Experience-driven Congestion Control: When Multi-Path TCP Meets Deep Reinforcement Learning

Zhiyuan Xu, *Student Member, IEEE,* Jian Tang, *Fellow, IEEE,* Chengxiang Yin, *Student Member, IEEE,* Yanzhi Wang, *Member, IEEE,* and Guoliang Xue, *Fellow, IEEE,*

***Abstract*—In this paper, we aim to study networking problems from a whole new perspective by leveraging emerging deep learning, to develop an experience-driven approach, which enables a network or a protocol to learn the best way to control itself from its own experience (e.g, runtime statistics data), just as a human learns a skill. We present design, implementation and evaluation of a Deep Reinforcement Learning (DRL) based control framework, DRL-CC (DRL for Congestion Control), which realizes our experience-driven design philosophy on Multi-Path TCP (MPTCP) congestion control. DRL-CC utilizes a single (instead of multiple independent) agent to dynamically and jointly perform congestion control for all active MPTCP flows on an end host with the objective of maximizing the overall utility. The novelty of our design is to utilize a flexible Recurrent Neural Network (RNN), LSTM, under a DRL framework for learning a representation for all active flows and dealing with their dynamics. Moreover, we, for the first time, integrate the above LSTM-based representation network into an actor-critic framework for continuous (congestion) control, which leverages the emerging deterministic policy gradient to train critic, actor and LSTM networks in an end-to-end manner. We implemented DRL-CC based on the MPTCP implementation in the Linux kernel. The experimental results show that 1) DRL-CC consistently and significantly outperforms a few well-known MPTCP congestion control algorithms in terms of goodput without sacrificing fairness; 2) it is flexible and robust to highly-dynamic network environments with time-varying flows; and 3) it is friendly to regular TCP.**

***Index Terms*—AI, Deep Learning, Experience-driven Control, Congestion Control, TCP, Multi-Path TCP**

I. Introduction

Many algorithms and protocols have been proposed to operate computer and communication networks and utilize its resources efficiently and effectively. Traditional methods for network control and resource allocation can be divided into two categories: *state-oblivious* and *optimization-based*. A state-oblivious method usually follows a pre-defined (fixed) policy for control and resource allocation. Typical examples include shortest-path routing that uses the hop-count as the routing metric and load-balancing routing (e.g., VLB [30]) that always splits traffic load evenly over all the candidate paths. An optimization-based method usually consists of two steps: 1) formulating a resource allocation problem into a mathematical programming problem based on certain mathematical models (such as queueing model [31] and interference model [9]); and 2) designing an algorithm to solve it according to its mathematical properties (such as convex programming). Typical examples include those well-known Network Utility Maximization (NUM) algorithms [19], [25]. We argue that neither of these two approaches will work well for modern or future networks (such as 5G network and Software Defined Network (SDN)), which have become or are expected to be very complicated and highly-dynamic. A state-oblivious method usually leads to a simple algorithm or protocol, which, however, may suffer from very poor performance (such as throughput and delay) in a highly time-variant network due to its lack of careful consideration for runtime states and suboptimal solutions. An optimization-based method, however, needs to have an accurate prediction for future values of some key parameters (such as user demands, link usages, etc) as input; and accurate mathematical models to estimate/characterize network behavior (after applying a given resource allocation solution). Both of them are very challenging, especially in complex networks. Hence, we aim to study networking problems from a whole new perspective by leveraging emerging Artificial Intelligence (AI) techniques, especially deep learning, to develop an experience-driven approach, which enables a network or a protocol to learn the best way to control itself from its own experience (e.g, runtime statistics data), just as a human learns a skill (such as swimming and driving). Unlike state-oblivious or optimization-based methods, the experience-driven approach does not rely on any mathematical model but is expected to make wise decisions on online control with full consideration for real-time runtime states. Even though the term, "data-driven", has been used in some related works, we argue that the term "experience-driven" is more accurate than "data-driven". There are three kinds of data in a communication network: user data (i.e., payload), control data (i.e., data in control messages and packet headers), and runtime statistics. If we use the term "data-driven", it is not clear what kind of data we refer to. As mentioned above, runtime statistics actually represent the past experience of a network. So "experience-driven" basically means that a network learns the best control policy from its runtime statistics collected in the past.

To demonstrate the feasibility and superiority of this design

Zhiyuan Xu, Jian Tang (Corresponding Author), Chengxiang Yin are with Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244, USA (e-mail: zxu105@syr.edu, jtang02@syr.edu, cyin02@syr.edu).

Yanzhi Wang is with Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115, USA (e-mail: yanz.wang@northeastern.edu).

Guoliang Xue is with Ira A. Fulton Schools of Engineering, Arizona State University, Tempe, AZ 85287, USA (e-mail: xue@asu.edu).

This research was supported in part by NSF grants 1704662 and 1704092. The information reported here does not reflect the position of the government.

philosophy, we focus on a fundamental networking problem, congestion control, in this paper. Specifically, we consider congestion control in Multi-Path TCP (MPTCP) [6], which was designed to make use of multiple network interfaces (e.g., Ethernet, WiFi and 4G/LTE) to improve end-to-end bandwidth and robustness, and has already become a widely-used standard protocol. MPTCP allows to split a single TCP flow into multiple sub-flows across multiple paths. It has attracted lots of attention from both industry and academia due to its potential on significant throughput improvements, which are highly desired for some emerging applications that demand high end-to-end bandwidth. Current TCP's congestion control does not perform well on lossy and high Round Trip Time (RTT) links, especially on MPTCP [4]. Moreover, most congestion control algorithms, including those designed particularly for MPTCP, pre-define some packet-level events as response signals, and specify a fixed control policy with one or multiple rules for different cases. For example, halving the congestion window when a packet loss is detected [13]; adjusting the congestion window by a certain amount based on the changing rate of RTTs [2]. Such congestion control algorithms may not work well in a complex and highly-dynamic network, in which many factors (such as random loss, a large range of RTTs, lossy links, rate reshaping at gateways or middleboxes, etc.) may affect its performance since it looks impossible to pre-define the best or even a good rule for each possible case that may occur at runtime. Note that traditional congestion control algorithms can be considered as heuristic algorithms for the corresponding optimization problems, which likely lead to suboptimal (instead of optimal) solutions. Hence, they can be categorized as optimization-based methods described above.

In this paper, we present design, implementation and evaluation of a Deep Reinforcement Learning (DRL) based control framework, DRL-CC (DRL for Congestion Control), which realizes our experience-driven design philosophy on MPTCP congestion control. We choose DRL as the basis for our design because DRL is a very promising technique for network control due to its support for model-free control, and its capability of handling dynamic and sophisticated state spaces, which have been discussed and analyzed in a recent work [43]. DRL-CC utilizes DRL to learn to take best actions according to runtime states without relying on any accurate mathematical model or any pre-defined control policy.

However, designing a DRL-based framework for MPTCP congestion control is not straightforward but quite challenging. First, a straightforward solution is to use a DRL agent to perform congestion control for each MPTCP flow independently. However, this solution may not work well since it lacks necessary and effective cooperations among these agents, while concurrent flows may interfere with each other due to their competition for common resources, which, however, cannot be well addressed by independent agents. DRL-CC leverages a single (instead of multiple independent) agent to dynamically and jointly perform congestion control for all active MPTCP flows on an end host with the objective of maximizing the overall utility (defined by a utility function). Note that according to TCP/MPTCP, congestion control is only done on sending hosts. Hence, here "all active MPTCP flows" only refer to those whose sending host is the one where the DRL agent is running. It is quite challenging to use DRL to dynamically handle multiple flows that may come and go at any time since a DRL agent usually uses a deep feed-forward neural network or a deep Convolutional Neural Network (CNN) as the function approximator for action inference, which has a fixed input size. The novelty of our design is to utilize a Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM [12]), under a DRL framework for learning a representation for all active flows and dealing with their dynamics. Moreover, basic Deep Q-Network (DQN) based DRL (proposed in the seminal work [21]), does not work for the congestion control problem since it is only able to deal with discrete control with a limited action space. We, for the first time, integrate the above LSTM-based representation learning network into an actor-critic framework called DDPG (Deep Deterministic Policy Gradient [18]) for continuous (congestion) control, and leverages the emerging deterministic policy gradient [32] to train critic, actor and LSTM networks in an end-to-end manner.

In addition, a unique and desirable feature of our design is that we use a DRL agent to control all (instead of a single) active MPTCP flows. In this way, we can hopefully obtain a global optimal solution for MPTCP flows; while all the traditional congestion control algorithms perform control for individual MPTCP flows independently, thus can only achieve local optimal or suboptimal solutions. So a traditional algorithm usually behaves conservatively because if it tries to be aggressive and selfish, one of flows may abuse all the available resources and starve other flows. However, our DRL agent has a global view over all the available resources and MPTCP flows. Due to the training and working mechanisms (e.g., back propagation) of the Reinforcement Learning (RL) framework, it is instructed to make full use of all available resources to maximize the reward function (defined in Section III), which is a widely-used utility function and is known to be able to achieve a good tradeoff between goodput and fairness [33].

