according to the selected policy.

IV. DETAILED DESIGN

In this section, we present the detailed design of DeepCC. We begin with describing the multi-objective function. Then we explain the offline training process with multi-objective optimization and the online tuning algorithm.

A. Representing multi-objective function

Note that it is non-trivial to directly learn to achieve the various performance requirements offline, since the application requirements may fall in a large performance space especially when considering different network conditions. Hence, we use the relative instead of the absolute value to express our objective function in offline learning. Further, the *goal* as a relative weight of the performance metrics can become a direct "knob" to be tuned by users or applications online to achieve the application desired performance.

The multi-objective expression includes not only multi-dimensional performance metrics, i.e., throughput, delay, and loss rate, but multiple trade-offs between them. The multi-objective function is composed of measurement and goal. Among them, the *measurement* indicates the current transmission performance that includes the throughput, delay, and loss rate. Table II summarizes these symbols.

Measurement. The measurement m is an n-dimensional vector as $m = (m^{(1)}, m^{(2)}, \cdots, m^{(n)})$. Considering the performance metric after the action taken in congestion control, we set $n = 3$ and the m_t at time step t as:

$$m_t = \left(\frac{throughput_t}{throughput_{max}}, \frac{delay_t}{delay_{min}}, loss\ rate_t\right) \quad (1)$$

At time step t, the $throughput_t$ is the instantaneous observed total throughput of the sender and the $throughput_{max}$ is the maximum value among all the history throughput; $delay_{min}$ is the minimum delay of the current connection; $delay_t$ is the 95th percentile delay; $loss\ rate_t$ is the observed loss rate.

Goal. The goal g_t is also an n-dimensional vector as:

$$g_t = (g_t^{(1)}, g_t^{(2)}, \cdots, g_t^{(n)}) \qquad s.t. \sum_{i=1}^{n} g_t^{(i)} = 1 \quad (2)$$

where $g_t^{(i)}$ is the relative weights of the corresponding performance metric at time step t. And the sum of all $g_t^{(i)}$ equals one. In CC problem, the *goal* represents the different trade-offs between throughput, delay and loss rate, i.e., $n = 3$. The larger $g_t^{(1)}$ signifies that higher throughput is preferable. The
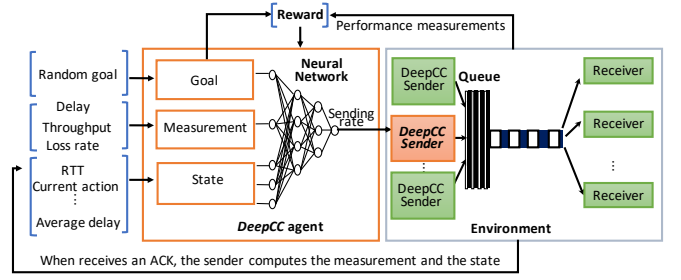

Fig. 3.   Offline learning with multi-objective optimization.

larger $g_t^{(2)}$ and $g_t^{(3)}$ indicate that the lower delay and loss rate are preferable respectively.

Reward. Similar to the reward function of DRL, Remy and PCC use the objective function or the utility function to evaluate the transmission performance and take them as the feedback that helps decision making. However, they only use two of the three performance metrics. Specifically, their objective function includes multiple dimensions rather than the different trade-offs between the performance metrics.

In our approach, the multi-objective function, i.e., *reward*, is set to reflect the desired performance of throughput, delay, and loss rate that we wish to optimize under different relative weights. So we set the reward r_t as the compound of measurement and goal at each time step t. For congestion control, we set the performance of throughput as an award, while the performance of delay and loss rate as a penalty. Therefore, when computing a reward, the 2nd-dimension and 3rd-dimension of measurement use the *negative* value of them.

$$r_t = g_t^{(1)} m_t^{(1)} - g_t^{(2)} m_t^{(2)} - g_t^{(3)} m_t^{(3)}/threshold \quad (3)$$

The $threshold$ represents the tolerance of packet loss. By explicitly introducing multi-objective reward, the agent learns multiple objectives with different goals. Moreover, the decision-making does not depend on the intermediate reward in one step, but takes the expected cumulative reward $J_{mul} = \mathbb{E}[\sum_{t=0}^{N} \gamma^t r_t]$ as the objective, where $\gamma \in (0,1)$ is a discount factor and N is the total steps.

B. Offline learning with multi-objective DRL

We consider a DeepCC agent generated by the multi-objective DRL through offline learning, which can potentially deal with the multiple objectives, the continuous decision space, and the adaptation problem by leveraging the great power of deep neural networks (NNs). Unfortunately, the basic Deterministic Policy Gradient (DDPG) algorithm [21], an advanced DRL algorithm that deals with the continuous action space, could not directly support for DeepCC with multi-objective optimization due to its limited expressiveness of the fixed scalar reward. Recently, some novel multi-objective reinforcement learning algorithms have been proposed, such as DFP [22] for the game Doom and UVFAs [23] for Atari games. However, both DFP and UVFAs cannot directly be applied to continuous control problems.

Hence, we design a multi-objective DDPG algorithm based on basic DDPG architecture. Specifically, we use a multi-objective function as the reward function of DeepCC (provided
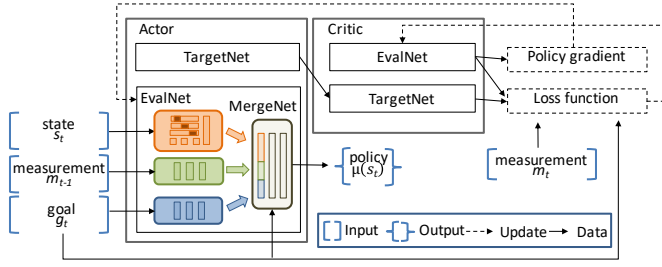
Fig. 4. Multi-objective DDPG optimization.

by §IV-A), expressed as the combination of the goal and performance metric measurement. In the offline training process, we set a *random goal* for each training episode to achieve a wide range of trade-offs between the performance metrics.

Unlike traditional CC schemes that use hardwired rules to regulate cwnds or sending rate, DeepCC agent learns the flexible policy of sending rate directly from the agent-environment interactions. As shown in Fig. 3, the input of agent includes not only network *state* but also *measurement* and *goal*. Then the agent derives the proper *action*, i.e., sending rate, when receiving an ACK. Moreover, the multi-objective function of the agent uses the compounded information of measurement and goal which is used to train and improve the neural network model. The ultimate aim of the learning algorithm is to maximize the expected cumulative discounted reward. Leveraging the power of DNNs, the agent can learn near-optimal control policies that mapping state, measurement, and goal to the action for each optimization objective.

Formulating congestion control problem as a DRL task in DeepCC requires specifying the state and action.

