Pareto: Fair Congestion Control With Online Reinforcement Learning

Salma Emara, *Graduate Student Member, IEEE*, Fei Wang, Baochun Li, *Fellow, IEEE*, and Timothy Zeyl

***Abstract*—Modern-day computer networks are highly diverse and dynamic, calling for fair and adaptive network congestion control algorithms with the objective of achieving the best possible throughput, latency, and inter-flow fairness. Yet, prevailing congestion control algorithms, such as hand-tuned heuristics or those fueled by deep reinforcement learning agents, may struggle to perform well on multiple diverse networks. Besides, many algorithms are unable to adapt to time-varying real-world networking environments; and some algorithms mistakenly overlooked the need of explicitly taking inter-flow fairness into account, and just measured it as an afterthought. In this paper, we propose a new staged training process to train *Pareto*, a new congestion control algorithm that generalizes well to a wide variety of environments. Different from existing congestion control algorithms running reinforcement learning agents, Pareto is trained for fairness using the first multi-agent reinforcement learning framework that is communication-free. Pareto continues training online adapting to newly observed environments in the real-world. Our extensive array of experiments shows that Pareto (i) performs well in a wide variety of environments, (ii) offers the best fairness when it comes to competing with other flows sharing the same network link, and (iii) improves its performance with online learning to surpass the state-of-the-art.**

***Index Terms*—Congestion control, deep reinforcement learning, fairness, online learning.**

I. Introduction

FOR over a quarter of a century, it is a fundamental challenge in networking research to design the best possible congestion control algorithms that optimize throughput and end-to-end latencies. Research interests in congestion control had recently been increasing as cloud applications have shown strong demands for higher throughput and lower latencies (e.g. [1]–[4]).

The design of congestion control algorithms is particularly challenging when multiple clients share a network in an uncoordinated and decentralized manner. Each client in the network has a limited number of observations it can perform to understand the state of the network. In most cases, clients do not have any access to information about other clients sharing the network, such as their demand, delivery rates, round-trip times (RTTs), or loss rates. The fundamental challenge is to achieve a fair and optimal share of resources despite having such limited visibility.

There are a number of important problems with the prevailing congestion control algorithms. First, especially reinforcement learning agents, these algorithms often overlook the need of explicitly taking fairness into account. As a result, they do not encourage fair behavior when competing with other flows sharing the same bottleneck network bandwidth, which was exhibited in Aurora [5] and Eagle [3]. We argue that fairness should be an important objective to consider when designing a new congestion control algorithm.

Second, unlike conventional algorithms such as BBR and CUBIC, some of these congestion control algorithms adapt slowly online as they learn from live evidence (e.g. PCC [6] and PCC-Vivace [2]). This solves the issue from the lack of online adaptation; however, adaptation in these algorithms is slow and can easily result in over or under utilization of the network.

Third, current congestion control algorithms are either hand-crafted or trained offline, and as a result, they use fixed mappings between network events and congestion control responses. The fixed set of rules is either manually designed (e.g., BBR [1] and CUBIC [7]) or previously learned actions on observed states from simulated environments (e.g., Remy [8], Indigo [9], Aurora [5], Eagle [3] and Orca [4]). Since these fixed rules may not always apply or the network's dynamics may deviate from those simulated environments, these congestion control algorithms may not perform well on a wide variety of environments, and hence can lack generalization.

Deep Reinforcement learning (DRL) has been used to develop several recent congestion control algorithms, such as Aurora [5], Eagle [3], Orca [4], TCP-RL [10] and DeepCC [11]. However, they were all trained offline, and hence they were not designed to adapt online and hence their mappings between states and actions were fixed. In addition, they were also not trained to be aware of competing flows on the network, and hence their behavior in shared networks was not studied/analyzed. However, Orca [4] and DeepCC [11] depend mainly on CUBIC to exhibit fair behavior.

In this paper, we propose *Pareto*, a new DRL-based congestion control algorithm that overcomes the aforementioned

Manuscript received 30 December 2021; revised 12 May 2022; accepted 16 June 2022. Date of publication 22 June 2022; date of current version 9 September 2022. Recommended for acceptance by Prof. Bo Ji. *(Corresponding author: Salma Emara.)*

Salma Emara, Fei Wang, and Baochun Li are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: salma@ece.utoronto.ca; silviafey.wang@utoronto.ca; bli@ece.toronto.edu).

Timothy Zeyl is with Huawei Canadian Research Institute, Markham, ON L3R 5A4, Canada (e-mail: timothy.zeyl@huawei.com).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNSE.2022.3185253, provided by the authors.

Digital Object Identifier 10.1109/TNSE.2022.3185253

issues of generalization, online adaptation and fairness. Our algorithm seeks to achieve (i) fairness by training to be fair to competing flows in a shared network, (ii) adaptation by learning online while remembering previously learned experiences, and (iii) generalization by learning on a wide variety of environments.

First, aiming to have Pareto behave fairly towards competing flows sharing the same link, we introduce *a fairness training algorithm* in the training process of Pareto. Our fairness training algorithm is a contribution to the field of reinforcement learning as it allows multiple agents to be deployed in a shared environment to achieve a common goal (fairness) *without* inter-agent communication.

Second, to avoid having fixed mappings between network events and congestion control responses, we introduce *a new online training algorithm* that allows Pareto to adapt to newly seen environments. While our algorithm improves the performance of Pareto on newly observed environments *quickly*, it also ensures that old experiences gained in offline learning are *not forgotten*.

Lastly, to achieve generalization and optimal performance on a wide variety of environments, Pareto is trained offline over *a newly designed staged training algorithm*, which involves three stages: (i) bootstrapping, (ii) advancing and (iii) fairness training. Each of these stages exposes the model to increasingly challenging sets of observations by encountering more challenging environments.

We have performed an extensive array of experiments using Pantheon [9], and showed that Pareto's offline trained model performs well on a wide range of environments and is behaving fairly to competing flows on the same link. Online learning improves the behavior of Pareto in terms of all performance metrics: fairness, network utilization, latency and loss rate. Pareto has a better trade-off between throughput, latency and loss rate compared to other algorithms fueled by reinforcement learning by up to 68%, and can learn online to further improve this trade-off by a maximum of 11% to surpass BBR and CUBIC.

II. Deep Reinforcement Learning for Congestion Control: Motivation and Challenges

Rate-based congestion control requires adaptation to the sending rate, which is the rate at which the sender is sending data to the receiver. The sending rate of the sender should achieve maximum throughput while minimizing the round-trip time (RTT), which is the time elapsed between sending a packet and receiving its acknowledgement. It is also desirable to minimize packet losses, which is the ratio of packets lost to packets sent, and not causing starvation of concurrent flows to achieve inter-flow fairness [12].

Why learning-based? The benefit of learning-based congestion control algorithms is that they avoid making assumptions on rules that govern the network responses to congestion. So, in cases where the assumptions do not hold, heuristics may fail to act adequately; learning-based algorithms will create their own rules by learning from the network, which actions are optimal. It can take engineers years to design a congestion control algorithm [1], while machines can automatically learn and adapt in minutes (or hours) to new environments.

Limitations of learning-based algorithms. Current learning-based congestion control algorithms do not fully solve issues in heuristics. They try to mitigate issues of heuristics by learning congestion control rules through interacting with simulated environments, such as Remy [8], Indigo [9], Eagle [3] and Orca [4]. Since learning occurs on a small subset of environments, it is a challenge to generalize well to a wider variety of environments. Moreover, since previous works train models only offline, they do not adapt and hence may perform sub-optimally in newly seen environments. Also, as mentioned in Aurora [5], fairness can be difficult to attain. Other online learning-based congestion control algorithms can demonstrate slow convergence properties PCC-Vivace [2].

Therefore, this leaves a need for a learning-based algorithm that adapts online to newly observed network events by not forgetting old experiences learned, while generalizing well to a wide variety of environments. All of this should not comprise the fairness property towards concurrent flows.

Why use DRL? Why deep? A motivation to use *RL* is that it offers several advantages over supervised learning algorithms, including the ability to continue learning *online* [13]. A more nuanced advantage of agents trained using *RL*-based algorithms is their ability to remember the history of observations and detect trends in network conditions to act more appropriately [14].

As opposed to supervised learning, RL considers accumulated rewards instead of instantaneous rewards, which allows agents trained using RL-based algorithms to be "farsighted" [15]. This is essential for congestion control, since current actions can have long-lasting effects.

On the other hand, *deep* neural networks have the potential of learning from low-level features in raw input data high-level features, hence it does not require preprocessing input features.

While we argue that DRL fits the congestion control problem, there are prevailing issues in existing work using DRL for congestion control, such as Eagle [3] and Aurora [5], that we aim at solving. Eagle [3] and Aurora [5] do not train their DRL agents for fairness, hence they do not exhibit fair behavior with flows sharing the same network link. To mitigate this issue, we introduce a new fairness training algorithm that trains a fair DRL-based congestion control algorithm.

Moreover, existing works of Eagle [3] and Aurora [5] are not designed to adapt online to new environments. Their offline learning algorithms cannot be easily extendable to continually learn, since they assume that the episodes are sampled from a stationary distribution. While in online learning the distribution of episodes can be non-stationary and the stability of the model can be threatened and forget old experiences learned.

Lastly, Eagle [3] and Aurora [5] lack *generalization to a wide variety of environments*. If a DRL agent were only trained on a small subset of environments, it will perform well on a subset of network environments and fail to perform well

in other environments. At the same time, as we will show later theoretically and empirically, training a DRL agent on a wide variety of environments at once can confuse the agent and lead to a worse-performing agent in all environments it observed. To address this dilemma, we propose a new *staged training process* to help in the generalization of our DRL agent, where in each stage we introduce more challenging environments to the DRL agent.

In summary, if we were to use an existing DRL algorithm to solve the congestion control problem, we would not achieve fairness, generalization and online adaptation. In this paper, we introduce a fairness training algorithm, a staged training framework and an online training algorithm to achieve the aforementioned goals using DRL.

III. Formulating Congestion Control as an RL Task

A flow is a stream of packets transmitted by a sender across a network link to a receiver. A sender can initiate several flows to one or many receivers. Pareto learns a policy by running a DRL algorithm on the sender side to control the sending rate of each flow independently.[1] This is with the main objective of achieving the best trade-off between a variety of metrics, such as throughput,[2] queueing delay[3], loss rate and fairness with competing flows sharing the same link. In addition, we aim to improve the online adaptability of our algorithm using online training.

Briefly, Pareto at the sender side is responsible for making decisions on the sending rate. The receiver will respond with acknowledgements. Based on acknowledgements received from the receiver, Pareto will perform calculations to measure throughput, queueing delay and loss rate. These measurements will be used as input to train the policy represented by the agent. As the agent experiences different environments, it gains knowledge on how to behave in each environment.

Modelling our sequential decision-making process as a Markov Decision Process, which is a sequential decision process for *fully* observable stochastic environments where the effects of actions depend only on the current state, is not fully correct. This is because our congestion control network environment does not satisfy the Markov property, since our agent can perform better actions if it has more information about the history of process [16]. This is also proven empirically in Aurora [5].

For example, an agent can increase its sending rate keeping it below the maximum bandwidth measured earlier, and still observe delay. This can be self-induced delay, or induced by other agents sharing the network link. This uncertainty is due limited observability of the environment, or having *hidden states*. Knowing the history about the last maximum sending rate which did not induce delay will help the agent in the control process. Therefore, it is convenient to model our environment as a Partially Observable Markov Decision Process (POMDP) as introduced in [16]–[19].

More formally, a POMDP can be described as a tuple $\langle S, A, T, R, \Omega, O \rangle$, where S is the finite set of states, A is the finite set of actions, T is the state-transition function, R is the reward function, Ω is the finite set of observations the agent can experience and O is the observation function, which gives the distribution over all possible observations given a state and action [20]. If the environment model is known with specified T, O, R, then the agent can compute the belief state b, which is the probability distribution over state space S, using methods like MLS or Q-MDP approximations and SPOVA algorithm [21]. These methods also require that the state space is known. However, in many real-world application, including our congestion control network environment, the environment model and the state space is unknown.

Another approach to approximately solve POMDPs is building a model of the congestion control environment. This model would be used to estimate the belief state using particle filtering or other sampling methods as reviewed in [22].

Model-based approaches do not fit our problem for two main reasons. *First*, training a policy on a pre-trained model of the congestion control environment would negatively affect the performance of the model. This is mainly due to the difficulty of modelling a real-world problem of stochastic congestion control. Therefore, the policy will perform well on the learned/modelled environment but will behave sub-optimally in the real-world environment. This loss in performance has been studied by [23], [24].

Second, to continue with online adaptation, we will need to adapt the pre-trained model of the environment, which would require high computational resources of training two models online: the model of the environment and the policy.

Without a model of the congestion control network environment, the only way to gain new experiences is to interact with the environment and learn a policy, which motivates the usage of reinforcement learning [20], [25]. One model-free approach is resolving the hidden state by incorporating in the current observation some representation of the history of observations and/or actions, as in Aurora [5] and Orca [4]. This aims at yielding a Markovian state signal.

