Eagle: Refining Congestion Control by Learning from the Experts

Salma Emara[1], Baochun Li[1] and Yanjiao Chen[2]
[1]*University of Toronto*, {salma, bli}@ece.utoronto.ca [2]*Wuhan University*, chenyj.thu@gmail.com

***Abstract*—Traditional congestion control algorithms were designed with a hardwired heuristic mapping between packet-level events and predefined control actions in response to these events, and may fail to satisfy all the desirable performance goals as a result. In this paper, we seek to reconsider these fundamental goals in congestion control, and propose *Eagle*, a new congestion control algorithm to refine existing heuristics. *Eagle* takes advantage of expert knowledge from an existing algorithm, and uses deep reinforcement learning (DRL) to train a generalized model with the hope of learning from an expert. Learning by trial-and-error may not be as efficient as imitating a teacher; by the same token, DRL alone is not enough to guarantee good performance. In *Eagle*, we seek help from an expert congestion control algorithm, BBR, to help us train a long-short term memory (LSTM) neural network in the DRL agent, with the hope of making decisions that can be as good as or even better than the expert. With an extensive array of experiments, we discovered that *Eagle* is able to match and even outperform the performance of its teacher, and outperformed a large number of recent congestion control algorithms by a considerable margin.**

I. Introduction

For a quarter of a century, congestion control has been a cornerstone challenge and classic problem in computer networking. Initially, congestion control meant to avoid congestion collapses. Nonetheless, modern congestion control algorithms are responsible for much more, as they are the foundation for network services such as web services, video streaming, and online gaming.

As new congestion control algorithms emerged, we are always in pursuit of higher throughput, lower latencies, faster convergence to the steady-state, and utilizing scarce bandwidth resources fairly and optimally. The fundamental design objective of congestion control is to operate well in a wide array of networking environments, from lossy wireless links to links with high bandwidth and low propagation delays.

Despite a substantial amount of research, the problem of congestion control is still considered open. Existing algorithms may suffer from the following deficiencies:

- ♢ *Lack of generalization.* Most congestion control algorithms were*point solutions*, *i.e.*, they were designed for specific network environments, but may not perform well in others. For example, Sprout [1] is robust in an LTE network environment, but not for wireline connections.
- ♢ *Oblivious to earlier traffic patterns.* Not keeping a memory of preceding network events can make us forget about traffic patterns of a flow, losing the opportunity to make better actions in the future. For example, for a very dynamic network connection such as a WiFi or LTE connection, where many users join and leave the network at the same time, by observing the history, one may know that the traffic conditions change significantly and rapidly. As a consequence, a sender may need to adapt quickly to such highly dynamic conditions, as opposed to a less dynamic environment.
- ♢ *Having fixed mappings between packet-level events and pre-defined actions.* Most existing algorithms are heuristics that map certain packet-level events, such as losses, to pre-defined actions. TCP CUBIC [2], for instance, assumes that packet losses are signals of congestion, which may not hold in lossy wireless links where losses may be due to reasons not related to congestion. Such heuristics are manually tuned for their target network environments, and may not be sufficiently versatile to meet the needs of a diverse set of new network environments.

Granted, recent research works, such as Remy [3] and Indigo [4], tried to overcome the challenges we have shown in conventional heuristics by using offline learning techniques. Unfortunately, the main limitations in existing heuristics, such as having a fixed mapping between locally observed network states and responses, still survived in offline learning techniques. The design of Remy [3] and Indigo [4] does not allow adaptation to new network conditions; therefore, the mappings in their models remain fixed. Not only are these mappings fixed, but they are also mostly limited to network environments seen while training. As a result, they will perform well in network environments they were trained on, but may not perform as well in the others. Besides, even for network environments, they were trained on, they were never able to explore and extend beyond the assumptions, targets and oracles that guided their training.

These observations on existing congestion control algorithms may have already paved the way towards refining them. Intuitively, an *online* learning approach may be our best bet for achieving our goals. *First*, it should adapt well to new environments and perform well in a broader range of the different network conditions (cellular and wireline networks)

This work is partially funded by an NSERC Collaborative Research and Development Grant, a Huawei Canada Research Contract, the NSERC Discovery Research Program, the National Natural Science Foundation of China under Grants 61972296 and 61702380, Wuhan Advanced Application Project under Grant 2019010701011419, and Hubei Provincial Technological Innovation Special Funding Major Projects under Grant 2017AAA125.

that it was not trained on. *Second*, it should avoid deterministic mappings between states and actions by using a stochastic policy.