State space. When the sender receives an ACK, the agent observes the current RTT and computes the history statistics. We narrow our attention to some statistics and observations as the state $s_t = (s_t^{(1)}, s_t^{(2)}, s_t^{(3)}, s_t^{(4)})$ that may facilitate the CC decisions. The detailed descriptions of $s_t^{(i)}$ are in Table II.

Action space. We choose the sending rate, a continuous variable, as the action of the agent. After receives s_t, the agent takes the action a_t, i.e., the sending rate. The action is selected by a policy $\mu(s_t)$ which is defined as a deterministic action $a_t \in [0, bound]$, where $bound$ is the upper limitation of the sending rate. We use NNs to represent the policy with a manageable number of adjustable parameters θ^μ.

Neural network architecture. We design a multi-objective DDPG algorithm, which involves four deep NNs and exploits the actor-critic algorithm to train the policies on continuous action space. In contrast to basic DDPG architecture, as shown in Fig.4, not only the measurement and goal as inputs are added to the agent, but the goal vector is added to each layer of NNs. These explicitly added inputs increase the sensitivity of decision policy to different optimization objectives.

Training with multi-objective optimization. We now describe the policy gradient training algorithm of DeepCC. The algorithm uses two evaluation networks (termed *evalNet*) to approximate the actor function $\mu(s, m, g|\theta^\mu)^2$ and the critic

^{2}In the following, we use $\mu(s|\theta^\mu)$ as a shorthand when no ambiguity exists.

function $Q^\mu(s, g, a|\theta^Q)$, respectively. The two target networks (termed *targetNet*), $\mu(s|\theta^{\mu'})$ and $Q^\mu(s, g, a|\theta^{Q'})$ are the copies of the evalNets accordingly. During training, only the two evalNets are trained in each time step, while the parameters of targetNets are updated by slowly tracking the evalNets: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau \ll 1$. The critic function can be described in a recursive expression:

$$Q^\mu(s_t, g_t, a_t|\theta^\mu) = \mathbb{E}[r_t + \gamma Q^\mu(s_{t+1}, g_{t+1}, \mu(s_{t+1}))] \quad (4)$$

where γ is the discount factor and r_t is the instant reward which is expressed as the combination of m_t and g_t. Recall from the definition of action space, the actor function $\mu(s|\theta^\mu)$ specifies the current policy by deterministically mapping inputs to a specific action. It can be updated by applying the chain rule to the expected cumulative reward J_{mul} with a refection to the actor parameters θ^μ:

$$\nabla_{\theta^\mu} J_{mul} \approx \frac{1}{N} \sum_t \nabla_a Q(s_t, g_t, a|\theta^Q)|_{a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s_t, g_t|\theta^\mu)$$
$$(5)$$

where N is the training batch size.

The objective of the training is to maximize J_{mul} and minimize the loss L of critic network, which is defined as:

$$L = \frac{1}{N} \sum_t (y_t - Q(s_t, g_t, a_t|\theta^Q))^2 \quad (6)$$

where y_t is estimated by:

$$y_t = r_t + \gamma Q'(s_{t+1}, g_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}) \quad (7)$$

To achieve the multi-objective training, we randomly set the goal vector g_t for each training episode. Therefore, the DeepCC could successfully generalize across a wide range of trade-offs between different performance metrics. When L tends to zero and the average J_{mul} of a batch size data no longer increases, the agent is considered to converge.

As a summary, the actor and critic function are trained by the policy gradient and value-based method respectively. After training, the actor function parameterized by θ^μ will converge to different near-optimal policies corresponding to different objectives with the help of the critic function. Once being trained, the agent learns near Pareto front with different goals under different network conditions (See §VI-A).

C. Online DeepCC tuning

How to adaptively choose the proper policy form learned Pareto optimal policies remains a challenge. To address this issue, we design a policy controller, i.e., an online tuning module to achieve the requirement about throughput, delay, and loss rate, i.e., the *target*. Our key insight is that we can regard the action regulation as a black box where the input includes state, measurement, and goal driven by the ACK signal. Then the *goal* performs as a knob that controls the agent to pursue different targets. Specifically, DeepCC continuously detects the changes in the difference between current measured performance and application requirements, and automatically tunes the *goal* with multi-dimensional gradient descent.

During a TCP session, we adopt the multi-dimensional gradient descent algorithm to tune the agent online. According to

the changes in network conditions and the prior measurements, the multi-dimensional gradients about the distance between measurement and target value are computed and applied to generate a new *goal* for the agent. Thereby the current measurement can be fed to enable a negative-feedback loop to influence the action choice until the performance converges to the application requirements under unseen network conditions. **Multi-dimensional gradient descent algorithm.** The online tuning algorithm is based on multi-dimensional gradient descent. Specifically, the objective of the online tuning algorithm is to minimize the loss function $J = Euclid(m_t, T)$ with respect to the goal g_t. Here, (1) m_t represents the performance under the corresponding goal g_t at time step t. In this way, we suppose that the measurement can be regarded as a function of g_t, i.e., $m_t := f(g_t)$; (2) the n-dimensional target value $T = (T^{(1)}, T^{(2)}, \cdots, T^{(n)})$ is specified by the application;(3) $Euclid(m_t, T)$ represents the euclidean distance between the measurements and the target value.

To reduce the variance of the estimated value m_t, we set a k-step average of J as the loss function in practice. Taking the relationship between m_t and g_t into account, the actual loss function is described as:

$$J(g_t) = \frac{1}{2k} \sum_{i=t-k}^{t} (Euclid(f(g_t), T)) \tag{8}$$

In this paper, the target value $T = (T^{(1)}, T^{(2)}, T^{(3)})$ is defined as the desired max-min bound for each dimensional performance metric. The detailed descriptions of $T^{(i)}$ are shown in Table II. In practice, if $m^{(i)}$ is satisfied the bound of $T^{(i)}$, we set $f(g_t^{(i)}) = T^{(i)}$. Namely, we only update gradients when the target requirements are not satisfied. If the application can not provide the concrete performance target but have a performance preference, e.g., the high throughput or low delay, the target T could be presented as an $(n\text{-}1)$-dimensional vector with the default value. In this case, the target T is not related to the uninterested metric. Under the low-delay mode, the target T is set as the minimum RTT and zero packet loss. While in the high-throughput mode, the target T can be defined as the observed maximum bandwidth and zero packet loss.

We use a gradient descent method to update the g_t. The gradient of J with respect to g_t can be derived as:

$$\bigtriangledown_{g_t} J(g_t) = \bigtriangledown_{f(g_t)} J(f(g_t)) \bigtriangledown_{g_t} f(g_t) \tag{9}$$

Since it is non-trivial to model the relationship between the goal g_t and the measurement m_t, we can not easily obtain the analytic expression of the function f. Here we use a numerical method to approximate $\bigtriangledown_{g_t} f(g_t)$:

$$\bigtriangledown_{g_t} f(g_t) \approx \frac{1}{k} \sum_{i=t-k}^{t-1} \frac{\Delta m_i}{\Delta g_i} = \frac{1}{k} \sum_{i=t-k}^{t-1} \frac{m_{i+1} - m_i}{g_{i+1} - g_i} \tag{10}$$

Finally, the goal can be updated by the following formula.

$$g_{t+1} \leftarrow g_t - \alpha * \bigtriangledown_{g_t} J(g_t) \tag{11}$$

where α is the learning rate.