Since we are unsure of the history of observations to incorporate in the current state, there are disadvantages of increasing the history size if we were to use traditional reinforcement learning. Traditional reinforcement learning uses a lookup table to map memory of observations to actions, and increasing history size would increase the training time. Instead, using a *deep* neural network is vital to speed up convergence if we were to increase the size of the current state space to incorporate more past observations. Since it can be difficult to represent long (possibly entire) histories, [26] suggests using LSTM neural network to extract arbitrary long-term dependencies in the observations over time.

Since our congestion control task is having hidden states and is therefore POMDP, we resolve the hidden states by using an LSTM to extract relevant information from past observations to yield a Markovian state signal. To formally

[1] Throughout this paper, one sender manages only one flow.

[2] Throughout this paper, throughput and delivery rate are used interchangeability.

[3] Queueing delay is also referred to as *delay* or *latency* in this paper.

formulate the congestion control problem as a DRL task, we now elaborate on the design of the action space, step size, observations, neural network model and reward function.

Action space. We choose to map the agent's output to change the sending rate of the sender according to $S_t = S_{t-1} \times (1 + a_t)$, where S_t is the sending rate and a_t is the agent's output. The agent's output is continuous and can range from -0.7 to 2. The maximum possible gain $\frac{S_{t+1}}{S_t}$ is 3, while the minimum is 0.3, which is slightly lower than the inverse of the maximum. As mentioned in [1], for example, if a gain of 3 was used to discover the maximum bandwidth and a queue was built as a result, it is required to use the inverse of the maximum gain in the range of actions to get rid of the excess queue. The range of our action space is inspired by BBR, so that we are able to efficiently cover all the potential actions that BBR takes.

Step size. One challenge in our environment is the delayed effect of actions. We need to wait for one RTT to start acknowledging packets that were sent after taking the current decision. We choose *a step size of 2 × recently measured RTT*. Therefore, any measurements made during a step size is a combination of observations due to actions in the current and the previous steps. This is why our measured RTT, delivery rate and loss rate for a step are exponentially weighted moving averages (EWMAs) to weigh recent observations more than past observations. Hence, all our future references to delay, RTT, delivery rate and loss rate are EWMAs of measurements obtained within one step.

Since each flow started by the sender has an end time, we model our decision-making process as a *finite horizon problem*. During offline training, there are 60 steps in one episode, and in online learning, the episode size is determined by the length of a flow.

Observations. After the agent selects a sending rate S_t at time step t, basic measurements such as EMWAs of RTT, loss rate and delivery rate are obtained from received packet-acknowledgements. Using these basic measurements, the agent is fed the following input:

1) A binary indicator specifying whether we have experienced delay before in the flow;
2) the ratio change in delivery rate, ΔR_t, which is calculated as $\frac{R_t - R_{t-1}}{\min(R_{t-1}, R_t)}$, where R_t and R_{t-1} are the delivery rates measured in the current and previous steps, respectively;
3) the speed with which delay changes or the current rate of change in delay, which is calculated as $(\frac{dD_{curr}}{dT})_t = \frac{d_t - d_{t-1}}{d_{t-1}} / \Delta t$, where d_t and d_{t-1} are the delays observed in the current and previous steps, respectively, and Δt is the step size in milliseconds. The delay, d_t, is calculated as the difference between the current RTT and the minimum RTT observed during the lifetime of an episode;
4) the speed with which loss rate changes or the current rate of change in the loss rate, which is calculated as $(\frac{dL_{curr}}{dT})_t = \frac{l_t - l_{t-1}}{l_{t-1}} / \Delta t$, where l_t and l_{t-1} is the loss rate of the current and previous steps, respectively;
5) the ratio of excess packets X_t in the link, which is calculated as

$$X_t = \begin{cases} \frac{S_t - R_t}{R_t} & d_t \geq d_{\min} \\ 0 & d_t < d_{\min} \end{cases}$$

where S_t is the sending rate and $d_{\min}$ is the minimum allowable delay.

In all our calculations, the maximum allowable delay is 4 ms, over which we consider the sending rate exceeding the available bandwidth and yielding a true queueing delay. Practically, measuring the maximum throughput is impossible without causing some queueing delay and observing the delivery rate at that point. To avoid affecting the model's interpretation of the network environment and provide extra room for the agent to observe the maximum bandwidth and measure it correctly, we allow the agent to observe a small delay, $d_{\min}$.

The design of the state space aims at (i) generalizing the model to a wide variety of environments and (ii) coping with the limited availability of observations. Each element in the state space represents a feature of the current network state that helps in generalizing the trained model and/or combines measurements to combat the problem of having limited observability. First, the binary indicating if the flow experienced delay before resembles if the sending rate reached bottleneck bandwidth before. If no delay was experienced in the lifetime of the flow, then the model should explore the bottleneck bandwidth more aggressively to measure the bandwidth of the link.

Second, ΔR_t combines measurements of the delivery rate to form a better visualization of the network and the availability of bandwidth. If the value of ΔR_t is positive, the last action acquired more bandwidth; if it was negative, less bandwidth was acquired than what is available; and if the value is around 0, then either the sending rate has not changed or the sender was and remains sending at a rate higher than the available bandwidth in the previous and current step.

Third, ΔR_t, $(\frac{dD_{curr}}{dT})_t$ and $(\frac{dL_{curr}}{dT})_t$ ignores the absolute values of delivery rate, delay and loss rate, respectively, which may not generalize well to all environments. Instead, they focus more on the effects of the last action only. For example, the delay can be high, while a negative $(\frac{dD_{curr}}{dT})_t$ shows that the last action helped in decreasing delay. Since a longer step can change the delay drastically given the same action in a link with queueing delay, we consider the "rate" of change in delay to be fair across steps with different sizes. A model cannot observe if the network induces random losses unrelated to congestion, but reporting $(\frac{dL_{curr}}{dT})_t$ can ignore those by focusing on the rate of change in loss rates.

Lastly, X_t helps the agent understand how far the model is from the beginning of draining its queue to reduce delay. X_t combines measurements of the sending rate and delivery rate to report delay in a way that generalize well to all environments.

In other words, since it is desirable for our model to behave well in a wide range of environments, the state space represents the network state in the same way if a similar problem occurred but in two different networks. We ensure this using metrics that are based on ratios, or rates of change such as ΔR_t, $(\frac{dL_{curr}}{dT})_t$, $(\frac{dD_{curr}}{dT})_t$, or X_t, but not absolute network-specific measurements.

To weigh and emphasize features of the state space equally, we want to keep the range of all features same. We scale down ΔR_t and $(\frac{dD_{curr}}{dT})_t$ by 20 and 5 to weigh every element equally when it is fed to the neural network.

Neural network model. The challenge of having a real-world POMDP calls for a model that additionally has a deeper understanding of the environment over several time steps. A suitable neural network for this environment is a Recurrent Neural Network (RNN), which can retain information from sequential observations and thus take actions based on history [16]–[19]. Hence, *long-short term memory (LSTM)* is chosen as our neural network model. LSTM networks share their weights across different time steps, rather than only across current input features [27].

Each experience tuple in the experience replay buffer contains the steps with previous experience tuples to be fed to the neural network. Each time a sample is fed to the LSTM model, the past states in the sample are input first, and the last output of the last state is our action for the actor network or estimation of action-value for the critic network.

Reward function. Inspired by the four phases of a BBR flow [1], we design our reward function for three main phases: startup, queue draining and bandwidth probing phases. The *startup phase* starts when the flow commences and ends when it experiences its first queuing delay. During this phase, we expect the first measurement made to the available bandwidth. When a flow is experiencing a delay greater than its maximum allowable delay $d_{\min}$, the flow is in the *queue draining phase*. Here, the flow should aim at having no excess packets X_t in the link, and to draining the queue. After the queue is drained, the flow enters the *bandwidth probing phase*. During this phase, the sender needs to probe for additional available bandwidth.

In Algorithm 1, we explain the reward function used in each phase. To avoid divergence and ensure stability in training, we bound the reward between $[-50, 50]$. During the startup phase, we only care about increases in delivery rate reflected in ΔR_t. If the delivery rate is decreasing, the reward is the minimum reward, which is -50, otherwise the reward is equal to $10 \times \Delta R_t$. As observed, ΔR_t is between $[-2, 2]$, so we multiply it by 10 to have rewards spread within bounds of the reward.

If it is the first time to experience delay, then this is the first time to see maximum throughput. In this first step of delay observance, we target increases in delivery rate and slightly penalize increases in delay. $(\frac{dD_{curr}}{dT})_t$ has an observed maximum of 60 in the first occurrence of delay, and ΔR_t has a maximum of 20. By multiplying $(\frac{dD_{curr}}{dT})_t$ with a small factor of -0.2, and ΔR_t with 10, we reward increases in delivery rate more than penalizing increases in delay. As a result, the reward is set to $-0.2 \times (\frac{dD_{curr}}{dT})_t + 10 \times \Delta R_t$.

If delay persists in subsequent steps, we strictly penalize delay presence by multiplying X_t with 10, where X_t lies between $[0, 1]$, and penalize increases in loss rate reflected in $(\frac{dL_{curr}}{dT})_t$, which range between $[0, 1]$ by multiplying it with a factor of -5. Increases in loss rate are a reflection of long delays; therefore, X_t and $(\frac{dL_{curr}}{dT})_t$ are not weighted equally.

Algorithm 1: Reward Function Calculation in one Episode

while episode is not over **do**
  if startup phase: $d_t < d_{\min}$ and not experienced delay before **then**
    if $\Delta R_t < 0$ **then**
      $r_t = -50$
    else
      $r_t = 10 \times \Delta R_t$
  if queue draining phase: $d_t \geq d_{\min}$ **then**
    if Did not experience delay before **then**
      $r_t = -0.2 \times (\frac{dD_{curr}}{dT})_t + 10 \times \Delta R_t$
    else
      $r_t = -5 \times (\frac{dL_{curr}}{dT})_t - 10 \times X_t$
      if $(\frac{dD_{curr}}{dT})_t$ not 0 **then**
        $r_t \leftarrow r_t - 0.25 \times ((\frac{dD_{curr}}{dT})_t - (\frac{dD_{curr}}{dT})_{t-1})$
      else
        $r_t \leftarrow r_t - 15 \times (\frac{dL_{curr}}{dT})_t$
  if bandwidth probing phase: $d_t < d_{\min}$ and experienced delay before **then**
    if 1st step out of delay **then**
      $r_t = -(\frac{dD_{curr}}{dT})_t + 10 \times \Delta R_t$
    else
      $r_t = 50 \times \Delta R_t$
  $r_t = clip(-50, 50)$

We additionally penalize any increase in $(\frac{dD_{curr}}{dT})_t$ between steps, because the agent decided to increase delay further though it already had increases in delay in the previous step reflected in $(\frac{dD_{curr}}{dT})_{t-1}$. The penalty is $0.25 \times ((\frac{dD_{curr}}{dT})_t - (\frac{dD_{curr}}{dT})_{t-1})$ subtracted from the reward.

We do not expect changes in the delay if the buffers in the route are full; instead, we expect jumps in the loss rate. Hence, if we have delays, and we do not see changes in delay $(\frac{dD_{curr}}{dT})_t$, we escalate the penalties further by adding $-15 \times (\frac{dL_{curr}}{dT})_t$.

Finally, during the bandwidth probing phase, if it is the first step out of queue draining, we want to penalize aggressive decreases to delivery rate and reward decreases in delay. We achieve the aforementioned by setting the reward to $-(\frac{dD_{curr}}{dT})_t + 10 \times \Delta R_t$, where $(\frac{dD_{curr}}{dT})_t$ has a minimum of -4 and ΔR_t has a maximum of 0.5.

Otherwise, if it is not the first step out of queue draining, we only focus on increases to the delivery rate in $50 \times \Delta R_t$. As most of the bandwidth acquisition happened during startup phase, ΔR_t is having a smaller range $[-0.4, 0.4]$ in bandwidth probing phase. Hence, we choose 50 as a multiplying factor instead of 10 to equally favor increases in the delivery rate in both phases.

IV. The Training Process of Pareto

Pareto uses a model-free approach employing Twin Delayed Deep Deterministic Policy Gradients (TD3) [28], which is an actor-critic reinforcement learning algorithm, to parametrize its policy. TD3 performs exceptionally well in continuous control action-space problems [28], and consequently is an excellent match for congestion control.

Staged training process helps in reducing interference. It is important to differentiate between two terms: interference and

catastrophic forgetting in DRL. Interference is when there are two or more tasks that are incompatible. This leads to catastrophic forgetting where the model's performance degrades in one task because another task is overwriting the model's behavior.