We implemented DRL-CC based on the MPTCP implementation in the Linux kernel. We conducted extensive real experiments to evaluate its performance. Specifically, we compared DRL-CC with the well-known congestion control algorithms proposed particularly for MPTCP, including LIA [27], BALIA [26], OLIA [17] and wVegas [42], which have all been implemented in the Linux Kernel [24]. Second, we tested our method under different settings and cases, such as different link bandwidths, link delays, packet loss ratios, etc. In addition, we conducted our performance evaluation in terms of goodput, fairness, robustness and TCP-friendliness. The experimental results well justify the effectiveness and superiority of DRL-CC.

To the best of our knowledge, we are the first to address congestion control in MPTCP using emerging DRL. In addition, the end-to-end trainable model integrating LSTM, actor and critic networks is novel and has not been used in the context of DRL. Moreover, it can handle a variable input size. We believe such a design may have a significant impact on future research along this line since it can be applied to many other system control problems with a time-varying input size, e.g., for routing in mobile ad-hoc networks, connection requests may come and go at any time as well.

II. Deep Reinforcement Learning (DRL)

In this section, we give a brief introduction to DRL. Under a regular Reinforcement Learning (RL) framework, an agent interacts with an environment (e.g., system) in discrete decision epochs. At each epoch t, the agent makes an observation of the state $\mathbf{s}_t$ of the environment, takes an action $\mathbf{a}_t$ according to its policy, and receives a reward r_t. The agent aims to find a policy $\pi(\mathbf{s})$ to map its state to a deterministic action or to a probability distribution over actions such that the discounted cumulative reward $R_0 = \sum_{t=0}^{T} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)$ is maximized, where $r(\cdot)$ is the reward function and $\gamma \in [0, 1]$ is the factor that discounts future rewards.

The deep version of RL was introduced in a well-known work [21] by Mnih *et al.* from DeepMind, which extends the traditional Q-learning to bridge the gap between high-dimensional sensory inputs (e.g. raw images) and actions. A unique feature of the DRL agent in [21] is to use a Deep Neural Network (DNN) called DQN as the function approximator. A DQN takes a state-action pair $(\mathbf{s}_t, \mathbf{a}_t)$ as input and outputs the corresponding Q value $Q(\mathbf{s}_t, \mathbf{a}_t)$, which is the expected discounted cumulative reward:

$$Q(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}\Big[R_t | \mathbf{s}_t, \mathbf{a}_t\Big], \tag{1}$$

where $R_t = \sum_{k=t}^{T} \gamma^k r(\mathbf{s}_t, \mathbf{a}_t)$. The action can be derived by applying a commonly-used greedy policy:

$$\pi(\mathbf{s}_t) = \operatorname*{argmax}_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t). \tag{2}$$

According to Q-learning, the training target value for each state-action pair can be derived using the Bellman equation:

$$y_t = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma Q(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1}) | \boldsymbol{\theta}^Q), \tag{3}$$

where $\boldsymbol{\theta}^Q$ is the parameters of the DQN. Based on the target value, the DQN can be trained by minimizing the following loss:

$$L(\boldsymbol{\theta}^Q) = \mathbb{E}\Big[y_t - Q(\mathbf{s}_t, \mathbf{a}_t | \boldsymbol{\theta}^Q)\Big]. \tag{4}$$

Even though neural network or DNN has been used as the function approximator for RL before, it is known that such a non-linear function approximator is not stable and may even lead to divergence. To improve the stability of learning, Mnih *et al.* [21] introduced two effective techniques: experience replay and target network. With experience relay, a DRL agent collects and stores state transition samples into a relay buffer, and then updates the DNN using a mini-batch sampled from the replay buffer instead of the immediately collected transition sample (used in traditional Q-learning). By doing so, the DRL agent could break correlations in the observation sequence, and learn from a more independently and identically distributed past experience, which is required by most of the training algorithms, such as Stochastic Gradient Descent (SGD). They proposed to use a separate target network to estimate target values $< y_t >$, which shares the same network structure as the original DQN. But its parameters are slowly updated every $C > 1$ epochs and are held fixed in between. These two techniques can smooth out the learning processing and avoid oscillations or divergence.

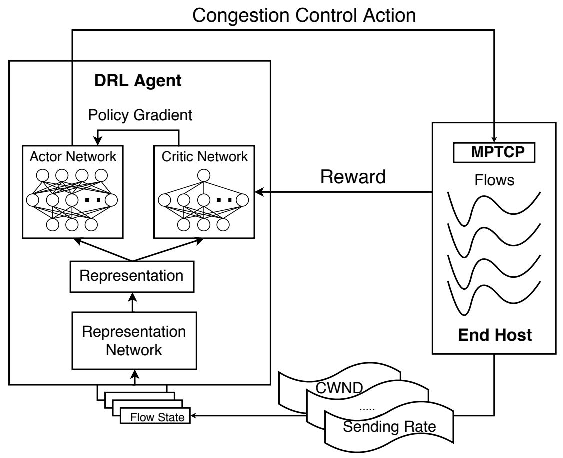


Fig. 1: The architecture of DRL-CC

As mentioned above, DQN-based DRL is restricted to discrete control with a limited action space and there is no trivial extension to continuous control, which, however, is quite common in computer and communication networks (e.g., congestion control). A commonly-used approach to continuous control is policy gradient [35]. In a recent work [18], Lillicrap *et al.* introduced an actor-critic approach called Deep Deterministic Policy Gradient (DDPG) for DRL, which leverages both DNNs and the emerging deterministic policy gradient [32] for continuous control. The key idea behind DDPG is to simultaneously maintain two functions: one is the parameterized actor function $\pi(\mathbf{s}_t | \boldsymbol{\theta}^\pi)$ used for deriving actions; and another is the parameterized critic function $Q(\mathbf{s}_t, \mathbf{a}_t | \boldsymbol{\theta}^Q)$ used for evaluating actions. The critic function is implemented using the DQN mentioned above, which takes a given state-action pair as input and outputs the corresponding Q-value. It can be trained as a regular DQN, which has been introduced above. The actor function can be implemented by another DNN, which takes a state as input and outputs the best action (that could be continuous). As shown in [18], to update the actor network, the chain rule can be applied to the the expected cumulative reward J with respect to the actor parameters $\boldsymbol{\theta}^\pi$:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}^\pi} J &\approx \mathbb{E}\Big[\nabla_{\boldsymbol{\theta}^\pi} Q(\mathbf{s}, \mathbf{a} | \boldsymbol{\theta}^Q)|_{\mathbf{s}=\mathbf{s}_t, \mathbf{a}=\pi(\mathbf{s}_t|\boldsymbol{\theta}^\pi)}\Big] \\ &= \mathbb{E}\Big[\nabla_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a} | \boldsymbol{\theta}^Q)|_{\mathbf{s}=\mathbf{s}_t, \mathbf{a}=\pi(\mathbf{s}_t)} \cdot \nabla_{\boldsymbol{\theta}^\pi} \pi(\mathbf{s}|\boldsymbol{\theta}^\pi)|_{\mathbf{s}=\mathbf{s}_t}\Big]. \end{aligned} \tag{5}$$

Note that both experience replay and target network can also be used together with DDPG to ensure the learning stability.

III. DRL-based Congestion Control for MPTCP

In this section, we present the proposed DRL-based framework for congestion control in MPTCP, *DRL-CC*.

A. Overview

First of all, we give an overview for DRL-CC, which is illustrated in Fig. 1. The key idea behind our design is to utilize a single (rather than multiple independent) DRL agent to perform congestion control for all active MPTCP flows on an end host to maximize the overall utility (defined by a utility function). As mentioned above, "all the active MPTCP flows" only refer to those whose sending host is the one where DRL-CC is running. To realize this idea, we design the architecture of DRL-CC (we will use DRL-CC and DRL-CC agent interchangeably in the following), which consists of the following components:

- Representation Network (Section III-B): It leverages LSTM to learn a *representation* of current states of all active MPTCP and TCP flows in a sequence learning manner.
- Actor-Critic (Section III-C): It trains an actor network and a critic network along with the LSTM-based representation network in an end-to-end manner and derives an action for congestion control of a MPTCP flow based on the learned representation and the state of the target flow.