Since the specific function form of $f(g_t)$ is unknown and the function is not necessarily a convex function, the traditional gradient descent method may fall into local optimum, resulting in performance degradation. To solve this problem, we design a simple rule-based approach to help "jump out" of the local optimum. In practical terms, if the distance between the ith-dimensional measurement and target is larger than a threshold, the ith-dimensional goal is increased by a pre-setting value.

Moreover, non-congestion packet loss is a common phenomenon in the Internet [16]. The goal of loss rate (i.e., $g^{(3)}$) should not change due to random packet loss. For DeepCC, we set the constraint for the gradient of the loss rate. That is to say, if the target is not satisfied with the measurement of throughput or delay, the gradient of loss rate will be set to equal that of delay. The above method ensures that the gradient of loss is not affected by non-congestion packet loss since it always follows the gradient of delay.

Since the sum of each dimension of goal vector is 1 in the training process, each dimension of updated goal $g^{(i)}$ will be normalized before fed into the agent as follow:

$$g^{(i)} = \frac{g^{(i)}}{\sum_{i=1}^{n} g^{(i)}} \tag{12}$$

In fact, it could cause a large variance of goal and huge fluctuations in the convergence process when updating g_t directly following e.q. (11). Additionally, it is difficult to choose the proper learning rate α. An adaptive learning rate algorithm Adam [24] is a first-order optimization algorithm that is less sensitive to the choice of learning rate than the basic gradient descent algorithm. It also has advantages in non-convex optimization problems. Therefore, we combine Adam algorithm to update the gradient $\bigtriangledown_{g_t} f(g_t)$ in practice.

V. IMPLEMENTATION AND TRAINING

In this section, we describe the DeepCC implementation, training and the interface.

DeepCC implementation. We implement the sender and receiver in the user space by adopting the UDP-based transmission skeleton, like Indigo [17]. The sending and receiving events are implemented by the message-triggered mechanism. DeepCC replaces the sender-side congestion control with the offline-learned agent and the online tuning module. The sender program receives the targets from the applications, executes the action, and controls the packets sending behavior. Unlike the sender, DeepCC remains the receiver unchanged.

For practical implementation, the sender firstly loads the multi-objective agent and sets the default *goal* according to the target. Every time step the sender received an ACK message, it updates the estimated measurement and state. Then it infers the next-step sending rate via the agent with the *goal*, measurement, and state. Once getting the sending rate, the sender can calculate the new *cwnd* size and pace these packets in an ack-clock. In the process of sending, if the current throughput and delay do not reach the target, the sender tunes the *goal* according to the online tuning algorithm.

The overhead of DeepCC mainly lies in obtaining the action and updating the goal, since the neural networks and the online tuning algorithm introduce extra complexity at the endpoint. To balance the overhead and efficiency, DeepCC triggers the

model inference and goal tuning every interval instead of at the packet level. Specifically, the model inference and goal tuning are triggered when an ACK is received and the time since the last decision exceeds the decision interval, e.g., 1/2 RTT and 4 RTT respectively. Furthermore, DeepCC takes an asynchronous interaction between the model inference, goal tuning, and packet sending so that they do not need to wait for other executions.

Training. We implement and train the multi-objective agent in Python with Tensorflow [25] for ease of development. To learn the control policies in the actor network, DeepCC first separately extracts the feature from the three inputs, i.e., state, measurement, and goal. The three inputs employ different neural network structures. States with 16 past values are passed to a 1D convolution neural network with 32 filters, each of size 3 with stride 1. The measurement and goal networks use three fully connected layers. Then mergeNet takes all the processed features from the above three networks and outputs the sending rate with the "tanh" activation function. The critic network takes the state, goal, and action as inputs and uses the same architecture as the actor network to conduct feature extraction. The extracted features of action are also processed by three fully connected layers, which is then fed to the last layer of the critic network. The difference between the actor and critic networks is that the final output of critic network is a linear neuron without activation function.

The agent is trained under the network environment where the fluctuation of bandwidth follows Poisson distribution, like the Wi-Fi link (See §VI-A). Once the multi-objective agent is well trained offline, this agent will not be retrained in practical deployment.

Interface. In order to satisfy diverse requirements, DeepCC provides the interface for applications to set their specified performance targets or preferences. This allows CC to directly perceive application requirements. The interface takes two forms. On one hand, applications can specify the target value of their required throughput, delay, and packet loss rate. The performance requirements provide optimization objectives for the online tuning algorithm. On the other hand, if the applications cannot provide the value of requirements, they can choose their performance preferences through the interface with the second form. In this case, the application can choose the high-throughput mode or low-latency mode (§IV-C).

VI. Evaluation

In this section, we demonstrate the advantages of DeepCC's multi-objective design in both emulator and real-world scenarios over various state-of-the-art schemes. In particular, we illustrate DeepCC's multi-objective high performance in different networks (§VI-A), high target completion ratio (§VI-B), decision interval and overhead (§VI-C), friendliness and fairness (§VI-D), and robustness (§VI-E).

A. Offline behavior over multi-objective optimization

To validate the effectiveness of multi-objective optimization, we repeatedly run DeepCC's agent under three network scenarios with different *goals* in a broad range using Mahimahi [15].

Among them, we select 5 representative results, which are in the mode of high throughput, low delay, or the trade-offs. We also compared DeepCC with the off-the-shelf schemes. As shown in Fig. 5(a) and (b), we intuitively visualize the performance of different schemes in a 2D throughput-delay space. The corresponding loss rate is shown via the color filled.

Emulated Wi-Fi link. First, we test DeepCC over an emulated Wi-Fi link (from Pantheon [17]) with high bandwidth variations, which DeepCC has been trained on. The link has an average 2.64 Mbps bandwidth following Poisson distribution, 176 ms minimum RTT and a drop-tail queue with 130 packets of buffer. As shown in Fig. 5(a), DeepCC with 5 representative goals achieves an efficient frontier that covers a wide range of trade-offs between throughput, queueing delay, and loss rate.

Specifically, DeepCC with g_1 in high-throughput mode achieves comparable throughput with the state-of-the-art CCs, while reduces the queueing delay by up to 75% with almost zero packet loss. When running in the low-latency mode, DeepCC with g_5 performs almost the same as Remy. Other trade-off points stand in the middle and attain different trade-offs. These results show the efficiency of our multi-objective optimization. So DeepCC can achieve a large range of accessible trade-offs only by tuning the goal without any retraining or redesigning efforts and it achieves almost zero packet loss without sacrificing other performance metrics due to the high penalty of loss rate in the reward function.