Interference is measured as the inner product of two gradients, representing their alignment. For example, the interference between two-gradient based processes with objective J computed on u and v and are sharing parameters $\boldsymbol{\theta}$ is in (1). u and v are two different samples, tasks or entire distributions [29].

$$\rho_{u,v} = \frac{\partial J(u)}{\partial \boldsymbol{\theta}} \cdot \frac{\partial J(v)}{\partial \boldsymbol{\theta}} \tag{1}$$

When a model is learning two destructively interfering tasks, gradient updates from samples of two different tasks are in different directions with their dot product or $\rho_{u,v} < 0$. This definitely leads to a decrease in the convergence speed or possibly divergence. The opposite occurs if the two tasks are constructive in nature, i.e. $\rho_{u,v} > 0$.

Congestion control environments can be destructive since they are diverse. Training our agent on two destructively interfering environments at the same time will result in having $\rho_{u,v} < 0$ and hence may lead to lack of convergence. Instead, to ensure convergence, we initially train our agent on a subset of environments that are constructive to ensure that the interference between any two tasks is positive $\rho_{u,v} > 0$. Theoretically, when the model converges, the gradients approach zero.

Lemma: After the convergence of the model on the subset of constructive tasks u, when we train on task v along with u, where u and v are destructively interfering on any $\boldsymbol{\theta}$, interference $\rho_{u,v} \approx 0$.

Proof: Assuming $\| \frac{\partial J(v)}{\partial \boldsymbol{\theta}} \| < \infty$, since $\| \frac{\partial J(u)}{\partial \boldsymbol{\theta}} \| \approx 0$ as model converged in task u, $\frac{\partial J(u)}{\partial \theta} \cdot \frac{\partial J(v)}{\partial \theta} \approx 0$. Therefore, learning task v will be orthogonal on the knowledge gained from task u. ■

Therefore, avoiding negative interference by separating destructively interfering tasks into stages of constructive tasks will help speed up convergence and avoid divergence. Hence, we introduce our staged training algorithm for congestion control, which avoids training on destructively interfering environments at the same stage.

During offline learning, we train Pareto offline over three main stages: (i) bootstrapping, (ii) advancing, (iii) fairness. Each of these steps introduces more challenging environments, which would be interfering if the model observes them all at once. Our staged offline learning process aim to (i) reduce interference in each stage, (ii) generalize well to a wide variety of environments, and (iii) create a model that behaves fairly to other flows sharing the same network. After offline learning, Pareto continues to learn online to (i) continue adapting to newly seen environments and (ii) improve performance on previously observed environments.

Bootstrapping stage. At the beginning of the first training stage, our model parameters are initialized randomly, hence actions are random when training commences. To build up experience, we train the model on *elementary* environments, where the bandwidth and RTT is fixed, and no other flow is sharing the link. Here, the model's task is to find the *single* highest sending rate that does not build up delay or cause packet losses.

Advancing stage. As the model converges during the bootstrapping stage, we start broadening and intensifying the set of training environments to include both dynamic and fixed bandwidths with fixed RTTs and unshared links. Instead of searching for the *single* highest sending rate, the model's task is to continuously adapt to changing bandwidth and search for the highest sending rate that does not increase delay or result in packet losses.

Fairness stage. In the final stage of offline learning, we introduce shared bandwidth environments. During this fairness training stage, the model observes one other flow sharing the link. The link has fixed bandwidth and RTT. The task of the model is to have flows share fairly the link, even if flows exit or join the link at different times.

Shared experience replay buffer. During the fairness training stage, each sender sharing a network link will use their local model to take decisions on the sending rate. Each sender will start an episode of sending to the receiver for 30 seconds and the senders start the flow instances 2 seconds apart.

In the original TD3 algorithm, an experience replay buffer stores experience tuples (s_t, a_t, r_t, s_{t+1}) of each step, where s_t and a_t are the observed state and action taken, respectively, at the beginning of a step. While r_t and s_{t+1} are the reward and observed state, respectively, at the end of a step. These experience tuples are used to update actor and critic model parameters. To achieve fairness during fairness training, all senders (which is one in our case) will send the experience tuples of the last episode to the head sender, which stores them in the shared experience replay buffer. The head sender will then update and share the new model parameters with all the senders.

The target of the model during the fairness training stage is to decide the optimum sending rate in the best interest of all flows without communicating with one another. Intuitively, each sender would aim at maximizing their throughput without causing delay or packet losses. Since our reward function penalizes increases in delays, whenever a flow in a shared network decides to increase its sending rate to take more than its fair share of the link, it would be penalized. By the same token, this would induce delay in the other flows sharing the same link, penalizing them too. This may cause all flows to decrease their sending rate as a reaction.

The shared experience replay buffer allowed all senders to share their observations. Therefore, the agent learns that if increasing the sending rate was accompanied by increases in delay, then there is a higher likelihood this delay is self-induced. By the same token, the agent learns that if it did not increase the sending rate and did observe delay, the delay is likely caused by another aggressive flow.

In addition, when the link reaches a steady state, no flow has an incentive to change its sending rate. Once a steady state is reached where each flow has a fair share of the link and there is no delay, if one flow tries to change their sending rate, this

Algorithm 2: Offline Staged Training of Pareto.

INPUT: Replay Buffer: B, set of environments: $\mathbf{\Omega}_{\text{bootstrap}}$, $\mathbf{\Omega}_{\text{advance}}$, $\mathbf{\Omega}_{\text{fairness}}$, probability distribution of sampling in B: P

▷ Bootstrapping stage

1: Sample randomly an environment from $\mathbf{\Omega}_{\text{bootstrap}}$
2: TD3B, N, $P \leftarrow \mathcal{U}_{[}0, \frac{1}{N}$
3: Repeat from Step *1* till stage is over

▷ Advancing stage

4: Increase size of B and freeze old samples
5: $P \leftarrow \mathcal{U}_{[}0, \frac{1}{N}]$
6: Sample randomly an environment from $\mathbf{\Omega}_{\text{advance}}$
7: TD3 (B, N, P)
8: Repeat from Step *6* till stage is over

▷ Fairness stage

9: Increase size of B, freeze old samples and **share** B among all sender
10: Fairness (B, N, P)
11: **Function** Fairness (B, N, P)
12: Sample randomly an environment from $\mathbf{\Omega}_{\text{fairness}}$
13: Each sender of the two plays an episode sharing one network link
14: One sender share its new experience tuples with the other head sender to update the shared buffer B
15: Run TD3B, N, P from Algorithm 3 in head sender
16: Share the updated model parameters ($\phi_{1,2}, \phi'_{1,2}, \theta, \theta'$) with other senders
17: Repeat from Step *12* until stage is over

will cause a delay or decrease in the delivery rate in the link and one or more flows will get a negative reward or penalty. In short, sharing experience tuples helped the model attain fairness. Algorithm 2 summarizes the steps of our offline learning process in Pareto.

To empirically study the effect of staged training, we train one model over steady and dynamic bandwidth environments in one single stage "Pareto-No-Staged" and compare it with a model trained over bootstrapping and advancing stage "Pareto-Staged." All hyperparameters, such as buffer size, and design decisions were kept the same for a fair comparison.

Table I compares stage-trained and non-stage-trained models using the performance metrics of average throughput, 95th percentile delay,[4] and average loss rate in three different scenarios: fixed bandwidth of 50 Mbps and changing bandwidth every 5 seconds from 10 to 20 Mbps and 10 to 30 Mbps. All environments have a fixed propagation RTT of 90 ms.

Although the average throughput of the stage-trained "Pareto-Staged" is either same or lower than "Pareto-No-Staged," the loss rate and 95th percentile one-way delay are always lower or almost the same. The "Pareto-No-Staged" is particularly aggressive in increasing its sending rate to find a higher throughput, which explains the high loss rate. As opposed to staged-training, in one stage, "Pareto-No-Staged" was confused between aggressively increasing the sending rate (as bandwidth changes often) and slowly increasing sending rate (as bandwidth does not change).

[4] Whenever we refer to 95th percentile delay or 95th percentile one-way delay, we mean half of the RTT.

TABLE I
COMPARING THE PERFORMANCE OF NON-STAGE-TRAINED VS. STAGE-TRAINED MODELS, AND THE PERFORMANCE OF THE STAGE-TRAINED MODEL ON RETAINED OLD EXPERIENCES VS. STAGE-TRAINED MODEL ON NEW EXPERIENCES ONLY. THE TESTING ENVIRONMENTS ARE (I) A FIXED 50 MBPS BANDWIDTH LINK, DYNAMIC BANDWIDTH LINK CHANGING FROM (II) 10 TO 20 MBPS AND (III) 10 TO 30 MBPS EVERY 5 SECONDS WITH FIXED RTT OF 90 MS

Metric	50 Mbps	10&20 Mbps	10&30 Mbps
Non-staged training — Single buffer (Pareto-No-Staged)			
Avg.s throughput(Mbps)	44.95	13.07	17.61
95th %ile delay(ms)	117.13	404.90	405.49
Avg. loss rate(%)	17.12%	18.25%	24.52%
Staged training — Did not retain old experiences (Pareto-Staged)			
Avg. throughput(Mbps)	45.64	13.05	14.82
95th %ile delay(ms)	113.26	402.40	405.16
Avg. loss rate(%)	1.92%	7.66%	10.36%
Staged training — Retained old experiences (Pareto-Advance-No-Prio)			
Avg. throughput(Mbps)	43.94	13.01	14.40
95th %ile delay(ms)	92.91	392.25	405.03
Avg. loss rate(%)	0.54%	4.38%	7.88%
Staged training — Prioritized replay buffer (Pareto-Advance)			
Avg. throughput(Mbps)	43.37	12.53	14.48
95th %ile delay(ms)	**79.48**	**336.05**	**405.00**
Avg. loss rate(%)	**0.64%**	**0.87%**	**4.8%**

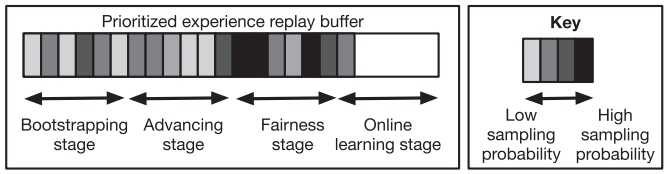


Fig. 1. An illustration of the segmented prioritized experience replay buffer during the online learning stage.

Segmented replay buffer: Retaining old experiences. Tasks learned in the previous stages may be forgotten, because old experiences in the experience replay buffer will be replaced by new experiences from new stages. To fight against catastrophic forgetting and improve stability, we retain old experiences from past stages in the experience replay buffer.

To retain old experiences from previous stages, after each stage, we increase the size of the experience replay buffer by a fixed size. Experiences from the new stage are only added to the new extension, and new experiences will only replace old experiences from the same stage. After convergence, the experience replay buffer will be segmented with experience tuples from all stages retained. To illustrate, in Fig. 1, we show an experience replay buffer during the online learning stage, with old experiences retained from all past stages.

To study the effect of retaining old experiences from previous stages, we carry out the advancing stage with and without retaining old experiences, naming the models "Pareto-Advance-No-Prio" and "Pareto-Staged," respectively. Table I shows the behavior of the models.

When we replace old samples from the bootstrapping stage of training with samples from the advancing stage, we observe a

forgetful behavior. The model trained with retaining old experiences "Pareto-Advance-No-Prio" had three times lower loss rates in the fixed bandwidth scenario at the cost of 5% lower throughput, which shows that without retaining old experiences, the model forgot how to take decisions in steady bandwidth environments. While in the dynamic bandwidth environment, not retaining old experiences also lead to high loss rates at the cost of a marginal gain in throughput. Therefore, retaining old experiences did not hurt the learning process in the new stage, instead it improved the performance of Pareto in old and new tasks.

Prioritized replay buffer for quicker convergence and better performance. When retaining old experiences, sampling from the experience replay buffer will favor old and new experiences equally, although old samples were used in updating the model several times earlier. Hence, to take the best of both worlds, i.e. to not forget old experiences and quickly learn new ones, a prioritized replay buffer allows us to train more on new experiences. However, since prioritized replay buffers were not initially designed for actor-critic policy gradient algorithms, we *adapt* the concept of prioritized replay buffer introduced in [30] for critic-only models to our TD3 actor-critic model.

New experiences generally contribute substantially to the actor and critic losses compared to old experiences. Hence, we can easily identify them in the replay buffer as follows. The actor learns a policy $\pi_{\boldsymbol{\theta}}(a \mid s)$ that maximizes the expected return (or value function) $J(\theta) = \mathbb{E}[Q^{\pi_\theta}(s, \pi_\theta(s))]$, and the critic estimates the action-value function $Q^{\pi}(s,a) = \mathbb{E}_{\pi}\{G_t \mid s_t = s, a_t = a\}$, where $G_t = \sum_{i=0}^{T} \gamma^i r_{t+i}$ and γ is the discount factor. The actor is updated using the deterministic policy gradient algorithm [31] using $\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta Q^{\pi_\theta} \ (s,a)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s)]$, and the critic is updated by minimizing $L(\phi) = \mathbb{E}[(Q_\phi(s,a) - (T_{\pi_{\theta'}} Q_{\phi'})(s,a))^2]$, where T is the Bellman operator $(T_\pi Q)(s,a) = r(s,a) + \gamma \mathbb{E}[Q(s,' \pi(s'))]$. We will use the actor loss $\nabla_\theta J(\theta)$ and critic loss $\delta = \nabla L(\phi)$ of each sample to identify which samples need more frequent sampling.