Next, we describe the state, action and reward of DRL-CC:

STATE: The state of a flow i at epoch t $\mathbf{s}_t^i = [\mathbf{s}_t^{1,1}, \cdots, \mathbf{s}_t^{i,k}, \cdots, \mathbf{s}_t^{N,K_i}]$, and $\mathbf{s}_t = [b_t^{i,k}, g_t^{i,k}, d_t^{i,k}, v_t^{i,k}, w_t^{i,k}]$, where $\mathbf{s}_t^{i,k}$ is the state of subflow k of flow i at t; $b_t^{i,k}$, $g_t^{i,k}$, $d_t^{i,k}$, $v_t^{i,k}$ and $w_t^{i,k}$ are the corresponding sending rate, goodput, average RTT, the mean deviation of RTTs and the congestion window size respectively; and N is the total number of both TCP and MPTCP flows, and K_i is the number subflows of flow i. If flow i is a TCP flow, then $K_i = 1$; and if flow i is a MPTCP flow, then $K_i \geq 1$. Then the state at epoch t, $\mathbf{s}_t = [\mathbf{s}_t^1, \cdots, \mathbf{s}_t^i, \cdots, \mathbf{s}_t^N]$. Here *goodput* can be considered as effective throughput, which only counts those successfully received packets. We select these key parameters into the state because they may have a significant impact on the end-to-end performance and have been considered in the design of some related works [41]. During our testing, we found that adding more parameters into the state does not necessarily result in noticeable performance improvement, which, however, undoubtfully increases data collection overhead. Note that the values of these parameters are all measured during the past epoch $(t-1)$. In order to well address interference and fairness on an end host, we consider all the flows (including both regular TCP and MPTPC) when designing the state space. Certainly, if the flow is a (regular) TCP flow, then there is only one subflow (i.e., $K_i = 1$).

ACTION: An action at epoch t $\mathbf{a}_t = [x_t^1, \cdots, x_t^k, \cdots, x_t^K]$, where x_k specifies how much change needs to be made to the congestion window of subflow k of the target MPTCP flow. The positive, negative and 0 values lead to increasing, reducing and staying at the same congestion window size respectively. Note that at each epoch t, DRL-CC only takes an action on one (target) MPTCP flow.

REWARD: the reward at epoch t, $r_t = \sum_i^N U(i,t)$, where $U(i,t)$ gives the utility of active MPTCP/TCP flow i . Note that the proposed framework is not restricted to any particular utility function. Many different functions (such as throughput, delay, α-fairness [33]) can be used here to calculate the network utility. This reward should be designed according to real needs from upper-layer applications. In our implementation, we chose a widely-used utility function $U(i,t) = \log g_t^i$ [41], where g_t^i is the average goodput of MPTCP flow i during the past epoch. It is known that maximizing this utility function leads to proportional fairness, which is considered to achieve a good tradeoff between goodput and fairness. Moreover, the reward takes into account both TCP and MPTCP flows for the sake of TCP-friendliness.

In short, DRL-CC works as follows. The DRL-CC agent interacts with the end host by collecting the above runtime state information $\mathbf{s}_t$ at each epoch t. The agent is periodically queried by each MPTCP flow and there is only one querying flow at each epoch t (i.e., target flow). At each epoch t, the agent derives an action using the actor and critic networks according to the representation learned by the LSTM-based network and the state of the target flow. Then it deploys the action via the MPTCP implementation (in the OS kernel) to the target flow.

B. Representation Network

The representation network takes as input the states of all active TCP and MPTCP flows (i.e., $\mathbf{s}_t$) at each decision epoch t and generates a representation (i.e., a vector with a smaller size), which is then used by the actor-critic method (Section III-C) for deriving actions. As mentioned above, the main difficulty is to deal with the situation in which the number of flows may change over time. Most DNN (such as a feed-forward neural network) need to have a fixed input size. A straightforward way to use a feed-forward neural network here is to zero-pad the input if the actual number of flows is smaller its input size. We tested this solution via experiments and found that it is ineffective, especially for the cases where the number of flows is much smaller. Similarly, if the number of flows is larger, then we have to exclude some flows, which obviously lead to poor representation learning too. We decide to choose LSTM [12] to serve this purpose, which can have a variable input size (length).

As illustrated in Fig. 2, the states of flows are fed into LSTM one by one (one at each step) and the representation is learned in a sequence learning manner [34] such that the last hidden state $\mathbf{h}_t^N$ is returned as the representation. For simplicity, we denote this representation for epoch t by $\mathbf{h}_t$ (rather than $\mathbf{h}_t^N$). It is worth mentioning that this LSTM-based representation network can be trained together with the actor and critic networks using back propagation in an end-to-end manner, which is discussed in the next section. This is very important since end-to-end training likely leads to better performance than training each part of a model separately.

C. Actor-Critic Method

At each decision epoch t, the representation $\mathbf{h}_t$ (learned by the LSTM-based network described above) is concatenated with the state of the target MPTCP flow and fed into the actor-critic method as input. Then the actor-critic method leverages the actor and critic networks to derive an action, which specifies how to adjust the congestion window size for each subflow of the target MPTCP flow. As mentioned above,

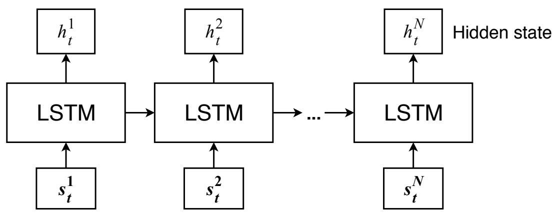


Fig. 2: The representation network

the network utility will be calculated used as a reward signal to optimize the decision policy.

We formally present the DRL-MPTCP framework in Algorithm 1. First the algorithm randomly initialize all parameters $\boldsymbol{\theta^R}$ of representation network $R(\cdot)$; $\boldsymbol{\theta^\pi}$ of actor network $\pi(\cdot)$; and $\boldsymbol{\theta^Q}$ of critic network $Q(\cdot)$. The target networks are used here to improve the learning stability. Target networks $R'(\cdot)$, $\pi'(\cdot)$ and $Q'(\cdot)$ clone the structures of their counterparts, whose parameters are initialized using their counterparts (line 2) and slowly updated using a control parameter τ (line 19). τ is usually set to a very small value such that these target networks are only slightly updated in this step. In our implementation, we set $\tau = 0.001$. This DRL agent will run as a daemon process, waiting for queries from MPTCP flows. So the main body of this algorithm includes a dead loop, where $\mathbf{e}_t$ is the state of the querying (i.e., target) MPTCP flow.

Since all the parameters of the DNNs are randomly initialized, in the early stage of training, the DRL agent cannot totally rely on the action derived from the actor network. An inexperienced DRL agent needs to explore sufficiently with random transition samples to gain necessary good and bad experience, and eventually learns a good (hopefully the best) control policy. Similar as in [18], we apply an Ornstein-Uhlenbeck process to add some random noise to a derived action for efficient and effective exploration in this continuous control task.

The representation of all active flows $\mathbf{h}_t$ is derived from the representation network $R(\cdot)$ (line 6), and the action for the target MPTCP flow is derived from the actor network $\pi(\cdot)$ (line 7). Experience relay has also been utilized here to improve learning stability. Transition samples are first stored into a replay buffer **B** (line 10), and then randomly sampled to a mini-batch of H samples (line 11) for training the representation, critic and actor networks. As introduced above, the critic network is basically a DQN. Hence, the parameters of the critic network $\boldsymbol{\theta}^Q$ are updated by minimizing the commonly-used squared error loss (line 14), where the target value y_i is evaluated by applying the Bellman equation (line 13). The parameters of the representation and actor networks $\boldsymbol{\theta}^R$ and $\boldsymbol{\theta}^\pi$ are updated together with the sampled (i.e., an average over the H samples) policy gradients using the chain rule defined in Equation (5) (lines 15–18). From this training process, we can see that the proposed neural network model (including the representation, critic and actor networks) is end-to-end trainable.

In our implementation, the representation network is a single-layer LSTM unit. The actor network is a fully-connected feed-forward neural network with 2 hidden layers, which includes 128 neurons in both layers. The Rectified Linear function is used for activation in hidden layers and the hyperbolic tangent function is used for activation in the output layer. The critic network has the same structure as the actor network except the output layer, which has only one linear neuron. In our implementation, the actor and critic networks are trained by the Adam optimizer [15], whose learning rates are set to 0.0001 and 0.001 respectively. The discount factor is set to $\gamma = 0.90$. To simplify the neural network implementation, we leveraged TFLearn [37], which provides a higher-level API to TensorFlow, to construct the above three neural networks.