Emulated cellular link. To evaluate the performance of DeepCC in an unseen network link, we replay the AT&T driving trace provided by [7]. The cellular link has 200 ms minimum RTT and 140 packets of buffer. As shown in Fig. 5(b), the performance of DeepCC with different goals also achieves an efficient frontier. This results indicates that through training DeepCC has learned an adaptive policy supporting multi-objective optimization even under an unseen scenario. However, to achieve a similar relative position of Fig. 5(a), DeepCC may require different goals. This phenomenon exposes the challenge on how to set the proper goal according to the network condition, which will be described in the following with our online tuning algorithm.

Emulated satellite link. We also evaluate DeepCC'agent on an emulated satellite link with the same setup in PCC [12] and Copa [2] papers. The satellite link has 42 Mbps capacity, 800 ms RTT, 1 BDP (bandwidth-delay product) buffer and 0.74% stochastic loss rate. Fig. 5(c) shows that DeepCC with g_1 in max-throughput mode obtains higher throughput than that of other schemes. DeepCC with g_5 in delay-sensitive mode achieves lowest queueing delay. BBR which performs well in the above two scenarios, can not achieve persistent performance in this scenario. Furthermore, DeepCC with g_2 outperforms PCC and Vivace with much higher throughput and lower queueing delay.

As shown in the above experiments, DeepCC can reach a wide range of performance by flexibly adjusting the goal. DeepCC can improve the throughput from a low-throughput point to a high-throughput one up to 9X. Likewise, it can reduce the queueing delay from a high-latency point to a low-latency one up to 10X.
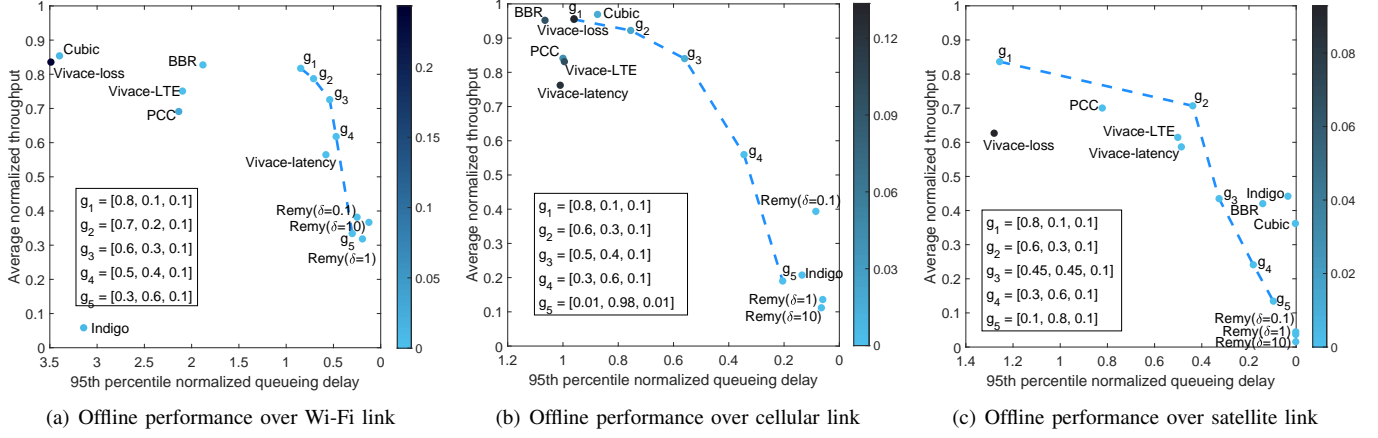
Fig. 5. Performance frontier achieved by offline learned DeepCC agent. The color bar represents the packet loss rate where darker is worse.
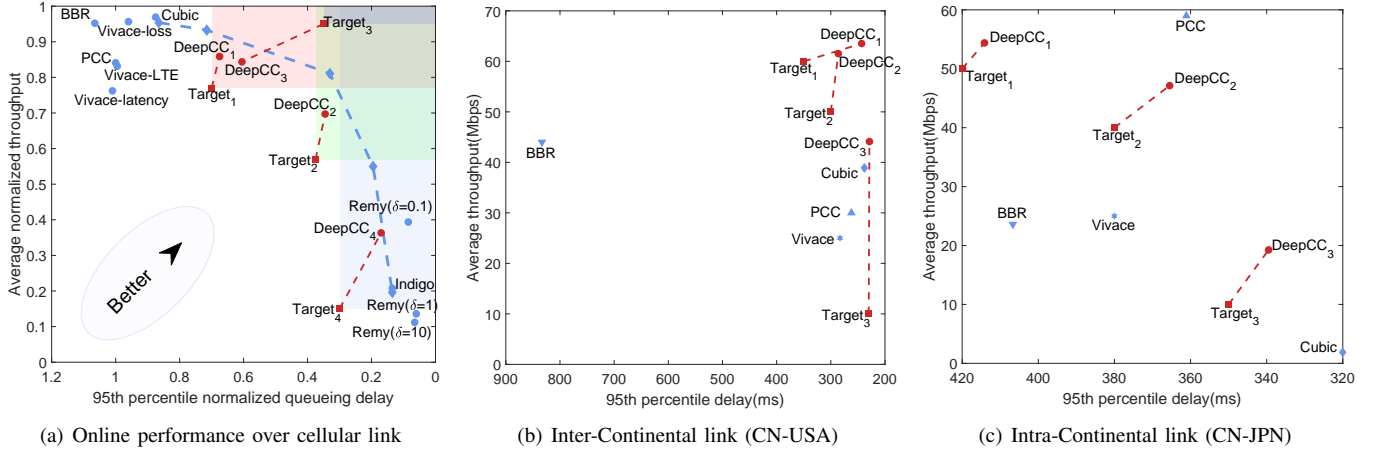
(a) Offline performance over Wi-Fi link   (b) Offline performance over cellular link   (c) Offline performance over satellite link

(a) Online performance over cellular link   (b) Inter-Continental link (CN-USA)   (c) Intra-Continental link (CN-JPN)

Fig. 6. Online performance frontier achieved by DeepCC. The latter two experiments are conducted over real-world network links.

B. Online performance over specified requirements

Emulated cellular link. As shown in Fig. 6(a), DeepCC gains an efficient performance frontier again in the online stage. Compared with the frontier achieved by offline learning shown in Fig. 5(b), DeepCC with online tuning achieves a wider frontier than purely through offline learning. That is because DeepCC takes advantage of both offline and online learning. When running online, it can timely regulate the near-optimal policy based on the offline trained agent according to the real-time network conditions. To provide a zoom-in view of online tuning, we carefully select four targets, among whom three are achievable (i.e., inside the frontier) and one is unachievable (i.e., outside the frontier). The shadow area of each achievable target represent the satisfactory region of the corresponding application. Note that the region under the frontier of three achievable targets are non-overlap so that they can not be satisfied with DeepCC with a single goal. DeepCC achieves them with proper goal-tuning through its online learning algorithm. For the unreachable target, DeepCC tries to achieve the performance close to the frontier.