As in [30], we set the probability of choosing sample i as $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where all p_i, $\forall i : 1 \leq i \leq N$, are set to 1 initially, N is the size of the buffer, and α is a constant. α is set to 0.6 as in [30]. In other words, the initial probability distribution is uniform. To adapt the prioritized replay buffer to actor-critic models, we update p_i every iteration according to (2).

$$p_i \leftarrow \tanh(\delta_i / c + \nabla_\theta J_i(\theta)) + 1 + \zeta \tag{2}$$

where c is a constant set to 10 to scale and equally weigh actor and critic loss values, $\delta = \nabla L_i(\phi)$ is the TD-error of sample i in the mini-batch (which is the critic loss), $\nabla_\theta J_i(\theta)$ is the actor loss of sample i and ζ is a small number, $1e^{-5}$, to prevent $P(i)$ from dropping to zero, if losses are 0. We included $\tanh$ to clip p_i and prevent it from diverging to high values to ensure stability.

Since the estimation of the expected Q-function and value function J relies on uniform sampling, losses of each sample has to be attenuated by a value proportional to its "importance" or probability $P(i)$. This is referred to as weighted importance sampling. The importance of a sample i is weighted by $w_i = \frac{(N \cdot P(i))^\beta}{\max_k w_k}$, where β is a hyperparameter. It is set to 0.4 and grows by 0.001 every iteration till it reaches 1, according to [30]. If the exponent β is 1, the bias introduced by non-uniform probabilities $P(i)$ is fully compensated by weighted importance sampling. In [30], the authors argue that using $\beta = 1$ would aggressively correct the bias. As a result, they gradually correct the bias by increasing β. For stability, weights are normalized by $\frac{1}{\max_i w_i}$, so the update is scaled down.

In simple terms, instead of updating weights of the critic and actor by $\phi \leftarrow \phi - \eta \nabla_\phi L(\phi)$ and $\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$, weights are updated according to $\phi \leftarrow \phi - \eta w \cdot \nabla_\phi L(\phi)$ and $\theta \leftarrow \theta + \eta w \cdot \nabla_\theta J(\theta)$, where w is the vector of importance sampling weights $w = [w_1, w_2, .., w_b]$ and b is the batch size of the mini-batch. Steps for modified TD3 algorithm for prioritized replay buffer are summarized in Algorithm 3.

To illustrate how significant can the performance of Pareto change with the usage of a prioritized replay buffer compared

Algorithm 3: TD3 Algorithm for Prioritize Replay Buffer.

INPUT: Replay Buffer: B, buffer size: N, learning rate: η, policy and critic parameters: θ, ϕ_1 and ϕ_2, target policy and critic parameters: θ', ϕ_1' and ϕ_2', probability distribution of sampling in B: P

1: **function** TD3 (B, N, P)
2: Play an episode
3: **if** reward $r_t < 0$ **then**
4: next action a' is taken by expert
5: Save experience tuple samples in buffer B
6: According to P, sample mini-batch b from B
7: **for** `j`**in** `range`(number of iterations) **do**
8: Compute the target function: $\triangleright\ \epsilon \sim \mathcal{N}(0, \sigma)$

$$y = r + \min_{i=1,2} Q_{\phi_i}(s,' \pi_{\theta'}(s') + \epsilon) \tag{3}$$

9: Compute the gradients of two critics and actor

$$\nabla_{\phi_i} L(\phi_i) = \frac{1}{|b|} \sum_{(s,a,r) \in b} \nabla_{\phi_i} (Q_{\phi_i}(s,a) - y)^2 \tag{4}$$

$$\nabla_\theta J(\theta) = \frac{1}{|b|} \sum_{s \in b} \nabla_\theta Q_{\phi_1}(s,a)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) \tag{5}$$

10: Calculate importance sampling weights w:

$$w_i = \frac{(N \cdot P(i))^\beta}{\max_k w_k} \tag{6}$$

11: Update critics' and actor gradients:

$$\phi_{1,2} \leftarrow \phi_{1,2} - \eta w \nabla_{\phi_{1,2}} L(\phi_{1,2}), \theta \leftarrow \theta + \eta w \nabla_\theta J(\theta) \tag{7}$$

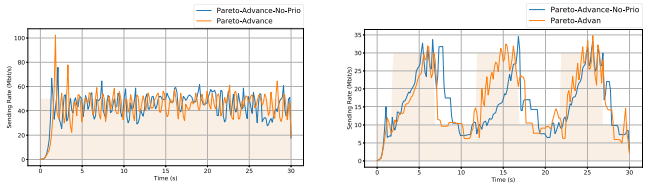
12: Update parameters of target networks:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta,' \ \phi_{1,2}' \leftarrow \tau\phi_{1,2} + (1-\tau)\phi_{1,2}' \tag{8}$$

13: Update probability distribution P:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \text{ where } p_i \leftarrow \tanh(\delta_i / c + \nabla_\theta J_i(\theta)) + 1 + \zeta \tag{9}$$

14: **return** $\phi_{1,2}, \phi_{1,2}', \theta, \theta,' B, N, P$



(a) Fixed Bandwidth of 50 Mbps with one-way propagation delay of 45 ms

(b) Dynamic Bandwidth changing from 10 to 30 Mbps every 5 seconds and one-way propagation delay of 45 ms

Fig. 2. Comparing the performance of Pareto when using a prioritized replay buffer, "Pareto-Advance," and when using the replay buffer without prioritized replay buffer, "Pareto-Advance-No-Prio." Shaded area under the graph is the available bandwidth.

to a regular experience replay buffer, in Fig. 2, we show the difference in performance of Pareto after the advancing stage when a priority replay buffer was ("Pareto-Advance") and was not ("Pareto-Advance-No-Prio") used. In Fig. 2(a), both models performed well and similarly on the fixed bandwidth set to 50 Mbps as it has been observed by both during the bootstrapping stage. While in Fig. 2(b), "Pareto-Advance" exhibits a more responsive behavior to changes in the available bandwidth compared to"Pareto-Advance-No-Prio".

In Table I, we also compare the performance of "Pareto-Advance-No-Prio" and "Pareto-Advance." There is a significant decrease in the loss rate and delay after using prioritized replay buffer, which shows that using prioritized replay buffer helps in converging at a more optimal behavior. In addition, "Pareto-Advance" converged in 40% less iterations than "Pareto-Advance-No-Prio". This is due to the emphasis the algorithm gave to the new samples from the advancing stage.

Expert Demonstrations. Many reinforcement learning algorithms use expert demonstrations in different ways as in [32]. For high dimensional spaces, expert demonstrations improve the sample efficiency of reinforcement learning because the agent will explore only when there is a chance of a valuable learning experience. Therefore, not having expert demonstrations will increase the training time.

As introduced by Eagle [3], we use BBR *expert demonstrations* to speed up convergence of our congestion control model to learn easy tasks quickly. As opposed to Eagle [3], during offline and online training, we use expert demonstrations only when the previous action yielded a reward value below a threshold. Since we only use the expert when our previous action was not rewarding, we expect our expert demonstration to yield a reward higher than our model in this particular step in the training process. Having a weaker expert than BBR may provide a valuable chance to learn better actions, but not most of the time as BBR does. Therefore, convergence may slow down.

Another outcome is that the model may get stuck on a local optimum if all the expert actions were yielding rewards below the current *untrained* model. In short, as long as the expert has better actions than the model at a specific time, the valuable experiences shared with the agent will help speed up training time.

Interested readers are referred to appendix B in supplementary material to see the ratio of expert actions decreases as the model's performance improves. Also, readers interested to know the significance of using expert demonstrations are referred to appendix C.

Algorithm 4: Online Training of Each Model of Pareto on a Real-World Network.

INPUT: Replay Buffer after offline training: B, set of all environments: $\mathbf{\Omega} \subseteq \mathbb{U}$, probability distribution of samples in B after offline learning: P, model parameters after offline training: $\phi_{1,2}, \phi'_{1,2}, \theta, \theta'$

1: Increase size of B and freeze old samples
2: Run TD3 (B, N, P) in Algorithm 3
3: Repeat Step *2* whenever a flow starts

Online training stage. During online learning, the models trained offline are deployed on different senders sharing the network. In our experiments, we assume that in real-world networks the models will observe all kinds of environments, including environments that have not yet been observed during the offline training process. Hence, we train our senders on the previously observed environments, in addition to new environments such as LTE networks and networks shared with a random number of senders.

As opposed to the fairness training stage, where different senders can append their experiences in the same shared prioritized replay buffer, to avoid overloading the network with experience tuples from the sender, during the online learning stage, each sender has their experiences appended to their local prioritized replay buffer. Since different senders train on their local experiences, their model parameters may diverge from one another; however, we believe retaining old experiences in the replay buffer prevents divergence from being disastrous.

Our objective is to learn new experiences while not forgetting old ones, hence we expect to observe improvements in performance — or at least no degradation in the behavior over old and new sets of environments. To achieve this objective, during the online learning stage, we continue to use our prioritized replay buffer produced from the fairness stage.

It is worth noting that most of the prioritized replay buffer of all senders across the network is the same, since the starting point of the prioritized replay buffer was after fairness training stage for all the senders. Retaining old experiences in the replay buffer is essential during online learning. First, this reduces the risk that model parameters on different senders may diverge.

Second, in real-world networks, environments are unpredictable and noisy. For example, the model can observe the same environment for prolonged periods. This raises the risk of the model overfitting and catastrophically forgetting. By retaining old experiences, we guarantee that the model will not forget old experiences and will avoid overfitting to one environment. Expert actions can be used only if the reward in the previous step is minimal, hence guaranteeing a safe-fail behavior.

Algorithm 4 summarizes the details about the online training process. Also, Fig. 3 summarizes the stages of the training process and names the models produced after each stage: *Pareto-Bootstrap*, *Pareto-Advance*, *Pareto-Fair*, and *Pareto-Online*.

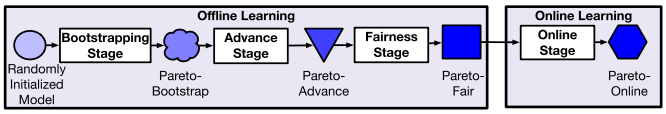


Fig. 3. Four staged training process with names of the models produced after each stage: Pareto-Bootstrap, Pareto-Advance, Pareto-Fair, Pareto-Online.

V. Experimental Results

A. Implementing the Training Framework

As highlighted in [33], recent RL-based congestion control approaches use interfaces that would block the sender while an agent updates gradients or considers taking a new decision/action. This does not translate to real-world networking environments, as these interruptions to sending packets would under-utilize the bandwidth, causing unexpected and inaccurate measurements and observations.

In response, we implemented an RL interface from scratch following the *OpenAI Gym* interface, where the sender will continue sending packets until the agent takes a new decision to change the sending rate of the sender. Therefore, throughout the episode, the sender will not be blocked from sending packets. This is feasible running two different blocks/processes on the sender: the *controller* and *agent*.

In Fig. 4, the *controller* block has three running threads that are: (1) sending packets to the receiver at the prescribed sending rate; (2) receiving acknowledgements from the receiver and making measurements based on acknowledgements received; and (3) sending those measurements to the agent and receiving decisions on the sending rate from the agent at every step. Even if the agent takes time in deciding on a new sending rate, the *controller* block will continue sending packets, and will not be blocked.

The *controller* block is implemented using the C++ programming language since it's running speed is an order of magnitude faster than Python. The receiver side is receiving packets from the *controller* block over UDP. Based on measurements received from the *controller* block, the *agent* block in Fig. 4 is responsible for (1) training Pareto using Algorithm 2, 3 and 4 to make decisions on the sending rate; and (2) sending these decisions to the *controller* block for every step. Since deep learning frameworks, such as `PyTorch`, are based on `Python`, we implement the *agent* block in `Python`. For a reliable communication, we have a TCP connection to exchange measurements and actions between the *agent* and the *controller* block.

Pareto observes different network environments by emulating real-world networks using Mahimahi shells [34]. We emulate fixed RTT, fixed and dynamic bandwidth, and a fixed buffer size of 440 KB. RTT varies between 10 and 90 ms and bandwidth varies between 5 and 100 Mbps. Every episode in our algorithm runs a different Mahimahi shell that emulates a different network environment.