Algorithm 1: DRL-CC

```
1: Randomly initialize representation network $R(\cdot)$, actor
   network $\pi(\cdot)$ and critic network $Q(\cdot)$, with parameters
   $\boldsymbol{\theta^R}$, $\boldsymbol{\theta^\pi}$ and $\boldsymbol{\theta^Q}$ respectively;
2: Initialize target networks $R'(\cdot)$, $\pi'(\cdot)$ and $Q'(\cdot)$ with
   parameters $\boldsymbol{\theta^{R'}} := \boldsymbol{\theta^R}$, $\boldsymbol{\theta^{\pi'}} := \boldsymbol{\theta^\pi}$, $\boldsymbol{\theta^{Q'}} := \boldsymbol{\theta^Q}$;
3: Initialize replay buffer **B**;
4: Initialize Ornstein-Uhlenbeck process $\mathcal{O}$ for exploration;
5: **while** (TRUE) **do**
6:     Derive hidden state $\mathbf{h}_t$ from the representation
       network $R(\mathbf{s}_t)$;
7:     Derive an action $\overline{\mathbf{a}_t}$ from the actor network $\pi(\mathbf{e}_t, \mathbf{h}_t)$;
8:     Apply the random process $\mathcal{O}$ to generate an action $\mathbf{a}_t$
       based on $\overline{\mathbf{a}_t}$;
9:     Execute action $\mathbf{a}_t$ and observe the reward $r_t$;
10:    Store transition sample $(\mathbf{s}_t, \mathbf{e}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1}, \mathbf{e}_{t+1})$ into
       replay buffer **B**;
       /**Training the three networks**/
11:    Sample H transitions $(\mathbf{s}_j, \mathbf{e}_j, \mathbf{a}_j, r_j, \mathbf{s}_{j+1}, \mathbf{e}_{j+1})$ from **B**;
12:    Obtain representation $\mathbf{h}_{j+1}$ from $R'(\mathbf{s}_{j+1})$;
13:    Compute target value for the critic network $Q(\cdot)$:
       $y_j := r_j + \gamma \cdot Q'(\mathbf{e}_{j+1}, \mathbf{h}_{j+1}, \pi'(\mathbf{e}_{j+1}, \mathbf{h}_{j+1}))$;
14:    Update the parameters of the critic network by
       minimizing the loss: $\frac{1}{H}\sum_{j=1}^{H}(y_j - Q(\mathbf{e}_j, \mathbf{h}_j, \mathbf{a}_j))^2$;
15:    Compute the policy gradient from the critic network:
       $\nabla_{\mathbf{a}} Q(\mathbf{e}, \mathbf{h}, \mathbf{a})|_{\mathbf{a}=\pi(\mathbf{e}_j, \mathbf{h}_j), \mathbf{h}=R(\mathbf{s}_j), \mathbf{e}=\mathbf{e}_j}$;
16:    Update the parameters of the actor network using the
       sampled policy gradients:
       $\frac{1}{H}\sum_{j=1}^{H} \nabla_{\mathbf{a}} Q(\mathbf{e}, \mathbf{h}, \mathbf{a}) \cdot \nabla_{\boldsymbol{\theta}^\pi} \pi(\mathbf{e}, \mathbf{h})|_{\mathbf{e}=\mathbf{e}_j, \mathbf{h}=R(\mathbf{s}_j)}$;
17:    Compute the policy gradient from the actor network:
       $\nabla_{\mathbf{h}} \pi(\mathbf{e}, \mathbf{h})|_{\mathbf{e}=\mathbf{e}_j, \mathbf{h}=R(\mathbf{s}_j)}$;
18:    Update the parameters of the representation network
       using the sampled policy gradient:
       $\frac{1}{H}\sum_{j=1}^{H} \nabla_{\mathbf{a}} Q(\mathbf{e}, \mathbf{h}, \mathbf{a}) \cdot \nabla_{\mathbf{h}} \pi(\mathbf{e}, \mathbf{h}) \cdot \nabla_{\boldsymbol{\theta}^R} R(\mathbf{s})|_{\mathbf{s}=\mathbf{s}_j}$;
       /**Updating the target networks**/
19:    Update the parameters of the corresponding target
       networks:
       $\boldsymbol{\theta}^{R'} := \tau\boldsymbol{\theta}^R + (1-\tau)\boldsymbol{\theta}^{R'}$;
       $\boldsymbol{\theta}^{Q'} := \tau\boldsymbol{\theta}^Q + (1-\tau)\boldsymbol{\theta}^{Q'}$;
       $\boldsymbol{\theta}^{\pi'} := \tau\boldsymbol{\theta}^\pi + (1-\tau)\boldsymbol{\theta}^{\pi'}$;
20: **end while**
```

D. Implementation of DRL-CC

We implemented the DRL-CC framework on Ubuntu 16.04. We chose to use the MPTCP v0.92 [24], which is a Linux kernel implementation of MPTCP and was built based on the Linux Kernel long-term support release v4.4.x. The available resource of a kernel program is strictly limited: even the floating point calculation is not allowed in the kernel. A DRL agent, however, may need to do lots of complex mathematical calculations (e,g, computing the gradients) for both forward passes and back propagations in the DNN training and inference. Thus, it is impossible to run the DRL agent in the kernel. We implemented the proposed DRL agent as a user-space process using Tensorflow [1]. The DRL agent runs as a daemon process, which is always kept active and waits for the MPTCP flow queries. Every flow reserves a memory space in the kernel for their subflows, and every subflow can fetch their congestion window size from its memory space. Whenever a MPTCP flow queries, the DRL agent derives an action and deploys it by updating the corresponding congestion window size for each subflow through the MPTCP implementation.

In order to be compatible to current MPTCP implementation, we implemented the proposed DRL-CC agent as a pluggable program following the Linux's specification for congestion control. First, we specified the congestion handler interface *tcp_congestion_ops*, which is a structure of function call pointers. Then we implemented a callback function *mptcp_drl_cong_avoid*, which will be called by each subflow every time an acknowledgment packet is received. Using this function, subflows can keep observing and updating their congestion window sizes.

In addition, before the online-testing, we trained the DRL-CC agent for over $50,000$ epochs (i.e., $50,000$ transition samples) in an offline manner, using iPerf3 [14] to continuously generate packets to keep the network always busy in the test environment, which produced sufficient transition samples for training. Due to different link delays, packet loss rates and bottleneck bandwidth settings, the offline training time varies from an hour to several hours. For example, in the setting of the bandwidth $b_1 = b_2 =$ 8Mbps, the delay $d_1 = d_2 =$ 200ms and the packet loss rate $p_1 = p_2 = 0.5\%$, it took 2.5 hours to complete the offline training process. Once it was taken online, it immediately became ready for use without any setup latency. Note that offline training only needs to be done once and no additional offline training is needed if the agent is rebooted. As mentioned above, even though we used DNNs for inference in our implementation, each of which, however, has only 2 hidden layers. According to our testing, the online inference time is really short, about 0.5ms, which causes negligible overhead for online decision making. Just as many other RL agents, re-training needs to be performed for DRL-CC when the network environment changes (e.g., from a low-bandwidth and high-delay network to a high-bandwidth and low-delay network). This is because sufficient transition samples need to be collected to update the DNN of the agent such that it can gain enough experience for the new environment to make good decisions when similar situations occur. However, what is the the best way to re-train a trained agent for a new environment is a fairly big research topic and is out of the scope of this paper, which will be studied in our future work.

IV. Performance Evaluation

We conducted a comprehensive empirical study for performance evaluation under various test scenarios. In this section, we describe the settings of our test environment, test scenarios, and then present and analyze the corresponding results.

A. Common Experimental Setup

We compared DRL-CC with a few baselines, including LIA [27], BALIA [26], OLIA [17] and wVegas [42], which are all well-known congestion control algorithms proposed particularly for MPTCP. We used their implementation in MPTCP v0.92 [24] for our experiments.

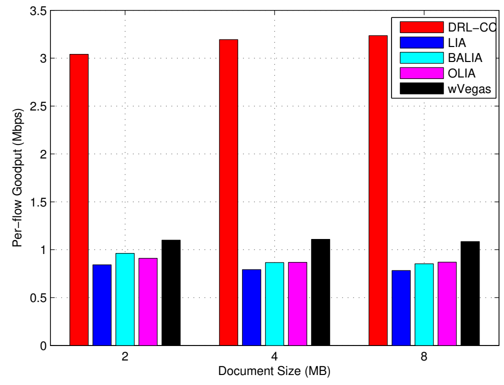
We set up a test environment in our lab for our experiments. The test environment consists of 2 laptops as client and server separately, both running Ubuntu Linux 16.04LTS. Due to the light weight of our design, there is no need for any special device (such as GPU) for training. We found that we could easily run and train the DRL-CC agent on a regular laptop, which has an Intel i7-3630QM CPU and 4GB memory. Two nodes are connected with a Gigabit switch. The server and client have two and one Gigabit Ethernet interfaces respectively, which created two different communication links (i.e., single-link paths) for our testing.

In the test environment, each MPTCP flow includes two subflows, which is the most common setting for MPTCP in practise and has also been used for testing in related works [4], [26]. Similar as in [4], we controlled some key parameters of the communication links in the test environment, such as delay, bandwidth and loss rate using netem [23], which can emulate the communication properties of a wide area network for testing network protocols. We considered a wide range of settings in our experiments: the link delay was set to range from 50ms to 400ms, the packet loss rate was set to range from 0.5% to 4%, and the bottleneck bandwidth varied from 2Mbps to 16Mbps. In our experiments, all the data packets were captured by tcpdump [38] and analyzed by wireshark [40].