Evaluation in the wild. We evaluate DeepCC over wide-area Internet paths spanning two continents. The senders are located in an Amazon [26] cloud server in USA and a server

located in Japan. The receiver is located inside our campus and connected to the senders through the wide-area network. We evaluate DeepCC with three *targets* and compare them against PCC, Vivace, Cubic and BBR [3]. The results are shown in Fig. 6(b) and Fig. 6(c). The average throughput and 95th percentile delay achieved by DeepCC are significantly different with three requirements. Guided by the *target* value, DeepCC can not only satisfy the corresponding requirement but achieve better performance in both throughput and delay. The delay of DeepCC with low-latency targets achieve significantly lower than that of BBR, PCC and Vivace. Notice that BBR, PCC and high-throughput DeepCC are not sensitive to packets loss, so that they both do well on throughput but high-throughput DeepCC achieves lower delay than that of BBR. **Target completion ratio.** To investigate the generalization ability of DeepCC in online stage, we randomly sample 500 target values T of application requirements over above-mentioned Wi-Fi link and cellular link. The range of these target values T is selected as follows: $T^{(1)}$ ranges from 0 to the achievable maximum throughput of all learning-based schemes on the corresponding link; $T^{(2)}$ varies from 0 to the achieved

---

[3]Cubic and BBR are implemented in Linux kernel 4.15.0.

(a) The target completion ratio over trained network link.

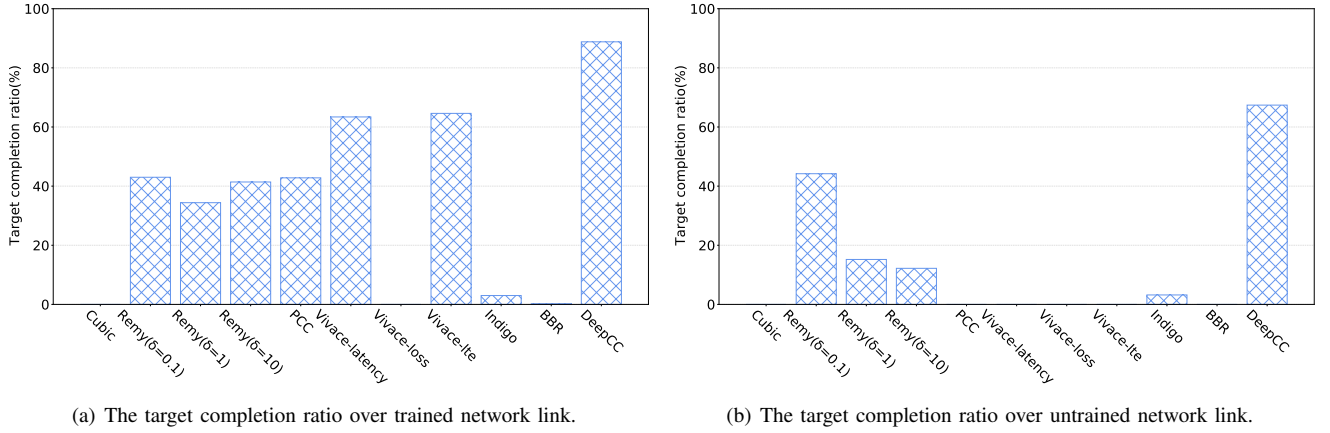(b) The target completion ratio over untrained network link.

Fig. 7. The target completion rate under 500 random targets of different congestion controls.

maximum queueing delay of all schemes on the corresponding link; and $T^{(3)}$, representing the loss rate, ranges from 0 to 10%. Obviously, some of them can not be achieved by any one of the existing schemes.

Here, we denote the *target completion ratio* (TCR) as the ratio of target achieved to all targets as our quantitative evaluation metric. Intuitively, for a given congestion control scheme, the higher completion ratio is obtained, the more application requirements can be achieved. Since all the existing learning-based schemes lack tunable objectives, we test each scheme with its default objective under the given network link and calculate its completion ratio. For DeepCC, we test it with different targets and leave the agent unchanged. The results are shown in Fig. 7(a) and Fig. 7(b). As expected, DeepCC achieves the highest target completion ratio among all the schemes. Even in an untrained network scenario, DeepCC still achieves 67.4% completion ratio. The results indicate that DeepCC can tune the objective function to adapt to both network conditions and application requirements through adjusting the *goal* .

C. Decision interval and overhead

DeepCC agent is trained on a PC with an Intel Core (TM) i7-6850k 3.6 GHz with 8 GB memory. It requires approximately two million training iterations to obtain the near Pareto optimal policies that perform well enough. In total, the training process took about 15 hours. At runtime, it takes about 0.5∼0.9 ms on average to get the *action* (i.e., the model inference time) or update the *goal*. We carefully set the decision interval as 1/2 minimum RTT and the online tuning interval as four RTT.

Here, we investigate the impact of different decision intervals on the performance. Intuitively, the decision interval determines the frequency of model inference. To evaluate the impact on average throughput, we set different intervals across different RTTs environments. Fig.8 depicts the results over different decision intervals. There are two takeaways here. First, fixed decision intervals are not advisable. The larger or smaller value of the decision interval can not achieve better performance. Second, the decision interval should change

dynamically in different RTT environments. According to the results, we empirically set 1/2 minimum RTT as DeepCC's decision interval.

Online tuning interval determines the execution frequency of the gradient descent. If the interval is too large, DeepCC cannot adjust the policy in time according to the network condition to satisfy application requirements. On the other hand, if the interval is too small, DeepCC will adjust its policy too frequently thus incurring unstablizaiton. To evaluate the impact of tuning intervals, we set experiments running the online tuning module with different intervals from half to 8 RTTs. The results in Fig.9 show that the average normalized throughput under 4 RTT is the best than that of other intervals. Therefore, we use 4 RTT as online tuning interval in this paper.
Overhead. To investigate the overhead of DeepCC and compare it with the off-the-shelf congestion controls, we set up an emulated network with 12 Mbps bottleneck link and 60 ms RTT for 60 seconds and send traffic from the sender to the receiver. We measure the average CPU utilization of these schemes on the sender. On account of Cubic and BBR implemented in the kernel, we evaluate the CPU utilization of iperf for sending their traffic. Results are shown in Fig. 10. It is worth nothing that the overhead of different offline-trained models is not the same due to their different complexity. For example, the overhead of Remy(100x)[4] is 40% higher than that of Remy($\delta = 1$)[5]. As we expected, DeepCC, similar to Indigo, achieves a lower overhead than that of other schemes except for BBR and Cubic. If we implement DeepCC in a more efficient language (such as C or C++ language) instead of Python, or convert DeepCC agent into a decision tree [27] in the Linux kernel, it could be possible to achieve even lower overhead. We leave this as our future work.