For the bootstrapping stage, every time we start an episode, we randomly pick an environment out of five with fixed bandwidth, either 5, 10, 12, 20, 50, 100 Mbps, and RTT is fixed at 90 ms. In the advancing stage, we pick one out of ten different

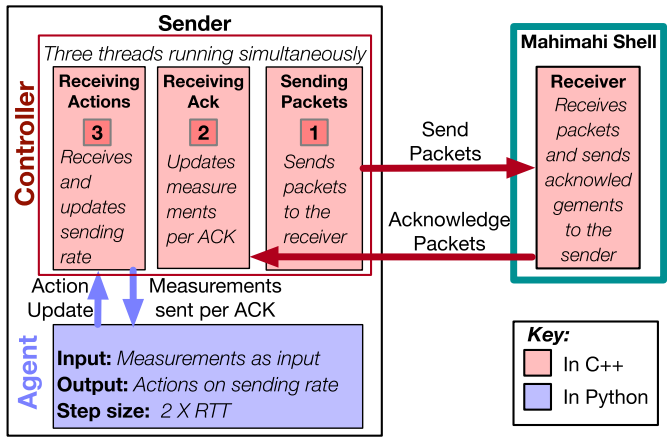


Fig. 4. RL training interface following the *OpenAI Gym* environment.

environments: five of them have fixed bandwidths, as in the bootstrapping stage, and five have dynamic bandwidth that changes suddenly every 5 seconds, and RTT is fixed again to 90 ms. The shaded area in Fig. 2(b) shows an example of dynamic bandwidths, where the available bandwidth changes every 5 seconds. During fairness training, we pick one out of five fixed bandwidth environments, as the ones used in bootstrapping stage.

To evaluate the performance of Pareto, we used the Pantheon experimental testbed [9], which is designed to assess new congestion control algorithms by comparing with existing work. Pantheon has been widely used since its launch [2], [3], [8]. Pantheon uses an emulated network environment using Mahimahi shells, and results from Pantheon are reproducible and accurately reflect real-world results.

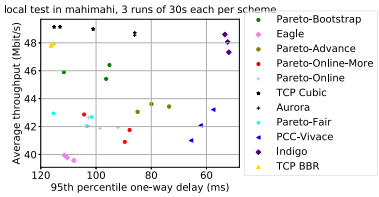
B. Evaluation Metrics

Originally, [35] showed that the operation point where power, defined as $\text{Power} = \frac{\text{Throughput}}{\text{Delay}}$, is maximized is the optimal point for the network and the individual flow. In addition, research papers use throughput and delay jointly as in [9] and in Fig. 5, which plots throughput and delay as we elaborate later, to analyze the performance of an algorithm.

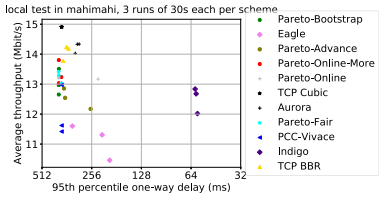
However, according to [12], a congestion control algorithm should be evaluated in terms of the trade-offs between a variety of metrics, such as throughput, delay, loss rate, response time and fairness between competing flows. [12] also states that it is useful to consider throughput, delay and loss rate jointly owing to the fact that they are related. For example, if an algorithm decides to increase its sending beyond network bandwidth, it will observe a high throughput but also a higher delay and packet loss. While if the algorithm decides to avoid delay and packet loss by having a low sending rate and hence throughput, this will under utilize the network link. In essence, judging an algorithm based on several metrics, but not jointly, may not be appropriate.

Accordingly, we introduce a new metric named Ankh's[5] number A_{kh}, that summarizes the trade-offs between

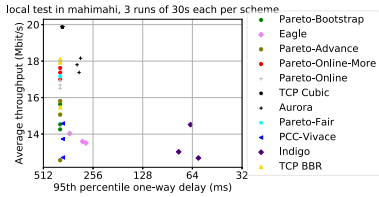
[5] Ankh is an ancient Egyptian hieroglyphic symbol for life, written as †.



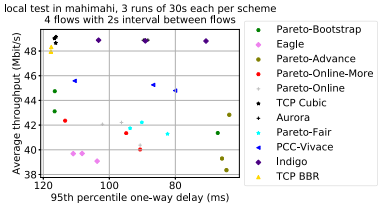
(a) Fixed Bandwidth of 50 Mbps with fixed one-way propagation delay of 45 ms



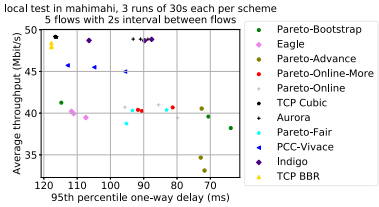
(b) Dynamic Bandwidth changing from 10 to 20 Mbps every 5 seconds and one-way propagation delay of 45 ms



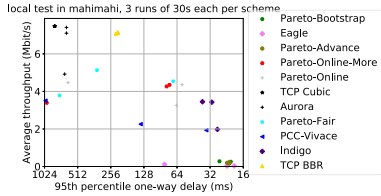
(c) Dynamic Bandwidth changing from 10 to 30 Mbps every 5 seconds and one-way propagation delay 45 ms



(d) Fixed bandwidth of 50 Mbps with one-way propagation delay of 45 ms and 3 other flows of Pareto sharing the link. Flows start at an interval of 2 seconds from each other.



(e) Fixed bandwidth of 50 Mbps with one-way propagation delay of 45 ms and 4 other flows of Pareto sharing the link. Flows start at an interval of 2 seconds from each other.



(f) LTE Verizon Trace with one-way propagation delay of 10 ms

Fig. 5. Comparing the behavior of Pareto after (i) bootstrapping, (ii) advancing, (iii) fairness and (iv) online training over 6 different environments with other congestion control algorithms. Each flow was run for 30 seconds. Each experiment was repeated for 3 times. The scatter plot shows the three runs for each scheme.

throughput, queueing delay and most importantly loss rate. Ankh's number provides a comprehensive reliable summary to help us rank different congestion control algorithms according to their throughput, delay and loss rate trade-off.

Ankh's number can be calculated as in (10),

$$A_{kh} = \frac{(1-U) + |\,(\frac{d_{max}-d_{95\%}}{d_{max}})\,| + l_{avg}}{3} \tag{10}$$

where U is the network utilization ratio $\frac{R_{avg}-C}{C}$, R_{avg} is the average throughput of the flow, C is the average bandwidth of the link, d_{max} is the maximum 95th percentile one-way delay of all the six congestion control algorithms we are comparing Pareto with, $d_{95\%}$ is the 95th percentile one-way delay of the flow and l_{avg} is the average loss rate of the flow in %.

Ankh's number is a unit-less reverse-order metric – the smaller, the better. All the terms in Ankh's number are ratios ranging from 0 to 1, with 0 being the best and 1 being the worst. Therefore, we rank congestion control algorithms from lowest to highest Ankh's number.

Additionally, we use Jain's fairness index [36] to evaluate the fairness between competing flows sharing one link. It is one of the most widely used metrics for assessing the fairness in allocating available resources. Jain's index ranges from 0 to 1, and it is maximized when all flows receive the same resource allocation. It is calculated as $\mathcal{J}(R_1, R_2, \ldots, R_n) = \frac{(\sum_{i=1}^{n} R_i)^2}{n \cdot \sum_{i=1}^{n} R_i^2}$, where R_i is the average throughput of the ith flow and n is the number of flows sharing the link.

C. Pareto Vs. The State-of-The-Art Congestion Control Algorithms

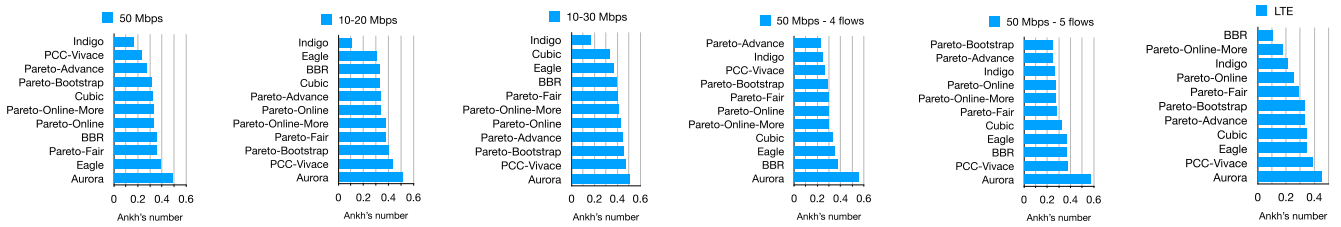
We compare Pareto with the state-of-the-art congestion control algorithms: Aurora [5], BBR [1], Indigo [9], Eagle [3], CUBIC [7], and PCC-Vivace [2].

1) Experiments Performed: We test the models of Pareto produced out of different stages in Pantheon, and compare them with different congestion control algorithms in 9 different environments. Testing environments include network links of (a) fixed bandwidth of 50 Mbps, dynamic bandwidth changing from (b) 10 to 20 Mbps, (c) 10 to 30 Mbps and vice versa every 5 seconds, fixed bandwidth of 50 Mbps with (d) two, (e) three, (f) four and (g) five flows sharing the link, and finally LTE environment while (h) standing and (i) driving. The propagation RTT in all the testing environments is set to 90 ms, except for LTE, it is 20 ms. Each congestion control algorithm/model is tested for 3 times on an environment for 30 seconds each time.

First, Fig. 5 plots the average throughput and 95th percentile one-way delay of each run in 6 out of 9 experiments. Fig. 5 compares different models of Pareto produced from different stages with different congestion control algorithms. All graphs in Fig. 5 show the 95th percentile one-way delay axis reversed, and hence better congestion control algorithms appear to the top and right part of the graph.

Second, for each congestion control algorithm, we calculate the average (i) throughput, (ii) loss rate and (iii) 95th percentile one-way delay of three experiment runs to calculate the Ankh's number for each congestion control algorithm to plot Fig. 6. The Ankh's number of different models produced at different stages in Pareto's training and other congestion control algorithms are ranked in Fig. 6, where the lower the Ankh's number, the better trade-off the algorithm has is in terms of jointly optimizing throughput, delay and loss rate. Fig. 6 shows the experiments on the same 6 environments as Fig. 5.

Third, in Fig. 7(d), we plot the Jain's fairness index for different congestion control schemes, when 2, 3, 4 or 5 flows share a link. The congestion control algorithms are ranked based on the average Jain's fairness index over 4 shared



(a) 50 Mbps fixed bandwidth and one-way propagation delay of 45 ms

(b) Dynamic Bandwidth changing from 10 to 20 Mbps every 5 seconds and one-way propagation delay of 45 ms

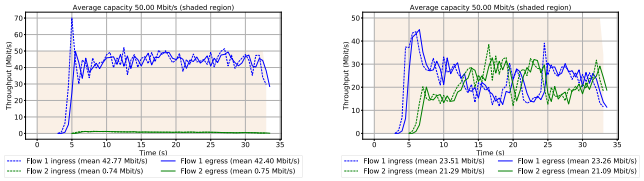
(c) Dynamic Bandwidth changing from 10 to 30 Mbps every 5 seconds and one-way propagation delay of 45 ms

(d) 50 Mbps fixed bandwidth, one-way propagation delay of 45 ms and 4 flows sharing the link.

(e) 50 Mbps fixed bandwidth, one-way propagation delay of 45 ms and 5 flows sharing the link.

(f) LTE Verizon Trace with one-way propagation delay of 10 ms

Fig. 6. Comparing the behavior of Pareto with different congestion control algorithm in terms of the Ankh's number over 6 different network environments.



(a) The behavior of the Pareto-Advance model when two flows share the link

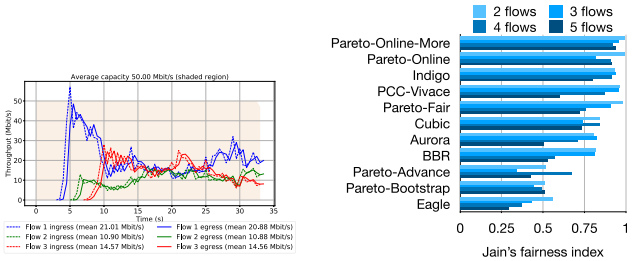


Fig. 7. The behavior of Pareto before and after training on fairness and Jain's fairness index compared to other congestion control algorithms.

network experiments, where the top most scheme has the highest average Jain's fairness index.

Finally, to get a better perspective of the fairness depicted by congestion control algorithms, in Fig. 8 and Fig. 10, we plot the throughput of different congestion control algorithms vs. time when four and five flows share a 50 Mbps bandwidth link, respectively. The flows start 2 seconds apart from each other. In Fig. 10, we additionally test the responsiveness of the algorithms by departing flows after 30 seconds from the experiment. The flows depart 10 seconds after the previous flow departed as depicted in Fig. 10.

2) Ankh's Number and Jain's Fairness: Aurora and Eagle. In Fig. 6, Pareto-Fair and Pareto-Online have 20% to 52% lower Ankh's numbers, respectively, than Aurora in all environments. We observe Aurora in the top middle-left of all figures in Fig. 5. This shows that Aurora has a high throughput; however, due to high loss rate, we observe Aurora ranking low in figures in Fig. 6. This shows how important it is to take loss rates into account when evaluating an algorithm.