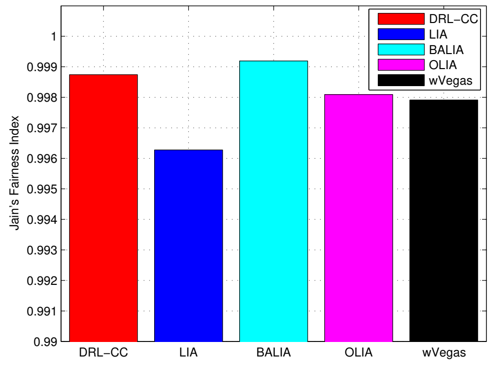
B. Test Scenarios and Experimental Results

We introduce our test scenarios and present the corresponding experimental results. In the first four test scenarios, we evaluated the performance of DRL-CC in a relatively steady environment. Specifically, 5 MPTCP flows (each with 2 subflows) were established between the server and the client and kept active through each experiment. The MPTCP data traffic was generated by retrieving a binary document from a simple HTTP server. The document size ranged from 2MB to 8MB. The goodput was calculated by diving the document size by the elapsed download time. Each number presented in the following figures is the *average goodput per MPTCP flow*.

Scenario 1: In this scenario, we show how the document size affects the goodput. We set the bandwidth $b_1 = b_2 =$ 8Mbps, the delay $d_1 = d_2 =$ 100ms and the packet loss rate $p_1 = p_2 = 2\%$. We used the documents with different sizes: 2M, 4M and 8M. The corresponding results



(a) Average per-flow goodput VS. document size



(b) Jain's fairness index

Fig. 3: Scenario 1: $b_1 = b_2 = 8$Mbps, $d_1 = d_2 = 100$ms and $p_1 = p_2 = 2\%$

are presented in Fig. 3. First, we can see that DRL-CC significantly outperforms all the other methods in terms of goodput. For example, when the document size is 8M, DRL-CC outperforms LIA, BALIA, OLIA, wVegas by 313%, 279%, 272%, 198% respectively. Moreover, since there are two links (paths) between the server and the client and each of them has a bandwidth of 8Mbps, the total end-to-end bandwidth is 16Mbps. There are 5 MPTCP flows and each of them obtains an average goodput of 3.2Mbps (if DRL-CC is used), which means that DRL-CC makes full use of all available bandwidth. In addition, we show the Jain's fairness index (calculated over all MPTCP flows) given by each algorithm in Fig. 3b. We can see that all the algorithms achieve very good fairness since the corresponding indices are all close to 1. Hence, compared to the baselines, DRL-CC leads to much higher goodput without sacrificing fairness. This is mainly due to the way how we define the reward (Section III-A), particularly the utility function, which usually leads to a good tradeoff between goodput and fairness.

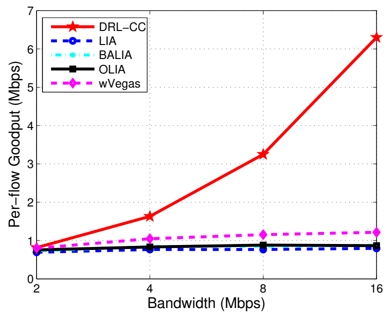
Since all the methods have a similar behavior with different document sizes, and in order to have a relatively long testing time, we used the 8M document in the following scenarios. We present the results corresponding to the next three scenarios in Fig. 4. In addition, we found all the algorithms led to similarly good fairness in the other scenarios. Due to space limitation, we omit the corresponding results and figures.

Scenario 2: In this scenario, we fixed the delay $d_1 = d_2 = 100$ms and the packet loss rate $p_1 = p_2 = 2\%$, we aimed to show the performance of all these methods with different bandwidths by setting the bandwidth b_1/b_2 to 2M, 4M, 8M and 16M in different experiments respectively. The corresponding results are presented in Fig. 4a. We can see when the bandwidth is small (i.e. 2Mbps), the goodputs of all the methods are fairly low and almost the same. When the bandwidth is increased, DRL-CC leads to sharp improvements on goodput, which are much more significant than those given by the other methods. Particularly, when the bandwith is 8Mbps, DRL-CC leads to $325\%, 280\%, 269\%$ and 181% improvements over the baselines respectively.

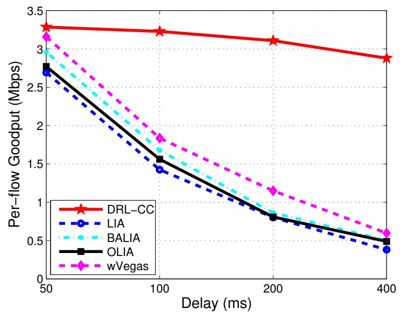
Scenario 3: This scenario was designed to show how the goodputs given by these methods vary with the delay. We set the bandwidth $b_1 = b_2 = 8$Mbps and the packet loss rate $p_1 = p_2 = 0.5\%$, the delay d_1/d_2 was changed from 20ms all the way to 400ms. The results are shown in Fig. 4b. Similar as in the last scenario, when the delay is small (i.e. 50ms), the goodputs of all the methods are fairly high and close. When the delay is increased, the goodputs given by all the methods drop as expected. However, DRL-CC only experiences pretty minor degradation on goodput; while the drops of the other methods are much more substantial. Particularly, when the delay is 400ms, DRL-CC offers $656\%, 473\%, 489\%$ and 382% improvements over the baselines respectively.

Scenario 4: We designed this scenario to see how the packet loss rate affects the goodputs given by all the methods. We set the bandwidth $b_1 = b_2 = 16$Mbps and the delay $d_1 = d_2 = 50$ms. The packet loss rate was set to 0.5%, 1%, 2% and 4% in different experiments respectively. The corresponding results are shown in Fig. 4c. Similar as in Scenario 2, when the packet loss rate is increased, DRL-CC maintains fairly statable performance without a sharp degradation on goodput. However, the goodputs given by all the baselines drop dramatically with the packet loss rate. Particularly, in the case of lossy links with a loss rate of 4%, DRC-CC outperforms these baselines by $468\%, 456\%, 409\%$ and 319% respectively.

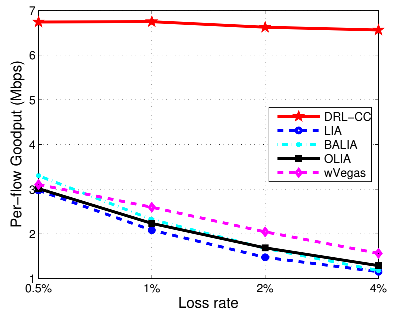
In summary, first of all, this set of scenarios and experiments well justify the superiority of DRL-CC on goodput. Particularly, we observe that DRL-CC significantly outperform the baselines in those cases with high bandwidth, long delay and high packet loss rate. Through good training, DRL-CC can find that making better use of available bandwidth leads to much higher goodput. So it always tries to increase congestion window sizes quickly and aggressively when detecting more available bandwidth. However, the other methods behave much more conservatively in this case since they don't have a mechanism that can explicitly and quickly utilize available bandwidth. In addition, DRL-CC is more suitable for tough network environments (e.g., lossy wireless networks) with a high delay or packet loss rate. As mentioned before, most existing methods follow pre-defined policies to control congestion windows, which are usually too conservative, i.e, reducing or significantly reducing window sizes once detecting long RTTs or packet losses but



(a) Scenario 2: Average per-flow goodput VS. bandwidth with $d_1 = d_2 = 100$ms and $p_1 = p_2 = 2\%$



(b) Scenario 3: Average per-flow goodput VS. delay with $b_1 = b_2 = 8$Mbps and $p_1 = p_2 = 0.5\%$



(c) Scenario 4: Average per-flow goodput VS. loss rate with $b_1 = b_2 = 16$Mbps and $d_1 = d_2 = 50$ms

Fig. 4: Performance of all the methods over different settings

opening congestion windows back up slowly. This certainly leads to low goodput. However, in these cases, DRL-CC usually makes a few attempts and quickly figures out the best ways to set up window sizes without being too conservative or aggressive. Therefore, we can observe DRL-CC brings much more improvements in these tough cases. Last but not the least, as mentioned above, DRL-CC features a joint congestion control over all active MPTCP flows, which is expected to deliver superior performance over those baselines that perform congestion control for flows independently.

Next we introduce two scenarios, in which we tested DRL-CC in a more dynamic and complicated environment. For example, we changed the number of MPTCP flows or even the number of subflows over time. Note that these situations may occur in practice, e.g. a user may open and close a website frequently over time, which leads to establishments and terminations of multiple MPTCP flows. The number of subflows of a MPTCP may also change due to the network state fluctuations, e.g. stepping away from a WiFi hotspot may cause the loss of the corresponding link on a mobile phone. Note that those baselines are not supposed to have any problem dealing with such dynamics since they all manage individual flows separately.