D. Coexistence of DeepCC and non-DeepCC flows

To evaluate DeepCC with competing flows, we set up our testbed to experiment with the coexistence of DeepCC or non-DeepCC flows. We use two hosts as the sender and receiver

---
[4]The model with a 100x range of link rates is provided by [19].
[5]The model which δ represents the relative importance of latency is provided by [4].
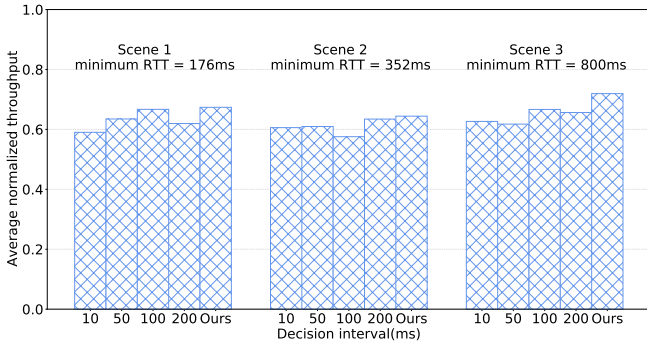
Fig. 8. The average normalized throughput across different decision intervals.
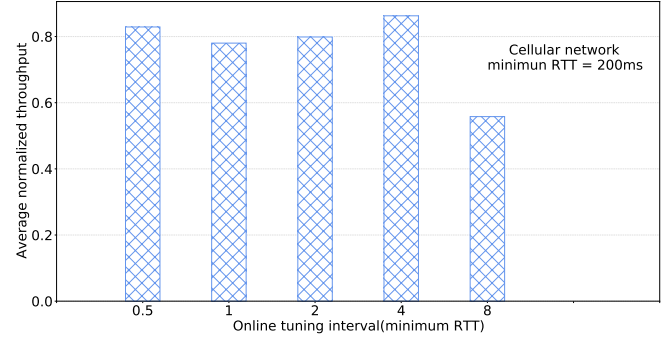

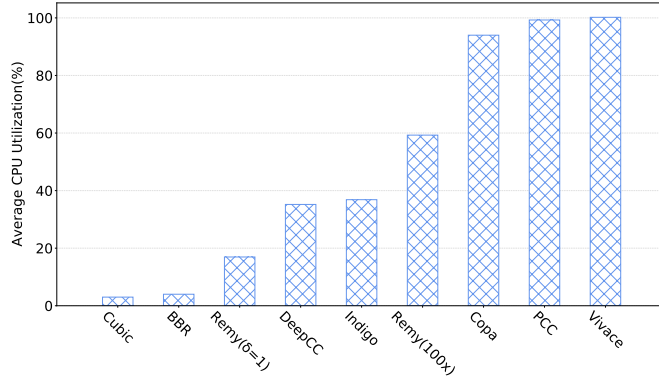Fig. 9. The average normalized throughput across different tuning intervals.

Fig. 10. Overhead of DeepCC compared with other schemes.

respectively. They are connected through a router running OpenWrt [28]. We use tc [29] in OpenWrt to regulate the bottleneck link.

Coexistence with non-DeepCC flows. To evaluate the DeepCC's friendliness, we examine different target flows of DeepCC competing with Cubic flows on our testbed. We choose Cubic flow as the reference flow rests on the fact that Cubic is the default deployment in Linux kernel. In our experiments, we start simultaneously two flows from the sender to the receiver using different schemes including Cubic, BBR, Remy, Indigo, PCC, Vivace, and DeepCC.

As shown in Fig. 11, we report the average throughput achieved by each scheme and Cubic throughout time. The results indicate that BBR, Vivace and Copa are aggressive and get nearly all the bandwidth from the Cubic flow. However, when completing with Remy, Indigo and PCC, Cubic is aggressive, while PCC's share of bottleneck link's bandwidth changes from the high to low and does not grow in the presence of Cubic.

In the first few seconds, Cubic quickly grabs the bandwidth. When the queue is full and packet loss occurs due to congestion, Cubic reduces its cwnds. However, at this time, BBR is not sensitive to packet loss and still takes two times of BDP as its cwnds. Therefore, BBR flow can fully leverage the queue while the Cubic flow can not share fairly the bandwidth due to its sensitivity to packets loss. PCC, Vivace and Copa set their sending rate based on their predefined utility function. They require good start point and need time to find the

good sending rates when competing with the Cubic flow. In other cases, Remy and Indigo are delay-sensitive congestion control schemes. They focus on the delay and have lower rate while Cubic can achieve high bandwidth. In above cases, the friendliness between these schemes and Cubic is not desirable.

In Fig.11, Cubic flow can share fairly the bandwidth with Cubic flow. However, DeepCC (the target is set with 5 Mbps, 180 ms, 1%) mainly uses the available bandwidth outside of the cubic flow, and achieves a proportional fairness with the cubic flow. When the sending rate of Cubic decreases, the available bandwidth grows and DeepCC increases the sending rate accordingly to obtain higher bandwidth. When the sending rate of Cubic increases, the available bandwidth reduces and DeepCC also reduces the sending rate. Although the obtained bandwidth of DeepCC fluctuates, in a long run it achieves similar average throughput with that of Cubic.

Furthermore, we use DeepCC with different targets to investigate its completing behaviors. Fig. 12 shows that the strong preemption of DeepCC in high-throughput mode benefits from the ability of utilizing the capacity of the bottleneck. It has a higher throughput than that of Cubic. When the target is in the low-latency mode, DeepCC has a weaker preemption of available bandwidth than that of Cubic. When the target is set in the middle, DeepCC achieves a better friendliness on average throughput with competing Cubic flows. As expected, DeepCC with different targets achieves varying proportional share of the bandwidth.

Coexistence of DeepCC flows with the same target. To understand the behavior of DeepCC facing other DeepCC flows, we set up two competing DeepCC flows with the same target, e.g., high-throughput target. The data transmission of the two flows initiates sequentially with a 15s interval and each flow transmits continuously for 80s. Fig. 13 depicts the fairness property between two DeepCC flows with the same target. As expected, DeepCC can achieve an acceptable fairness with other DeepCC flows. If there are two long flows competing with the bandwidth, DeepCC can achieve a good fairness characteristic through a period of adjustment.