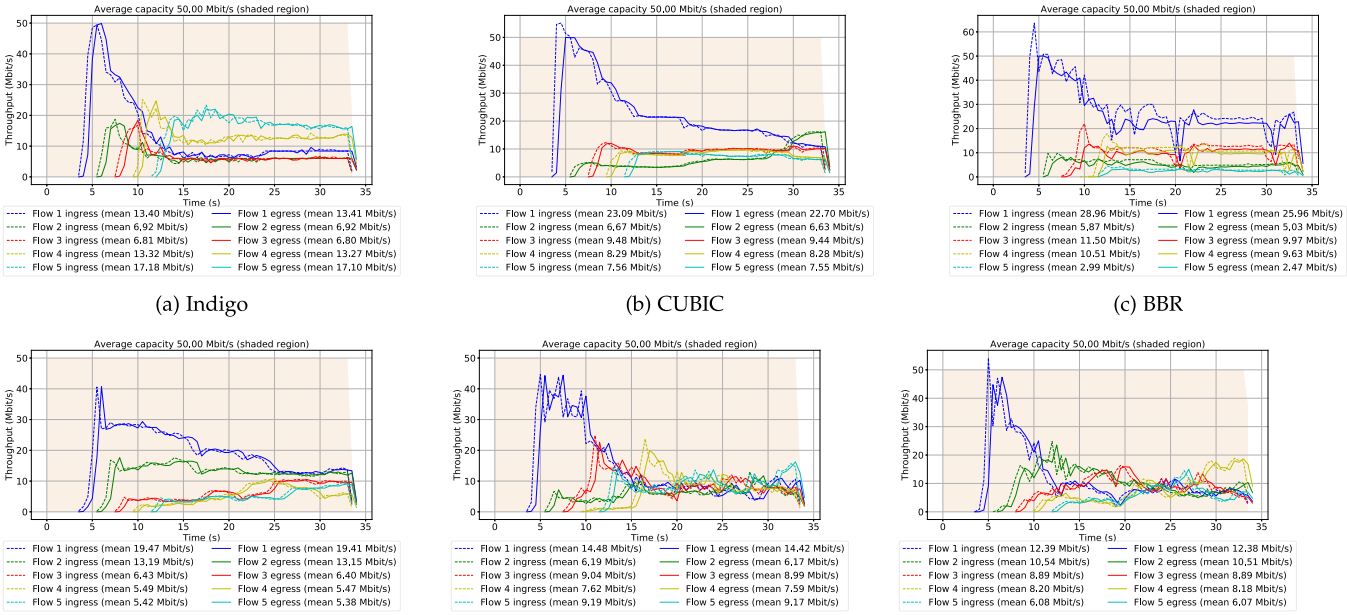
Compared to Eagle, Pareto-Fair and Pareto-Online have lower Ankh's number in *most* environments. For example, in the LTE environment, Pareto-Fair and Pareto-Online have 16% and 26% lower Ankh's number, respectively, compared to Eagle. In fixed (shared or unshared) bandwidth environments, Pareto-Fair and Pareto-Online have 7% to 25% lower Ankh's number, respectively, compared to Eagle. In contrast, in dynamic environments, where the bandwidth changes from 10 to 20 Mbps and 10 to 30 Mbps, Eagle performs better than Pareto-Fair with 26% and 10% lower Ankh's number, respectively.

Fig. 7(d) shows that Pareto-Fair and Pareto-Online are all better with respect to fairness than Eagle and Aurora, showing higher Jain's fairness indices. Pareto-Fair surpasses Aurora by 19%, and Eagle by 119%. In summary, we can safely conclude that in our experiments Pareto out-performed state-of-the-art DRL-based algorithms in terms of fairness, and achieving a better trade-off between throughput, latency and loss rate by at most 52%.

BBR. On the contrary, it is difficult to claim an absolute performance superiority over other congestion control schemes. For example, BBR is the best performer in LTE environments as shown in Fig. 6(f), and Pareto-Online is the fourth best. In the fixed bandwidth setting where four or five flows share the link, BBR has a higher Ankh's number by at least 27% compared to Pareto-Online as shown in Fig. 6(d) and (e). In other environments, BBR and Pareto-Online have comparable behavior. Consequently, it is difficult to claim if Pareto is superior to BBR.

Attempting to reach a consensus on fair comparisons, we rank the overall performance of different congestion control schemes by averaging their Ankh's numbers for 9 different environments in Fig. 9(a). The lower the average Ankh's number, the better the overall performance is. Additionally, we average Jain's fairness index from the experiments with 2, 3, 4 and 5 flows in Fig. 9(b). The algorithm with the highest index is the fairest algorithm overall in our experiments. We show the standard error on the bars of both figures.

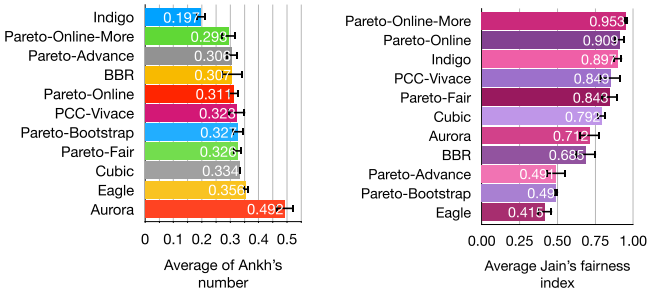
Fig. 9(a) shows that BBR and Pareto-Online have similar average Ankh's number within 1%. In terms of fairness, Fig. 9 (b) shows Pareto-Online having 33% higher Jain's fairness index than BBR. In Fig. 8(c), we observe that BBR stabilizes to an unfair resource allocation to flows with the first flow acquiring most of the capacity. A similar pattern is also observed in Fig. 10(c), where the first flow has the most bandwidth. In addition, BBR's fairness drops as the number of flows grow as observed in Fig. 7(d). Overall, BBR performs better than Pareto-Online on dynamic environments such as LTE, but Pareto-Online always outperforms BBR in fixed bandwidth environments and in terms of fairness.



(a) Indigo (b) CUBIC (c) BBR

(d) PCC-Vivace (e) Pareto-Online (f) Pareto-Online-More

Fig. 8. Fairness of different schemes when five flows starting 2 seconds apart share a 50 Mbps fixed bandwidth link.



(a) Average Ankh's number for each congestion control algorithm over 9 environments

(b) Average Jain's fairness index for each congestion control algorithm over 2, 3, 4 and 5 flows environments

Fig. 9. Average Ankh's number and Jain's fairness index for each congestion control algorithm.

PCC-Vivace. In Fig. 9(a), we observe that the average Ankh's number of Pareto-Online is 0.311, which is close to 0.323 of PCC-Vivace. PCC-Vivace behaves very well in fixed environments with 50% lower Ankh's numbers than Pareto-Online as shown in Fig. 6(a). Conversely, it behaves poorly in dynamic environments with 22%, 8%, 27% and 33% higher Ankh's numbers as shown in Figs. 6(b), 6(c), 6(e) and 6(f), respectively. It seems when the environment is dynamic or has a higher number of flows such as 5, the known drawback, i.e. slow convergence property, of PCC-Vivace emerges. In conclusion, PCC-Vivace does not generalize well to a wide array of environments.

In terms of fairness, Pareto-Online is having comparable fairness to PCC-Vivace with 7% higher Jain's fairness index as observed in Fig. 9(b). This is due to the slow convergence property of PCC-Vivace depicted in Fig. 8(d), and the steady-state behavior is not absolutely fair. A slow response is also portrayed in Fig. 10(d) when flows depart. Additionally, the fairness performance of PCC-Vivace decays when the number of flows increases as shown in Fig. 7(c).

CUBIC. The average Ankh's number of CUBIC compared with Pareto-Online and Pareto-Fair is similar within 7%. In fixed bandwidth environments as in Fig. 6(a), the Ankh's number is similar within 9%. However, for dynamic bandwidth environment where bandwidth changes from 10 to 30 Mbps, CUBIC performs better by 22% compared to Pareto-Online as in Fig. 6(c). While, in LTE environments and shared links, CUBIC has 15% to 30% higher Ankh's number compared to Pareto-Online and Pareto-Fair.

In terms of fairness, the Jain's fairness index of Pareto-Fair and Pareto-Online is 7% and 14% higher than CUBIC, respectively. Also, Fig. 8(b) shows that CUBIC is slow in converging to a fair behavior. When flows depart in Fig. 10 (b), the responsiveness of CUBIC to attain a fair resource allocation is slow. Pareto clearly outperforms CUBIC in terms of fairness and overall generalized behavior in a wide set of environments.

Indigo. Indigo has a 36% lower average Ankh's number in Fig. 9(a), and it has a comparable 2% lower Jain's fairness index in Fig. 9(b) compared to Pareto-Online. However, in Fig. 7(d), we observe that as the number of flows grows, Jain's fairness index drops, which shows that Indigo does not scale well with several flows sharing a link. This is clearly observed in Fig. 8(a) where Indigo does not reach a fair steady-state behavior.

Regarding responsiveness to departing flows, Indigo depicts almost no response by marginally increasing the sending rate of existing flows when other flows depart in Fig. 10(a), as opposed to Pareto-Online. It is likely that Indigo is not responsive as it was not trained on this scenario during offline learning.

In addition, Indigo is an offline learned model and will not be able to adapt to newly seen environments that are substantially different from the environments it was trained on. As we

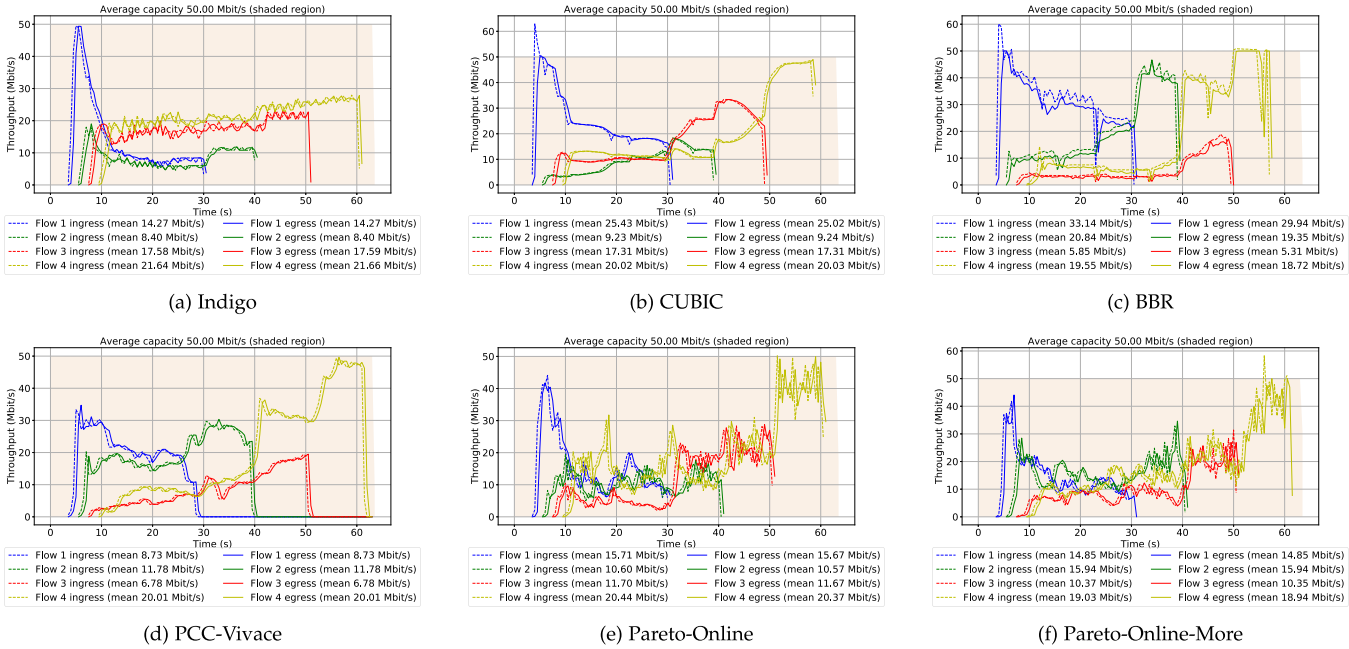


Fig. 10. Fairness of different schemes when four flows starting 2 seconds apart share a 50 Mbps fixed bandwidth link. The flows start departing after 30 seconds at intervals of 10 seconds.

show in our next section, Pareto can adapt online and hence outperforms Indigo with respect to both online adaptability and fairness. In summary, in terms of Ankh's number, Indigo, Pareto and BBR are the best congestion control algorithms; and in terms of Jain's fairness, Pareto, Indigo and PCC-Vivace are the best.

3) Pareto's Online Adaptability: Pareto continues to learn online after the last stage of offline learning. When Pareto-Fair is trained online for 30 and 100 minutes, it produces Pareto-Online and Pareto-Online-More, respectively. Online training is over a random combination of shared, fixed and dynamic bandwidth environments, including LTE.