Most settings in Scenarios 5 and 6 are the same as the last few scenarios. We used those key parameters as follows: the bandwidth $b_1 = b_2 = 8$Mbps, the delay $d_1 = d_2 = 100$ms and the packet loss rate $p_1 = p_2 = 2\%$. In the following scenarios, rather than requesting a file from server, we directly used iPerf3 [14] to continuously generate packets to keep the network busy.

Scenario 5: In this scenario, we tested DRL-CC's capability of dealing with the case with dynamic establishments and terminations of MPTCP flows. During a testing period of 150 seconds, establishments of MPTCP flows followed a Poisson process where the lambda was set to 10; and each flow lasted for 30 seconds. The average (over time) total goodputs of all MPTCP flows are shown in Fig. 5. We can see that DRL-CC is robust to such a highly-dynamic environment. Compared to the baselines, DRL-CC can still achieve 382%, 351%, 336%, 257% improvements on total goodput.

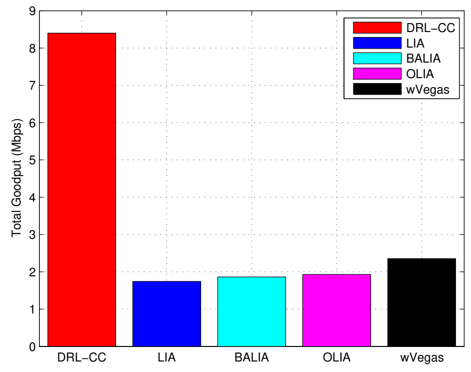


Fig. 5: Scenario 5: Average total goodput in the case with dynamic establishments and terminations of MPTCP flows

Scenario 6: As the number of subflows of a MPTCP flow may be changed during the running time, we considered the scenario where one of two subflows suddenly disappeared. Specifically, a total of 5 MPTCP flows were established in the beginning. During a testing period of 200 seconds, one of the subflows of each flow was closed via shutting down a network interface at 60s. The corresponding results are shown in Fig. 6. Similar as in the last scenario, DRL-CC can deliver robust performance in this dynamic case. Specifically, DRL-CC outperforms those baselines by 193%, 178%, 186%, 204% respectively. In order to show the behavior of flows and the performance of DRL-CC over time, we plot Fig. 7. We can see that DRL-CC experiences a sharp drop right at the subflow termination time 60s. However, we observe that its total average goodput stabilizes at 8Mbps (maximum possible after the termination), which shows that DRL can quickly adjust itself to the single network interface setting at 60s, and utilize the rest of available bandwidth.

In summary, we conclude that DRL-CC is robust to highly-dynamic network environments. As mentioned above, our design features an LSTM-based network that can learn an effective representation of all active flows. Unlike feed-forward neural networks or CNNs (commonly used in DRL), our model can well handle a variable input size (i.e., the cases with dynamic establishments and terminations of flows and subflows). We actually observed that DRC-CC was able to adjust its control policy quickly and properly whenever there was change during these experiments, which ensures good and stable overall performance. Our results have confirmed the effectiveness and robustness of our design.

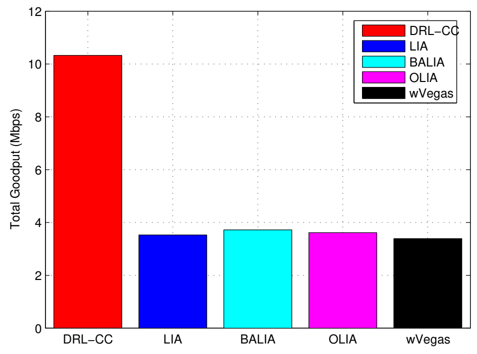


Fig. 6: Scenario 6: Average total goodput in the case with dynamic terminations of MPTCP subflows

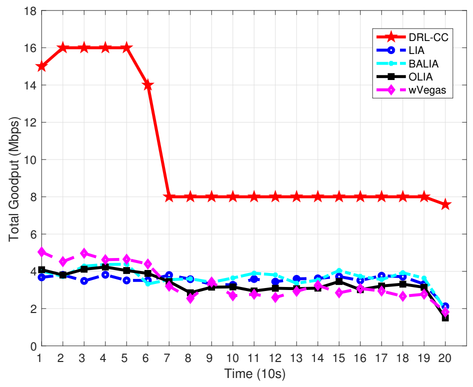


Fig. 7: Scenario 6: Average total goodput over time in the case with dynamic terminations of MPTCP subflows

Another important property of MPTCP is its friendliness to (regular) TCP flows. If there simultaneously co-exists both MPTCP and TCP flows in a network, MPTCP should not bring goodput improvements for its own flows at the cost of those TCP flows. It is quite common to have both TCP and MPTCP flows in a network since some servers/clients may not support MPTCP.

Scenario 7: We designed this scenario to evaluate the goodputs of all active flows given by all these congestion control methods in a MPTCP and TCP co-existing environment. In this scenario, there were a total of 5 MPTCP flows (that used two different links (path) for communications as described above), and 5 regular single-path TCP flows that competed for one of MPTCP's links. Those key parameters were set as follows: $b_1 = b_2 = 8$Mbps, $d_1 = d_2 = 100$ms and $p_1 = p_2 = 2\%$ We measured the average per-flow goodput for both TCP and MPTCP, and presented the results on Fig. 8. We can see that if DRL-CC is used, the goodput of a TCP

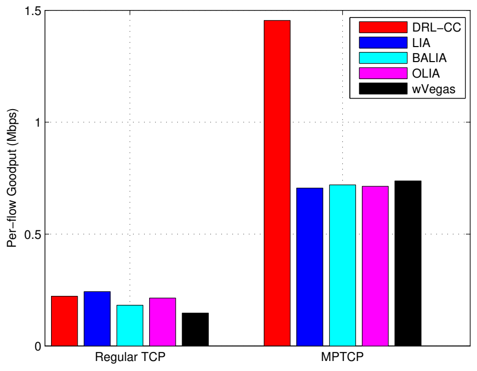


Fig. 8: Scenario 7: Per-flow goodputs of regular TCP and MPTCP flows

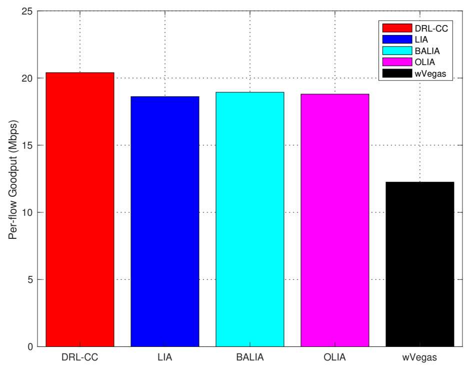


Fig. 9: Scenario 8: Average per-flow goodput in the case with asymmetric wireless links

flow is quite similar as those given by the baselines; however, the corresponding MPTCP flows have a much higher goodput. Specifically, compared to the best baseline LIA, the per-flow TCP goodput corresponding to the use of DRL-CC is slightly lower but the corresponding MPTCP goodput is 106% higher. Moreover, DRL-CC offers higher goodputs for both TCP and MPTCP flows than all the other baselines. This is also mainly due to the way how we define the reward (Section III-A), particularly the utility function, which takes into account both TCP and MPTCP flows. This observation confirms that DRL-CC is TCP-friendly.

In addition, we conducted an additional experiment in a practical wireless environment, in which a laptop was equipped with two WiFi network interfaces and there existed asymmetric links. Most of other settings are the same as those in the above scenarios, and there were 5 MPTCP flows in total.

Scenario 8: In this scenario, we aimed to demonstrate how DRL-CC performs in a practical wireless environment with asymmetric links. The bandwidth and delay were limited to $b_1 = 100$Mbps and $b_2 = 10$Mbps; and $d_1 = 5$ms and $d_2 = 2$ms respectively. We measured the average per-flow goodput for MPTCP flows, and presented the corresponding results in Fig. 9.

Since both the delay and the packet loss ratio are fairly low in this scenario, the performance gaps between different methods are relatively smaller compared to other scenarios but they are still noticeable. Specifically, we can see that DRL-CC still outperforms all the other baseline methods in terms of goodput, by 9.6%, 7.7%, 8.51% and 66.5% on average respectively. Thus, we can conclude that DRL-CC is able to make good use of available bandwidth and perform consistently well under different network conditions such as those with asymmetric characteristics.