E. Robustness

In the following, we evaluate the robustness and convergence of DeepCC. The emulated network link is set as a steady link with 12 Mbps, 60 ms RTT, and 90 KB of buffer.
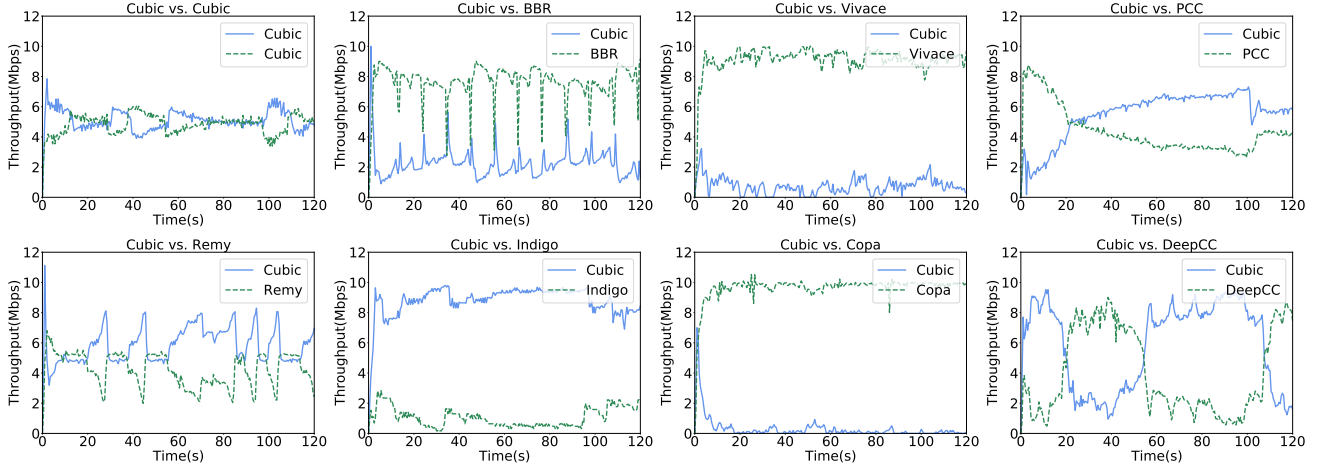
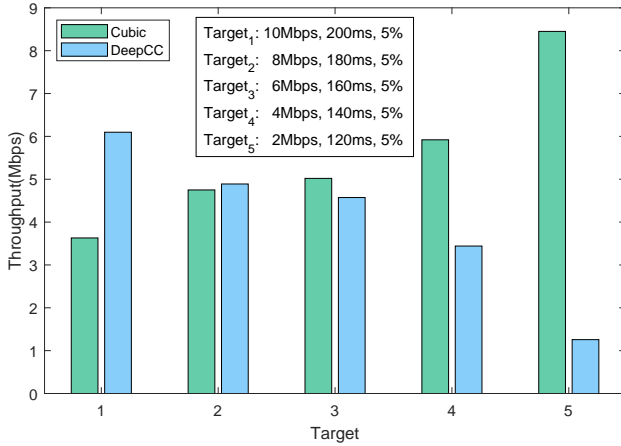Fig. 11. Throughput dynamics of different congestion controls competing the bottleneck link with Cubic.

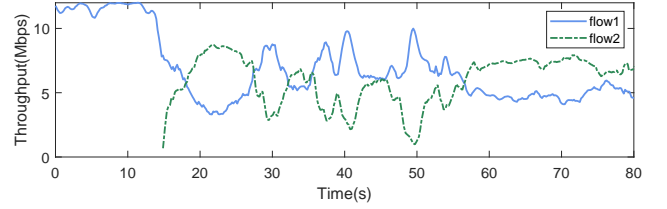Fig. 12. Throughput of different DeepCCs vs. Cubic.

Fig. 13. Dynamic behavior of competing DeepCC flows with the same target.

If the observed loss rate is higher than that of target and the other two dimensions of measurement are satisfied with the target, DeepCC does not update the gradient according to the constraint of loss rate gradient in the online tuning algorithm. **Convergence time of learning-based schemes.** Here the convergence time refers to the time for the learning-based algorithms to reach a stable state. Although online schemes could react timely to the variable network conditions, the performance may be greatly impacted by convergence time. To evaluate DeepCC, we set the *target* value T as 12 Mbps throughput, 100 ms maximum delay, and 1% loss rate. Fig. 15 illustrates the convergence process of several congestion control schemes with 0.5s granularity. As the results showed, DeepCC has similar stable convergence behavior as the offline algorithms, i.e., Indigo and Remy (ver. δ=1.0), but the throughput of DeepCC is higher than that of them. Compared with online learning schemes, DeepCC has the same ability to quickly lift throughput to 12 Mbps as Vivace (ver. latency), but the throughput of DeepCC oscillates less than that of it. In contrast, PCC takes the longest time to reach the throughput ceiling. Benefiting from offline and online learning, DeepCC gains a good start point from offline training and quickly adapts to network conditions through online fine-tuning.

Robustness to packet loss. When facing the lossy network condition, the loss-based and delay-based schemes often perform poorly, which is often shown as a sharp decline in throughput. On the contrary, the learning-based schemes can combat more non-congested packet loss since they do not take packet loss or delay as the explicit congestion signal.

To evaluate the robustness to packet loss of DeepCC, we set up a steady link with a stochastic loss rate ranging from 0% to 6%. The target for DeepCC is set as 11 Mbps throughput, 100 ms delay, and 5% loss rate. The target loss rate represents the maximum tolerable link loss rate. As shown in Fig. 14, Remy is insensitive to packets loss, whereas it only achieves low throughput. PCC maintains throughput until the packet loss rate reaches 5%, i.e., the predefined loss tolerance included in its objective function. The throughput of Vivace gradually decreases as the stochastic loss rate increases.

DeepCC remains insensitive to random packet loss and obtains consistent high throughput. This is because the goal (more precisely $g^{(3)}$) is not updated by the online tuning algorithm when the observed loss rate is less than that of target. Thus DeepCC ignores the random loss and runs to pursue the other two dimensions of the target (i.e., throughput and delay).

VII. DISCUSSION

DeepCC is a flexible congestion control generated by multi-objective learning and dynamically tuning policy, i.e., optimization objective for different network conditions and application requirements. Although the performance of DeepCC achieved is encouraging, some issues remain to be solved.
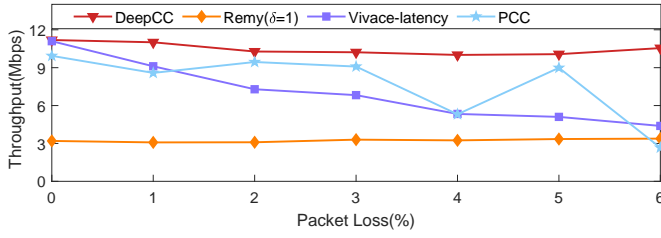
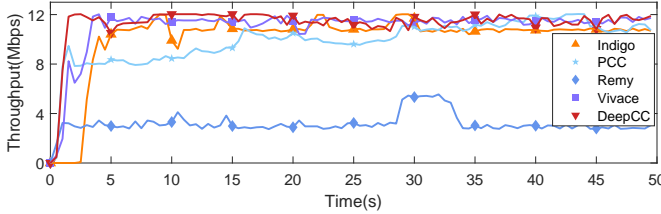Fig. 14. Robustness to stochastic packet loss.