In this paper, we propose *Eagle*[1], a new congestion control algorithm that uses both expert knowledge and deep reinforcement learning to achieve our design objectives. Our key insight is to learn from an expert in an online fashion, with the hope of matching and even outperforming the expert in a diverse set of environments. The intuition is that learning is best carried out by imitating a teacher, which should be an expert, rather than purely with trial-and-error.

Highlights of our original contributions in this paper are two-fold. *First*, *Eagle* is designed to match the performance of the expert in terms of a variety of metrics, which include the convergence rate, steady-state behaviour and the ability to drain the queue when a queuing delay is encountered. It is even able to *outperform* the expert on average, as its model will continue to learn online in the real-world after the training phase completes. *Second*, as an online learning algorithm, *Eagle* is designed to adapt to newly seen network conditions or practice more on network conditions that it has already seen. Yet, it will not deviate substantially from its teacher to avoid model overfitting and the ensuing poor performance. Therefore, it can only continue to get better.

We have performed an extensive set of experiments using Pantheon, comparing to a large collection of modern congestion control algorithms. Our results show that *Eagle* has indeed been successful in outperforming its teacher, and achieved its objective in generalizing well to different network environments, even those it was never trained on. In our test environments, *Eagle* outperforms most existing congestion control protocols in the literature.

II. Preliminaries

In deep reinforcement learning (RL) [5], a learner, which is referred to as the *agent*, discovers what to do by interacting with the environment, trying several actions and learning from errors by taking higher rewarding actions. In most exciting and challenging environments, actions affect not only the current reward but also future ones.

At each discrete time step $t \in 0, 1, 2 \ldots$, the agent observes a state of the environment s_t and selects a specific action a_t based on that observed state. At the following time step $t+1$, it observes the next state of the environment s_{t+1} along with a scalar reward r_{t+1}, representing how good or bad the action was (given the observed state on step t). The agent's role is to learn a policy π, which is a mapping from observable states to actions that maximize the total expected cumulative reward in an *episode* (representing a sequence of steps), as described by $V^{\pi}(s)$ in Eqn. (1). Here, T is the number of steps in the episode, $0 < \gamma \leq 1$ is the discounted factor that weights future reward differently depending on their significance, and $\mathbb{E}_{\pi}\{\}$ denotes the expected total cumulative value if the agent follows policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\{R_t \mid s_t = s\} = \mathbb{E}_{\pi}\{\sum_{i=0}^{T} \gamma^i r_{t+i+1} \mid s_t = s\} \quad (1)$$

A large collection of DRL algorithms have emerged in the past several decades, and many of them were successful in solving real-world challenges. An important class of DRL algorithms is *policy-based* algorithms. These are algorithms that produce a probability distribution over actions given a particular state, denoted by $\pi(a \mid s)$. To be more elegant when describing a policy, it is typically parameterized by $\boldsymbol{\theta}$ as shown in Eqn. (2):

$$\pi(a \mid s, \boldsymbol{\theta}) = P\{a_t = a \mid s_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\} \quad (2)$$

Policy-based DRL algorithms work on parameterizing the $\boldsymbol{\theta}$ to give the highest probabilities to most preferred actions. Two commonly used policy-based algorithms are the *policy gradient* (and its extensions) and *cross-entropy* methods.

In policy gradient methods, we try to ascent the gradient of $\pi(a \mid s)$ with respect to our parameterization $\boldsymbol{\theta}$, searching for the local maxima in $V(s)$. Despite theoretically showing faster convergence properties, policy gradient methods usually converge to a local optimum and do not exhibit quick convergence in noisy environments.

Cross-entropy methods are Monte Carlo based methods where we consider playing entire trajectories following our current policy $\pi(a \mid s, \boldsymbol{\theta})$. The cumulative reward of an episode gives us a reasonable estimate of $V(s)$. If we train our model on episodes with higher $V(s)$, we can to converge to a global optimum. However, little information is known about whether the method can converge to a global optimum in noisy environments[2].

III. Using DRL for Congestion Control: Design Challenges

DRL is most suitable for playing and solving challenging games such as AlphaZero [7], where DRL was successful in attaining superhuman performance. If we think of congestion control as a new game where there is no specific or fixed technique to play and win the game, it is appropriate to model it as a DRL task.