V. Related Work

Congestion Control: Congestion control, as a fundamental problem in networking, has been widely studied in the context of TCP [2], [13]. Most of the classical congestion control algorithms proposed for regular TCP are either loss-sensitive (such as NewReno [13]), or delay-sensitive (such as TCP Vegas [2]). They usually pre-define some packet-level events as congestion signals, and conduct the congestion window adjustment based on a fixed control policy. Recently, several works targeted at learning a control policy from the runtime state. In a pioneering work [41], the authors presented a congestion control approach called Remy, which can generate control rules for different cases. Dong *et al.* [3] proposed Performance-oriented Congestion Control (PCC), in which each sender continuously observes the connection between its actions and empirically experienced performance, enabling it to consistently adopt actions that result in high Performance. The authors of [44] introduced a congestion control protocol, Verus, which continuously learns a delay profile that captures the correlation between end-to-end packet delay and the outstanding window size, and uses this correlation to adjust the congestion window.

Unlike these related works targeting at the regular TCP, we aim to optimize performance of more recent MPTCP, which is quite different from MPTCP. It has also been shown [26], [27] that MPTCP may suffer from serious performance degradation when directly applying a regular TCP congestion control algorithm separately on each sub-flow.

Congestion control is also a critical problem for MPTCP and has also been well studied recently in the literature. In [20], Michio *et al.* proposed a congestion control scheme, which enables an end-to-end connection that uses flows along multiple paths to fairly compete with TCP flows at shared bottlenecks, and in the meanwhile, maximizes the utilization of different paths. Hassayoun *et al.* [11] proposed Dynamic Window Coupling (DWC), a multipath congestion control mechanism that seeks to be fair to other flows in the network while being able to maximize its own throughput. DWC detects shifting bottlenecks in the network and responds by dynamically regrouping subflows. In [27], Raiciu *et al.* designed Linked Increase Algorithm (LIA), which couples the congestion control policies running on different subflows by linking their increase functions. The authors of [17] presented Opportunistic Linked Increase Algorithm (OLIA), which resolves some performance issues of LIA while retaining non-flappiness and responsiveness. As an extension of the well-known TCP Vegas [2], the authors of [42] proposed weighted Vegas for MPTCP, which adopts the packet queuing delay as a congestion signal, achieving fine-grained load balancing. In [26], Peng *et al.* proposed BAlanced LInked Adaptation (BALIA) to generalize existing congestion control algorithms through a fluid model and strike a good performance. In [5], the authors quantified the penalty of the coupled congestion control for links that do not share a bottleneck, then designed and implemented a practical shared bottleneck detection (SBD) algorithm for MPTCP, namely MPTCP-SBD, to overcome the penalty. A recent work [45] presented MPTCPD, an energy-efficient variant of MPTCP particularly for datacenters, which can provide energy efficiency by minimizing the flow completion time. Morevoer, Le *et al.* [16] developed ecMTCP, which is an energy-efficient congestion control algorithm. Dong *et al.* [4] designed mVeno particularly for wireless communications with multiple radio interfaces. Raiciu *et al.* [28] implemented MPTCP in Linux kernel and evaluated its performance. They mainly focused on the algorithms needed to efficiently use paths with different characteristics, notably send and receive buffer tuning and segment reordering. They also compared the performance of their implementation with regular TCP on web servers.

As mentioned above, unlike these related works presenting pre-defined policies for congestion control in MPTCP, we develop a novel model-free experience-driven framework based on DRL, which learns the best control policy based on real-time runtime states. Moreover, the proposed framework features a novel end-to-end trainable DNN model for action inference, which can even deal with a variable input size.

Deep Reinforcement Learning (DRL): DRL has won his world-wide fame due to its impressive successes on game-playing tasks such as Go and Atari games. It has recently attracted extensive research attention from both industry and academia. In a pioneering work [21], Mnih *et al.* proposed deep Q-learning and DQN, which can learn successful policies directly from high dimensional sensory inputs. As introduced above, they introduced two new techniques, experience replay and target network, to ensure learning stability. The authors of [10] proposed Double Q-learning as a specific adaptation of the DQN and an improvement to the earlier work [21]. Another improvement was introduced in [29] to use prioritized experience replay in DQN such that important transition samples can be replayed more frequently, which can lead to more efficient learning. In [39], Wang *et al.* presented a new dueling neural network architecture, which includes two separate estimators: one for the state value function and one for the state-dependent action advantage function. So far, we only discuss works related to discrete control with a limited action space. Continuous control has also be addressed in the context of DRL. Lillicrap *et al.* [18] proposed an actor-critic-based and model-free algorithm, DDPG, based on the deterministic policy gradient, which represents a state-of-the-art DRL-based solution to continuous control. Gu *et al.* [7] proposed normalized advantage functions for reducing sample complexity for continuous control. In [8], the authors proposed an interesting policy gradient method Q-Prop, which uses a Taylor expansion of the off-policy critic as a control variant. The authors of [22] proposed asynchronous gradient descent for optimizing learning with DNNs, and showed its successes on a wide variety of continuous motor control tasks.

Even though, DRL has made tremendous successes, the research on the feasibility and effectiveness of using it in the context of quite different network control problems is still in its infancy. To the best of our knowledge, we are the first to leverage DRL for congestion control in DRL. Moreover, the neural network architecture in DRL-CC is different from those in all these related works.

VI. Conclusion

In this paper, we presented design, implementation and evaluation of a DRL-based framework, DRL-CC, for congestion control in MPTCP. DRL-CC utilizes a single agent to dynamically and jointly perform congestion control for all active MPTCP flows on an end host with the objective of maximizing the overall utility (such as goodput). DRL-CC features a novel end-to-end trainable DNN model for action inference, which consists of a flexible LSTM-based representation network, a critic network and an actor network. This neural network architecture can be used to learn an effective representation of all active TCP and MPTCP flows to enable the above joint control, and deal with network dynamics with time-varying flows. We implemented DRL-CC based on the MPTCP implementation in the Linux kernel. We conducted a comprehensive empirical study to evaluate the performance of DRL-CC under seven different scenarios. The experimental results have well justify its effectiveness and superiority over a few well-known MPTCP congestion control algorithms (including LIA, OLIA, BALIA and wVegas) in all of these scenarios in terms of goodput and fairness; its robustness to highly-dynamic environments with time-varying flows; as well as its friendliness to the regular TCP.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen *et al.*, Tensorflow: Large-scale machine learning on heterogeneous distributed systems, *arXiv preprint*, 2016, arXiv:1603.04467.

[2] L. L Brakmo and L. L Peterson, TCP Vegas: End to end congestion avoidance on a global Internet, *IEEE Journal on selected Areas in communications*, Vol. 13, No. 8, 1995, pp. 1465–1480.

[3] M. Dong, Q. Li, D. Zarchy, P. B Godfrey and M. Schapira, PCC: Re-architecting congestion control for consistent high performance, *USENIX NSDI'2015*, 2015.

[4] P. Dong, J. Wang, J. Huang and H. Wang and G. Min, Performance enhancement of multipath TCP for wireless communications with multiple radio interfaces, *IEEE Transactions on Communications*, Vol. 64, No. 8, 2016, pp. 3456–3466.

[5] S. Ferlin, O. Alay, T. Dreibholz, D. A Hayes, D. A. Hayes and M. Welzl, Revisiting congestion control for multipath TCP with shared bottleneck detection, *IEEE INFOCOM'16*, 2016.

[6] A. Ford, C. Raiciu, M. Handley and O. Bonaventure, TCP extensions for multipath operation with multiple addresses, *IETF RFC 6824*, 2013.

[7] S. Gu, T. Lillicrap, I. Sutskever and S. Levine, Continuous deep Q-Learning with model-based acceleration, *ICML'16*, pp. 2829–2838.

[8] S. Gu, T. Lillicrap, Z. Ghahramani, R. Turner and S. Levine, Q-prop: Sample-efficient policy gradient with an off-policy critic, *arXiv preprint*, 2016, arXiv:1611.02247.

[9] P. Gupta and P. R. Kumar, The capacity of wireless network, *IEEE Transactions on Information Theory*, Vol. 46, No. 2, 2000, pp. 388–404.

[10] H. v. Hasselt, A. Guez, and D. Silver, Deep reinforcement learning with double Q-learning, *AAAI'16*, pp. 2094–2100.

[11] S. Hassayoun, J. Iyengar and D. Ros, Dynamic window coupling for multipath congestion control, *IEEE ICNP'11*, 2011, pp. 341-352.

[12] S. Hochreiter and J. Schmidhuber, Long Short-Term Memory, *Neural Computation*, Vol. 9, No. 8, 1997, pp. 1735-1780.

[13] J. C. Hoe, Improving the start-up behavior of a congestion control scheme for TCP, *ACM SIGCOMM'96*, Vol 26, No. 4, 1996, pp. 270–280.

[14] iPerf3: *https://iperf.fr/*

[15] D. Kingma and J. Ba, Adam: a method for stochastic optimization, *ICLR'15*, 2015.

[16] T. A. Le, C. Hong, M. Razzaque, S. Lee and H. Jung, ecMTCP: an energy-aware congestion control algorithm for multipath TCP, *IEEE communications letters*, Vol. 16, No. 2, 2012, pp, 275–277.