Fig. 15. Convergence time of different learning-based schemes.

(1) Multi-objective representation. In this paper, DeepCC adopts the form of weighted objective functions about three metrics, i.e., throughput, delay, and packet loss, as its offline optimization objective. If applications also concern other performance metrics, such as jitter, the objective function of DeepCC can be easily extended to support it. Further, we leave the cases that the optimization objective involves more complex forms or constraints for our future work.

(2) Overhead of large-scale deployment. DeepCC can be deployed on the server-side and maintain the client unchanged. The control decision of DeepCC is obtained from the neural networks at the server. When multiple concurrent connections are established at the server, the memory and CPU consumption will increase, and thus influence the model inference time. The server can mitigate this impact by 1) running the model inference as a service (e.g., tensorflow serving [30]) with multiple instances, 2) performing the load balancing, and 3) limiting the maximum number of concurrent connections.

(3) Coexistence of DeepCC flows with different targets. A potential concern for DeepCC is how high-throughput flows affect the low-latency flows. We experiment with our testbed using a bottleneck link of 10 KB of buffer [31]. The results show that the performance of these flows achieves a similar throughput and delay respectively. Specifically, the queueing delay is close to zero. Due to the shallow-buffered network, the controllable space of DeepCC is narrow. Under this circumstance, DeepCC would tune to a similar policy even facing different targets, which degenerates from the multi-objective to the single-objective optimization. However, these results can not fully reflect the performance of DeepCC in the real networks since the last mile is typically the speed bottleneck in communication networks [32]. Therefore, DeepCC can benefit from different bottlenecks to satisfy different targets in §VI-B.

VIII. RELATED WORK

Congestion control has been continuously studied since the advent of computer networks. As conventional schemes, such as Cubic [5] and Vegas [33], were designed for general purposes with a "best effort" mentality, they show disadvantages in satisfying modern applications that pose strict requirements on networking. Despite some efforts in [10] are made to realize explicit control for buffer delay, it only works well in the cellular network and cannot extend to other network scenarios or metrics e.g., throughput or loss rate.

Encouraged by the successful experience of machine learning in other fields [13], [34], [35], network researchers turn to learning-based approaches. Rather than leveraging the control rules designed by humans, they propose to use objective-based approaches to guide the decision-making process in a network environment. Many solutions are raised in this way. For example, Remy [4] and Indigo [17] perform optimization by learning CC rules offline. Once being trained, neither Remy nor Indigo can be adjusted to satisfy the requirements without retraining. PCC [12] and Vivace [16] depend on online learning to make right decisions. Owing to online learning, PCC and Vivace are able to provide no-regret guarantees whereas they do not customize and adapt to the requirements. Orca [36] designs two levels of control which combines Cubic and a agent. The agent learns the factor of cwnds for Cubic based on DRL algorithm. Though Orca can achieve high performance in some scenarios, it can not guarantee the application-specified demands.

IX. CONCLUSION

To bridge the gap between application requirements and congestion control, we propose DeepCC, a congestion control that combines the ideas from both offline learning and online tuning. DeepCC leverages a novel multi-objective DRL to learn the multi-objective control policy offline and automatically achieves desired outcomes with the gradient-based online tuning method. The experiment results show that our approach not only achieves a wide range of performance trade-offs but also works well for untrained network scenarios.

ACKNOWLEDGMENT

REFERENCES

[1] K. Miller, A. K. Altamimi, and A. Wolisz, "Qoe-based low-delay live streaming using throughput predictions," *ACM Transactions on Multimedia Computing Communications & Applications*, vol. 13, no. 1, p. 4, 2016.

[2] 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.

[3] B. Jansen, T. Goodwin, V. Gupta, F. Kuipers, and G. Zussman, "Performance evaluation of webrtc-based video conferencing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 45, pp. 56–68, 2018.

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

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

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

[7] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Usenix Conference on Networked Systems Design and Implementation(NSDI)*, 2013, pp. 459–472.

[8] 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.

[9] Q. Xu, S. Mehrotra, Z. Mao, and J. Li, "Proteus: Network performance forecast for real-time, interactive mobile applications," in *Proceeding of the International Conference on Mobile Systems, Applications, and Services*, 2013, pp. 347–360.

[10] 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.

[11] W. K. Leong, Y. Xu, B. Leong, and Z. Wang, "Mitigating egregious ack delays in cellular data networks by eliminating tcp ack clocking," in *IEEE International Conference on Network Protocols(ICNP)*, 2013, pp. 1–10.

[12] 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.

[13] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, d. D. G. Van, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot, "Mastering the game of go with deep neural networks and tree search." *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[14] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805v2*, 2018.

[15] 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.

[16] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "PCC vivace: Online-learning congestion control," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA, 2018, pp. 343–356.

[17] 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.

[18] Y. Jin and B. Sendhoff, "Pareto-based multiobjective machine learning: An overview and case studies," *IEEE Transactions on Systems Man & Cybernetics Part C*, vol. 38, no. 3, pp. 397–415, 2008.

[19] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, "An experimental study of the learnability of congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 479–490.

[20] M. Schapira and K. Winstein, "Congestion-control throwdown," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 2017, pp. 122–128.

[21] 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.

[22] A. Dosovitskiy and V. Koltum, "Learning to act by predicting the future," *ICLR*, 2017.

[23] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," *ICML*, 2015.

[24] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Computer Science*, 2014.

[25] "Tensorflow," https://www.tensorflow.org.

[26] "Amazon cloud," https://aws.amazon.com/.

[27] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu, "Interpreting deep learning-based networking systems," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, New York, NY, USA, 2020, p. 154–171.

[28] "Openwrt." https://openwrt.org/.

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

[30] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017.

[31] N. McKeown, G. Appenzeller, and I. Keslassy, "Sizing router buffers (redux)," *Computer Communication Review*, vol. 49, pp. 69–74, 2019.

[32] S. Sundaresan, N. Feamster, and R. Teixeira, "Home network or access link? locating last-mile downstream throughput bottlenecks," Jan. 2016, pp. 111–123, 17th International Conference on Passive and Active Measurement, PAM 2016.

[33] S. W. O'Malley, L. S. Brakmo, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," *SIGCOMM*, vol. 24, no. 4, pp. 24–35, 1994.

[34] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *ACM Workshop on Hot Topics in Networks*, 2016, pp. 50–56.

[35] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," *Conference of the ACM SIGCOMM*, 2017.

[36] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 632–647.