We observe improvement in performance in newly seen environments and a slightly better or similar behavior in previously observed environments. The former is due to online learning, and the latter is attributed to the large segmented experience replay buffer with a wide variety of old retained experiences and new ones, which increased stability and prevented catastrophic forgetting.

Fixed unshared bandwidth links. In terms of fixed bandwidth environments, Pareto-Online and Pareto-Online-More in Fig. 5(a) have lower 95th percentile one-way delay and a similar throughput compared to Pareto-Fair. Also, in Table II, which summarizes the metrics used to calculate the Ankh's number and compares the performance of different Pareto's models with its teacher — BBR, we observe a decrease in loss rate and delay for the 50 Mbps bandwidth environment, and hence there is a 6% and 9% decrease in Ankh's number for Pareto-Online and Pareto-Online-More, respectively.

Dynamic unshared bandwidth links. In dynamic bandwidth environments, where the bandwidth changes from 10 to 20 Mbps and 10 to 30 Mbps every 5 seconds, the behavior of Pareto-Online and Pareto-Online-More is similar to the behavior of Pareto-Fair with small changes to loss rate, average throughput and 95th percentile one-way delay. There is less than 1% change in Ankh's number from Pareto-Fair to Pareto-Online-More in Table II. Hence, Pareto didn't benefit or get harmed from online adaptation on previously observed dynamic environments.

Fixed shared bandwidth links. When two flows share a fixed bandwidth link, the behavior of Pareto improves where Pareto-Online has 50% lower loss rate than Pareto-Fair in Table II. This significant decrease in loss rate is reflected in a decrease in Ankh's number.

On the other hand, when three flows share the network link, which the model did not observe during offline learning, Pareto seems to improve its delay and loss rate over time as it learns online. Table II shows an overall 43% and 12% decrease in loss rate and delay, respectively, from Pareto-Fair to Pareto-Online-More, which lead to a decrease in Ankh's number by 10%.

In addition, in terms of fairness, Pareto benefitted from online learning as well. Fig. 7(d) shows an increase in Jain's fairness index from Pareto-Fair to Pareto-Online to Pareto-Online-More when 2-5 flows share the network.

LTE. Finally, in the LTE environment, another significant improvement is observed in the performance of Pareto from Pareto-Fair to Pareto-Online and Pareto-Online-More. Pareto has a 40% decrease in Ankh's number from Pareto-Fair to Pareto-Online-More as shown in Table II. This is mainly a result of 80% and 70% decrease in the 95th percentile delay and loss rate, respectively, from Pareto-Fair to Pareto-Online-More. Since LTE environment was not observed in offline learning, during online learning Pareto got the chance to train and gain more experience in the LTE environment.

Overall, the behavior of Pareto improves or remains the same in environments observed earlier during the offline learning stage; however, in newly seen environments, where it

TABLE II
COMPARING THE PERFORMANCE OF PARETO AT DIFFERENT TRAINING STAGES IN OFFLINE AND ONLINE LEARNING WITH ITS TEACHER – BBR OVER A WIDE SET OF ENVIRONMENTS. PERFORMANCE METRICS USED FOR COMPARISON ARE AVERAGE THROUGHPUT, 95TH PERCENTILE DELAY, AVERAGE LOSS RATE AND ANKH'S NUMBER

Model	Average throughput (Mbps)	95th %tile delay (ms)	Average loss rate(%)	Ankh's no.
Fixed bandwidth 50 Mbps				
Pareto-Bootstrap	45.91	101.06	0.79	0.320
Pareto-Advance	43.37	**79.48**	**0.64**	**0.275**
Pareto-Fair	42.56	106.78	1.08	0.360
Pareto-Online	42.15	97.79	0.81	0.336
Pareto-Online-More	41.84	93.93	0.96	0.327
BBR	**47.87**	116.00	1.70	0.353
Dynamic bandwidth 10 & 20 Mbps				
Pareto-Bootstrap	13.04	404.97	7.97	0.403
Pareto-Advance	12.52	**336.05**	**0.87**	0.335
Pareto-Fair	13.24	400.595	3.03	0.379
Pareto-Online	13.20	348.136	2.95	0.336
Pareto-Online-More	13.36	400.43	2.81	0.375
BBR	**14.06**	366.18	0.90	**0.325**
Dynamic bandwidth 10 & 30 Mbps				
Pareto-Bootstrap	14.81	405.15	9.50	0.452
Pareto-Advance	14.48	404.99	4.82	0.441
Pareto-Fair	**17.44**	405.16	8.32	0.404
Pareto-Online	16.72	405.14	11.21	0.425
Pareto-Online-More	17.34	404.98	8.92	0.407
BBR	17.17	**404.74**	**3.78**	**0.393**
LTE				
Pareto-Bootstrap	0.22	22.91	**0**	0.332
Pareto-Advance	0.20	**21.78**	0.03	0.333
Pareto-Fair	4.49	386.62	0.41	0.291
Pareto-Online	4.03	249.68	0.70	0.258
Pareto-Online-More	4.30	77.97	0.12	0.175
BBR	**7.12**	223.76	0.20	**0.109**
Fixed 50 Mbps shared by 2 flows				
Pareto-Bootstrap	45.63	98.76	0.84	0.312
Pareto-Advance	42.7	**86.47**	0.60	**0.296**
Pareto-Fair	44.14	98.07	0.80	0.320
Pareto-Online	43.84	94.87	**0.40**	0.312
Pareto-Online-More	43.91	90.63	0.41	0.299
BBR	**49.07**	117.39	6.95	0.363
Fixed 50 Mbps shared by 3 flows				
Pareto-Bootstrap	45.48	90.13	0.62	0.291
Pareto-Advance	42.63	**79.31**	**0.31**	**0.278**
Pareto-Fair	46.10	100.95	0.77	0.318
Pareto-Online	45.26	98.96	1.44	0.320
Pareto-Online-More	45.22	88.61	0.33	0.287
BBR	**50.00**	117.61	11.24	0.373

was behaving well after the offline learning stage, it continued to improve in terms of several metrics such as throughput, loss rate, latency and fairness. In Fig. 9(a), the overall improvement in Ankh's number is 11%, and in Fig. 9(b), the overall improvement in Jain's fairness index is 13%.

Despite not observing divergence in online learning in our experiments, we are still using expert actions whenever a reward goes below a threshold. If it happens more, automatically we will have the expert as a backup, which guarantees a fail-safe learning experience.

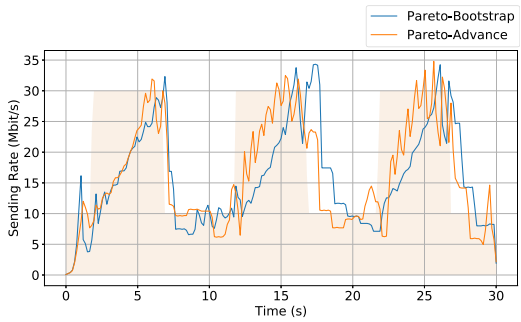


Fig. 11. Comparing the dynamics of the decisions taken by Pareto after the bootstrapping stage vs. after the advancing stage.

Did online learning help Pareto surpass other congestion control algorithms? Compared to the state-of-the-art algorithms, Pareto-Online-More surpassed BBR and PCC-Vivace in terms of the average Ankh's number by 5% and 10%, respectively. Overall, Pareto-Online-More did not surpass Indigo's Ankh's number; however, in terms of performance on individual environments, we observe Pareto-Online-More surpassing Indigo in LTE environment. While initially Pareto-Fair was having 35% higher Ankh's number than Indigo, Pareto improved its performance by 40% to surpass Indigo by 19% in LTE. Neither Indigo nor Pareto were trained offline over LTE, but Pareto was capable of surpassing Indigo with online training. Pareto also surpasses the Jain's fairness index of PCC-Vivace and Indigo by 11% and 6%, respectively.

Therefore, online adaptability gave Pareto an edge over Indigo, which is offline trained, in LTE and shared environments. A fair conclusion is that Indigo and Pareto-Online-More — with its ability to adapt online — are the top two congestion control algorithms in terms of Ankh's number and Jain's fairness index. We believe Pareto is scoring the best when taking into consideration (i) fairness, (ii) trade-off between throughput, delay and loss rate, (iii) online adaptability, and (iv) generalized performance over a wide array of environments.

D. Analysis of Pareto's Progression

Pareto has a generalized behavior over a wide set of environments, and as it learns from new environments without forgetting old experiences. This is owed to the staged training process, which trains on a wide set of environments and retains old experiences. We show in this section how Pareto gains fairness behavior, generalization and suffers no catastrophic forgetting.

Pareto-Bootstrap. In the fixed bandwidth environment of 50 Mbps, Pareto-Bootstrap performed better than its teacher – BBR. Pareto-Bootstrap is having almost 54% less loss rate compared to BBR in Table II. Since Pareto-Bootstrap is not trained over dynamic bandwidth environments, as observed in Figs. 6(b) and (c), Pareto-Bootstrap performs worse than BBR and other Pareto models trained on these dynamic environments as it has a higher Ankh's number.

Pareto-Advance. Pareto-Advance gained experience to better utilize dynamic bandwidth environments. Fig. 11 shows

the dynamics of decisions taken by Pareto-Bootstrap and Pareto-Advance in the 10 to 30 Mbps bandwidth environment. We observe in Fig. 11 that Pareto-Advance has a faster convergence rate to changing bandwidths compared to Pareto-Bootstrap. This improves the utilization of the network link without resulting in delays and losses.

In Table II, we observe Pareto-Advance has $\times 9$ and $\times 2$ less loss rate than Pareto-Bootstrap in the 10 to 20 Mbps and 10 to 30 Mbps environments, respectively. This is also reflected in the lower Ankh's numbers of Pareto-Advance compared to Pareto-Bootstrap in Figs. 6(c) and (c).

Both Pareto-Bootstrap and Pareto-Advance do not exhibit fair behavior on shared network links, since they were not trained for fairness. We observe low Jain's fairness indices in Figs. 7(d) and 7(b).

Pareto-Fair. In Figs. 7(a)–Fig. 7(c), we show a summary of the behavior of Pareto before and after training for fairness. The experiment runs for 30 seconds, and the next flow commences after 2 seconds after the first flow. Fig. 7(a) shows Pareto-Advance not allowing for a fair-allocation of bandwidth.

After training for fairness, we observe in Fig. 7(b) the decrease in the sending rate of the old flow and increase in sending rate of the new flow as they reach a settling average throughput of 22 Mbps, which is approximately half of what Pareto utilizes if one flow uses the link. A similar fair behavior is observed when 3 flows share the bandwidth link in Fig. 7(c). In Fig. 7(d), Pareto-Fair has at least 125% improvement in Jain's fairness index compared to Pareto-Bootstrap.

Apart from the fairness of Pareto, Pareto-Fair also improved the network utilization compared to Pareto-Advance when considering all environments, except the fixed bandwidth environment. For example, in Table II, Pareto-Fair shows an increased average throughput by 20% in the dynamic environment where bandwidth changes from 10 to 30 Mbps.

However, the increase in network utilization of Pareto-Fair sometimes leads to higher delay or loss rate in some environments, leading to an increase in Ankh's number compared to Pareto-Advance. For example, in fixed bandwidth environment in Fig. 6(a), Pareto-Advance has lower Ankh's number compared to Pareto-Fair.

This was at the cost of generalization to a wider set of environments. Pareto-Fair additionally performed well on LTE environment, which it was not trained on. There is an improved average throughput by more than $22\times$ as illustrated in Table II. Ankh's number of Pareto-Fair is 14% lower than Pareto-Advance. Our understanding is that the experiences gained during fairness training added experiences that made Pareto more knowledgeable to act better in environments it did not observe before.

Despite Pareto not observing old environments during fairness training, degradation in its performance on old environments is minor. This is owed to the prioritized replay buffer that retained old experiences for training. This shows that Pareto is avoiding catastrophic forgetting, and generalizing to a wide set of environments.

The success of Pareto is attributed to the (i) staged training process, which allowed training on a wide variety of environments, (ii) training for fairness, (iii) online training, which improves the performance of Pareto in terms of Ankh's number and in terms of fairness after offline training, and finally (iv) training on old retained experiences, which help in avoiding catastrophic forgetting and reproducing stability in performance.

VI. Related Work

More than three decades of active research in congestion control has yielded a wide variety of congestion control schemes, including a variety of TCP variants.

Heuristics. Among early TCP variants are TCP Reno [37] and TCP NewReno [38]. These are AIMD-based algorithms that use losses as congestion signals. Later, CUBIC [7] replaced linear incremental changes to congestion window size to follow a cubic function. Since loss-based congestion signals lead to higher delays in the network, TCP Illinois [39] and BBR [1] use delay and loss as congestion signals. Other works illustrate limitations of BBR, such as high packet loss and unfairness in certain situations [40].