[17] R. Khalili, N. Gast, M. Popovic, U. Upadhyay and J. Le Boudec, MPTCP is not Pareto-optimal: performance issues and a possible solution, *ACM CoNEXT'12*, 2012.

[18] 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, *ICLR'16*.

[19] S. H Low and D. E Lapsley, Optimization flow control. I. Basic algorithm and convergence, *IEEE/ACM Transactions on networking*, Vol. 518, No. 6, 1999, pp. 861–874.

[20] H. Michio, *et al.* , Multipath congestion control for shared bottleneck, *PFLDNeT Workshop*, 2009.

[21] V. Mnih, *et al.* , Human-level control through deep reinforcement learning, *Nature*, Vol. 518, No. 7540, 2015, pp. 529–533.

[22] V. Mnih, *et al.* , Asynchronous methods for deep reinforcement learning, *ICML'16*, pp. 1928–1937.

[23] netem: *https://wiki.linuxfoundation.org/networking/netem*

[24] C. Paasch, S. Barre, *et al.* , Multipath TCP in the Linux Kernel, *https://www.multipath-tcp.org*

[25] D. P. Palomar and M. Chiang, A tutorial on decomposition methods for network utility maximization, *IEEE Journal on Selected Areas in Communications*, Vol. 24, No. 8, 2006, pp. 1439–1451.

[26] Q. Peng, A. Walid, J. Hwang and S. H Low, Multipath TCP: analysis, design, and implementation, *IEEE/ACM Transactions on Networking*, Vol. 24, No. 1, 2016, pp. 596–609.

[27] C. Raiciu, M. Handley, and D. Wischik, Coupled congestion control for multipath transport protocols, *RFC 6356*, 2011.

[28] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure and M. Handley, How hard can it be? designing and implementing a deployable multipath TCP, *USENIX NSDI'12*, 2012.

[29] T. Schaul, J. Quan, I. Antonoglou and D. Silver, Prioritized experience replay, *arXiv preprint*, 2015, arXiv:1511.05952

[30] R. Z. Shen, Valiant Load-Balancing: building networks that can support all traffic matrices, Algorithms for Next Generation Networks, 2010.

[31] J. F. Shortle, J. M. Thompson, D. Gross and C. M. Harris, Fundamentals of queueing theory (5th Edition), *Wiley*, 2018.

[32] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller, Deterministic policy gradient algorithms, *ICML'14*, 2014.

[33] R. Srikant, The mathematics of Internet congestion control, *Springer Science & Business Media*, 2012.

[34] I. Sutskever, O. Vinyals, and Q. V Le, Sequence to sequence learning with neural networks, *NIPS'14*, pp. 3104-3112.

[35] R S Sutton, D A McAllester, S P Singhand and Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, *NIPS'00*, 2000, pp. 1057–1063.

[36] R. S. Sutton and A. G. Barto, Reinforcement learning: an introduction 2nd Edition), *MIT press*, 2018.

[37] TFLearn: *http://tflearn.org/*

[38] Tcpdump: *https://www.tcpdump.org/*

[39] Z. Wang, T. Schaul, M. Hessel, H. Van, M. Lanctot and N. De Freitas, Dueling network architectures for deep reinforcement learning, *ICML'16*, pp. 1995–2003.

[40] wireshark: available from https://www.wireshark.org/

[41] K. Winstein and H. Balakrishnan, TCP ex machina: computer-generated congestion control, *ACM SIGCOMM'13*, 2013, pp. 123–134.

[42] M. Xu, Y. Cao and E. Dong, Delay-based Congestion Control for Multipath TCP, *IEEE ICNP'12*, 2015.

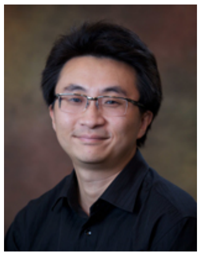
[43] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu and D. Yang, Experience-driven networking: a deep reinforcement learning based approach, *IEEE INFOCOM'18*, 2018.

[44] Y. Zaki, T. Ptsch, J. Chen, L. Subramanian, & C. Grg, Adaptive congestion control for unpredictable cellular networks, *ACM SIGCOMM'15*, Vol. 45, No. 4, 2015, pp. 509–522.

[45] J. Zhao, J. Liu, and H. Wang and C. Xu, Multipath TCP for datacenters: From energy efficiency perspective, *IEEE INFOCOM'17*, 2017, pp. 1–9.

[46] D. Zhou, W. Song and M. Shi, Goodput Improvement for Multipath TCP by Congestion Window Adaptation in Multi-Radio Devices, *IEEE CCNC'13*, 2013, pp. 508–514.

Zhiyuan Xu is currently pursuing the Ph.D. degree at the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, USA. He received the B.E. degree in School of Computer Science and Engineering from University of Electronic Science and Technology of China, Chengdu, China, in 2015. He was an exchange student in 2013 at Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan. He was a visiting student in 2015 at Dalhousie University, Halifax, NS, Canada. His current research interests include deep reinforcement learning and communication networks.



Jian Tang (F19) is a professor in the Department of Electrical Engineering and Computer Science at Syracuse University. He received his Ph.D degree in Computer Science from Arizona State University in 2006. His research interests lie in the areas of Wireless Networking, Machine Learning, Big Data and Cloud Computing. Dr. Tang has published over 130 papers in premier journals and conferences. He received an NSF CAREER award in 2009, the 2016 Best Vehicular Electronics Paper Award from IEEE Vehicular Technology Society (VTS), and Best Paper Awards from the 2014 IEEE International Conference on Communications (ICC) and the 2015 IEEE Global Communications Conference (Globecom) respectively. He has served as an editor for a few IEEE journals, including IEEE Transactions on Big Data, IEEE Transactions on Mobile Computing, IEEE Transactions on Network Science and Engineering, IEEE Transactions on Wireless Communications, IEEE Internet of Things Journal and IEEE Transactions on Vehicular Technology. In addition, he served as a TPC co-chair for the 2019 IEEE/ACM International Symposium of Quality of Service (IWQoS), the 2018 International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous), the 2015 IEEE International Conference on Internet of Things (iThings) and the 2016 International Conference on Computing, Networking and Communications (ICNC); as the TPC vice chair for the 2019 IEEE International Conference on Computer Communications (INFOCOM); and as an area TPC chair for INFOCOM 2017-2018. He is also an IEEE fellow, an IEEE VTS distinguished lecturer, and the vice chair of the Communications Switching and Routing Committee of IEEE Communications Society.

Chengxiang Yin is currently pursuing the Ph.D. degree at the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, USA. He received his B.S. degree from the School of Information and Electronics at Beijing Institute of Technology, Beijing, China, in 2016. His research interests include Deep Learning and Computer Vision.

Yanzhi Wang is currently an assistant professor in the Department of Electrical and Computer Engineering at Northeastern University. He has received his Ph.D. Degree in Computer Engineering from University of Southern California (USC) in 2014, and his B.S. Degree with Distinction in Electronic Engineering from Tsinghua University in 2009. Dr. Wang's current research interests are the energy-efficient and high-performance implementations of deep learning and artificial intelligence systems. Besides, he works on the application of deep learning and machine intelligence in various mobile and IoT systems, medical systems, and UAVs, as well as the integration of security protection in deep learning systems. His works have been published in top venues in conferences and journals (e.g. ASPLOS, MICRO, HPCA, ISSCC, AAAI, ICML, ICLR, ECCV, ACM MM, CCS, VLDB, FPGA, DAC, ICCAD, DATE, LCTES, INFOCOM, ICDCS, Nature SP, etc.), and have been cited for around 4,000 times according to Google Scholar. He has received four Best Paper Awards, has another seven Best Paper Nominations and two Popular Papers in IEEE TCAD. His group is sponsored by the NSF, DARPA, IARPA, AFRL/AFOSR, and industry sources.

Guoliang Xue (F09) is a professor of Computer Science and Engineering at Arizona State University. He earned a PhD degree in Computer Science in 1991 from the University of Minnesota, an MS degree in Operations Research in 1984, and a BS degree in Mathematics in 1981, both from Qufu Normal University. His research interests include resource allocation in computer networks, security and survivability issues in networks, and machine learning enabled crowdsourcing. He is an Area Editor of IEEE Transactions on Wireless Communications for the Wireless Networking Area overseeing 12 editors. He is a past editor of IEEE/ACM Transactions on Networking, and Computer Networks. He was a TPC co-chair of IEEE INFOCOM2010 and a co-General Chair of IEEE CNS2014. He was a Keynote Speaker at IEEE LCN2011, IEEE ICNC2014, and IEEE ICT-DM2018, and IFIP WWIC2018. He is an IEEE Fellow. He served as the VP-Conferences of the IEEE Communications Society (ComSoc) in 2016 and 2017.