Apart from the hard-wired mappings in classic heuristics, other heuristics focus on the nature of the environment. For example, Sprout [41] targeted cellular networks, while DCTCP [42] and LDP [43] focused on data center networks. The main issue with heuristics is that the mappings between network events and control responses is fixed; therefore, if the network does not behave according to the prior assumptions used to design the control responses, these algorithms may not perform well.

Learning based. Besides heuristics, several learning-based approaches have been studied too. The main benefit of learning-based approaches is to avoid hard-wired mappings between events and control responses that are based on certain assumptions on the network.

Online learning. To avoid hard-wired mappings between events and control responses, PCC [6] and PCC-Vivace [2] work on live evidence from the network environment to try to learn online the behavior of the network. However, as they try different actions and move towards sending rates that give them a higher utility, their critical limitation is the high convergence time to reach the optimum throughput and delay within a flow.

Offline learning. Indigo [9] trains a LSTM using imitation learning, where an oracle serves as an expert to label actions to observations. Remy [8] is another offline optimization framework for congestion control. Even though Indigo and Remy has a near-optimal performance on environments they were trained on, the apparent limitation is that their behavior does not adapt when they see newly seen environments.

RL approaches. The sequential decision-making process in congestion control fits very well with reinforcement learning objectives [4]. Aurora [5], Eagle [3] and Orca [4] used DRL algorithms to train either an LSTM model or simple neural network using off-the-shelf RL algorithms.

Aurora and Eagle did not perform well in our experiments because their design did not allow them to(i) generalize well to a wide variety of environments and (ii) train for fairness. While actions of Orca were used to improve CUBIC's performance, yet both (Orca and CUBIC) are offline learned

algorithms, and hence won't be able to perform well on newly seen networks.

QTCP [44] and SmartCC [45] are examples of RL approaches not using deep neural networks. However, as mentioned earlier, the training time would increase significantly if larger history size is to be incorporated.

Network-assisted. Some other algorithms require feedback from entities on a path. For example, XCP [46] uses explicit feedback from the network, where the routers inform the senders about the degree of congestion at the bottleneck. However, this requires new of routers/switches and collaboration of different entities on a path, which may not be feasible for the Internet.

Stability analysis. There are various papers analyzing the stability and fairness in heterogeneous networks, such as the Internet and wireless networks [47]–[49]. These works use Lyapunov theory-based approaches for stability analysis. Conditions for stability of heterogeneous-flow networks are obtained to efficiently and fairly support multiple applications. However, these methods require information about the network that may be hidden to clients, for example, queue-length at a buffer. Hence, these methods require the replacement of network switches, which is not practical for the Internet. Our DRL method does not require replacement of switches or routers, and uses only data that can be measured from receiver acknowledgements.

Fairness in multi-agent reinforcement learning (MARL). As raised by Aurora [5], fairness in congestion control powered by RL is challenging as the agent is supposed to take actions fairly guided only by the reward function. Several works work on providing fairness in MARL environments; however, they require communication between agents such as CommNet [50] and ATOC [51].

VII. Conclusion

In this paper, we present *Pareto*, a new congestion control protocol fueled by staged DRL training process, fairness training and online learning. Our staged training technique allowed Pareto to gain more experiences compared to other algorithms powered by DRL, hence Pareto performs well in a wide variety of environments. During fairness training, Pareto learned to excel in shared networks. Pareto has up to 40% and 20% better fairness compared to BBR and CUBIC respectively, and 129% and 33% better fairness compared to Aurora and Eagle, respectively, which are fueled by DRL too.

Pareto's online learning algorithm ensures adaptation to newly seen environments. The trade-off between throughput, latency and loss rate improved by 40% after online training in LTE networks. In addition, fairness of Pareto improved by $\approx$ 13% after online adaptation. Overall, Pareto is capable of surpassing the state-of-the-art congestion control algorithms by improving the trade-off between throughput, latency and loss rate, fairness towards flows sharing the same link, generalization and online adaptability.

In our future work, we aim to study the possibility of deriving theoretical guarantees on performance boundaries when designing learning-based congestion control algorithms.

References

[1] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *Commun. ACM*, vol. 60, no. 2, pp. 58–66, Feb. 2017.

[2] M. Dong, T. Meng, D. Zarchy, E. Arslan, and Y. Gilad, "PCC vivace: Online-learning congestion control," in *Proc. 15th USENIX Symp. Networked Syst. Des. Implementation*, 2018, pp. 343–356.

[3] S. Emara, B. Li, and Y. Chen, "Eagle: Refining congestion control by learning from the experts," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2020, pp. 676–685.

[4] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 632–647.

[5] N. Jaya, N. Rotman, P. B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 3050–3059.

[6] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance," in *Proc. 12th USENIX Symp. Networked Syst. Des. Implementation*, 2015, pp. 395–408.

[7] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[8] K. Winstein and H. Balakrishnan, "TCP ex machina: Computer-generated congestion control," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architecture Protoc. Comput. Commun.*, 2013, pp. 123–134.

[9] F. Y. Yan *et al.*, "Pantheon: The training ground for internet congestion-control research," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 731–743.

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

[11] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Wanna make your TCP scheme great for cellular networks? let machines do it for you!," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 1, pp. 265–279, Jan. 2021.

[12] E. S. Floyd, "Metrics for the evaluation of congestion control mechanisms," 2008. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5166

[13] K. Khetarpal, M. Riemer, I. Rish, and D. Precup, "Towards continual reinforcement learning: A review and perspectives," 2020, *arXiv:2012.13490*.

[14] Y. Li, "Deep reinforcement learning: An overview," 2017, *arXiv: 1701.07274*.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[16] S. D. Whitehead and L.-J. Lin, "Reinforcement learning of non-Markov decision processes," *Artif. Intell.*, vol. 73, pp. 271–306, 1995.

[17] K. Murphy, "A survey of POMDP solution techniques," *Environment*, vol. 2, no. 10, 2000.

[18] L. Lin and T. Mitchell, "Memory approaches to reinforcement learning in non-Markovian domains," Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-CS-92-138, 1992.

[19] D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber, "Solving deep memory POMDPs with recurrent policy gradients," in *Proc. Int. Conf. Artif. Neural Netw.*, 2007, pp. 697–706.

[20] S. W. Hasinoff, "Reinforcement learning for problems with hidden state," University of Toronto, Toronto, Canada, Tech. Rep., 2002.

[21] W. S. Lovejoy, "A survey of algorithmic methods for partially observable Markov decision processes," *Ann. Operations Res.*, vol. 28, no. 1, pp. 47–65, 1991.

[22] A. Doucet, S. Godsill, and C. Andrieu, "On sequential Monte Carlo sampling methods for Bayesian filtering," *Statist. Comput.*, vol. 10, no. 3, pp. 197–208, 2000.

[23] D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution," in *Proc. 32th Adv. Neural Inf. Process. Syst.*, 2018, pp. 123–134.

[24] H. van Hasselt, M. Hessel, and J. Aslanides, "When to use parametric models in reinforcement learning," in *Proc. 32th Adv. Neural Inf. Process. Syst.*, 2019.

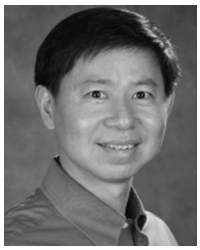
[25] B. Bakker, "Reinforcement learning with LSTM in non-Markovian tasks with long-term dependencies," Leiden University, Tech. rep., 2001.

[26] B. Bakker, "Reinforcement learning with long short-term memory," in *Proc. 14th Adv. Neural Inf. Process. Syst.*, 2001, pp. 1475–1482.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[28] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 1587–1596.
[29] E. Bengio, J. Pineau, and D. Precup, "Interference and generalization in temporal difference learning," in *Proc. 37th Int. Conf. Mach. Learn.*, 2020, pp. 767–777.
[30] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *Proc. 4th Int. Conf. Learn. Representations*, 2016.
[31] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller , "Deterministic policy gradient algorithms," in *Proc. 31th Int. Conf. Mach. Learn.*, 2014, pp. 387–395.
[32] M. Jing *et al.*, "Reinforcement learning from imperfect demonstrations under soft expert guidance," in *Proc. 34th AAAI Conf. Artif. Intell.*, 2020, pp. 5109–5116.
[33] V. Sivakumar *et al.*, "MVFST-RL: An asynchronous rl framework for congestion control with delayed actions," in *Proc. NeurIPS Workshop Mach. Learn. Syst.*, 2019.
[34] R. Netravali, A. Sivaraman, S. Das, and A. Goyal, "Mahimahi: Accurate record-and-replay for HTTP," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 417–429.
[35] R. Gail and L. Kleinrock, "An invariant property of computer network power," in *Proc. Int. Conf. Commun.*, 1981, pp. 1–63.
[36] R. Jain, D.-M. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *Eastern Res. Lab. Digit. Equip. Corporation*, vol. 21, pp. 1–37, 1984.
[37] V. Jacobson, "Congestion avoidance and control," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architecture Protoc. Comput. Commun.*, vol. 18, no. 4, pp. 158–173, 1988.
[38] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, *The NewReno Modification to TCP's Fast Recovery Algorithm*, Internet Eng. Task Force (IETF), Fremont, CA, USA, IETF Standard RFC 6582, 2012.
[39] S. Liu, T. Başar, and R. Srikant, "TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks," *Perform. Eval.*, vol. 65, no. 6, pp. 417–440, 2008.
[40] Y.-J. Song, G.-H. Kim, I. Mahmud, W.-K. Seo, and Y.-Z. Cho, "Understanding of BBRv2: Evaluation and comparison with BBRv1 congestion control algorithm," *IEEE Access*, vol. 9, pp. 37 131–37 145, 2021.
[41] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Proc. 10th USENIX Symp. Networked Syst. Des. Implementation*, 2013, pp. 459–471.
[42] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architecture Protoc. Comput. Commun.*, 2010, pp. 63–74.
[43] H. Zhang, X. Shi, X. Yin, F. Ren, and Z. Wang, "More load, more differentiation – a design principle for deadline-aware congestion control," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2015, pp. 127–135.
[44] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis, "QTCP: Adaptive congestion control with reinforcement learning," *IEEE Trans. Netw. Sci. Eng.*, vol. 6, no. 3, pp. 445–458, Jul.-Sep. 2019.
[45] W. Li, H. Zhang, S. Gao, C. Xue, X. Wang, and S. Lu, "SmartCC: A reinforcement learning approach for multipath TCP congestion control in heterogeneous networks," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 11, pp. 2621–2633, Nov. 2019.
[46] D. Katabi, M. Handley, and C. Rohr, "Congestion control for high bandwidth-delay product networks," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2002, pp. 89–102.
[47] H. Balakrishnan, N. Dukkipati, N. Mckeown, and C. J. Tomlin, "Stability analysis of explicit congestion control protocols," *IEEE Commun. Lett.*, vol. 11, no. 10, pp. 823–825, Oct. 2007.
[48] L. Wang, L. Cai, X. Liu, and X. Shen, "Stability and fairness analysis of AIMD/RED system with heterogeneous delays," in *Proc. IEEE Glob. Telecommun. Conf.*, 2007, pp. 1813–1817.
[49] L. Georgiadis, M. J. Neely, and L. Tassiulas, *Resource Allocation and Cross-Layer Control in Wireless Networks*. Foundations Trends Netw. vol. 1, no. 1, pp. 1–144, 2006.
[50] S. Sukhbaatar, A. Szlam, and R. Fergus, "Learning multiagent communication with backpropagation," in *Proc. 30th Adv. Neural Inf. Process. Syst.*, 2016, pp. 2252–2260.
[51] J. Jiang and Z. Lu, "Learning attentional communication for multi-agent cooperation," in *Proc. 32th Adv. Neural Inf. Process. Syst.*, 2018, pp. 7265–7275.

Salma Emara (Graduate Student Member, IEEE) received the B.A.Sc. degree in electronics and communications engineering from the American University in Cairo, New Cairo, Egypt, in 2018. She is currently is currently working toward the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada. Her research interests include reinforcement learning, congestion control, and resource allocation.

Fei Wang received the B.Eng. degree in computer science and technology from Hongyi Honor College, Wuhan University, Wuhan, China, in 2020. She is currently working toward the M.A.Sc. degree with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada. Her research interests include intersections of reinforcement learning, networking, and communication.



Baochun Li (Fellow, IEEE) received the Ph.D. degree from the Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA, in 2000. Since then, he has been with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada, where he is currently a Professor. He holds the Bell Canada Endowed Chair in computer engineering since August 2005. His research interests include large-scale distributed systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. He is a Member of the ACM.

Timothy Zeyl received the B.Eng. and M.A.Sc. degrees from McMaster University, Hamilton, ON, Canada, in 2007 and 2010, respectively, and the Ph.D. degree in biomedical engineering from the University of Toronto, Toronto, ON, Canada, in 2015. He is currently a Staff Engineer with Distributed Scheduling and Data Engine Laboratory, Huawei Canadian Research Institute. His research interests include machine learning, signal processing, and data science.