A policy, as defined earlier, represents the mappings from states to actions. It can be either deterministic or stochastic. In our case, since our environment is a Partially Observable Markov Decision Process (POMDP), our policy needs to be stochastic as we do not wish to have fixed mappings between states and actions. Our environment is only partially observable because the agent can only observe and measure some network conditions such as the loss rate and round-trip time (RTT). Nevertheless, we may not be able to obtain full knowledge of the environment, such as the amount of cross-traffic sharing the bottleneck link and the bottleneck link

[1]In golf, an *eagle* implies two under par, the number of strokes needed by an expert player. In congestion control, we also wish to score better than an expert.

[2]For more details about policy gradient and cross-entropy methods, interested readers are referred to [5], [6], respectively.

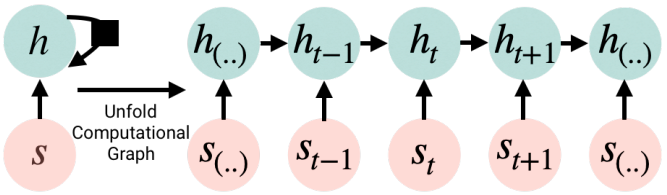


Fig. 1. The computational graph in a recurrent neural network: an example

capacity. Therefore, it is very likely that particular action to a specific state can yield different rewards in different scenarios. The policy that we will converge to will give us the maximum reward in expectation.

Concerning the choice of our neural network model in the DRL agent, we need a neural network to help us with making correct sequential decisions. A fully connected feedforward neural network would have different weights for different input features. In contrast, a recurrent neural network (RNNs) has the same parameters shared across different time steps [8]. Such a feature helps in representing each output as a function of previous outputs, leading to the sharing of parameters in a very deep computational graph, as shown in Fig. 1. This trait helps RNNs operate on a sequence of inputs, revolutionizing the research on sequential decision making. Since congestion control works on a series of successive events, a RNN becomes the best candidate for our choice of the neural network model in the DRL agent. RNN will help us to genuinely choose an action that depends on previous actions taken, hence considering the history to target a better future.

In conventional congestion control protocols governed by heuristics, a heuristic knows how good or poor its performance is by examining the observations made by the sender. Based on these observations, certain actions are taken. In contrast, if we resort to online learning with DRL, the DRL agent only has access to a scalar reward value, representing how good our performance is. Therefore, this reward function should reflect the actual performance as accurately as possible. However, since different states can imply different rewards, finding a universal reward function is not trivial, and is perhaps the most challenging problem for using reinforcement learning in the context of congestion control.

To improve the convergence rate of a DRL model — especially during the online learning phase after deployment — we wish to carefully design the reward function to produce scalar reward values that are not specific to certain environments, and that represent the performance of the agent given the chosen action.

To design a reward function so that it is even more general and universally applicable to a variety of environments, one can consider several *phases* that any flow will have to experience during its lifetime. These phases may include *startup*, *queue draining*, and *bandwidth probing*, similar to those that have been governing the behaviour of conventional heuristics, such as BBR and TCP CUBIC. The upshot of designing separate reward functions for different phases is that, we can focus on a certain subset of performance metrics in each phase (but not all of them), depending on their relative priorities.

For example, in the *startup* phase, we can focus more on the change in delivery rate until queuing delays start to be experienced. In the *queue draining* phase, on the other hand, we may only focus on negative changes in queuing delays until the queue is drained completely. Finally, in the *bandwidth probing* phase, we should focus on the changes in both delays and delivery rates. This way, we may be able to design much simpler reward functions, without the risk of losing its ability to generalize to a variety of network environments and flows.

IV. System Design

Training a model using DRL is seldom straightforward, since we have many design parameters such as the action and state space, the reward function, the neural network model and the DRL algorithm. Even the best possible conceptual design on paper may still not work well in practice, due to reasons we may only find with an extensive array of experimentation. In this paper, we take a practical approach and make our design decisions based on hands-on experiences learned from actual experiments.

A. Designing the GOLD DRL Agent: A First Cut

We begin our exploration of designing the DRL agent with a first cut, tentatively called *GOLD*, which may help us visualize and spot through experiments our issues and locate areas for improvement. Results from our preliminary experiments may offer some lessons to help achieve better performance in the next iteration of our design.

Our first cut design, GOLD, used the Vanilla Policy Gradient DRL algorithm to train its agent. We used the congestion window size (cwnd) as our control parameter and an aggressive multiplicative action space containing four actions: increasing or decreasing cwnd by a factor of 2.89, 1.5, 1.05 or does nothing.

We only had one reward function in GOLD, Eqn. 3, similar to the one Aurora [9] was using. However, instead of throughput, we replaced that with a more interesting measure called "goodness." Here we define "goodness" as the ratio between the current cwnd and the cwnd that gave us the best utility as per Eqn. 4, which is used by PCC Vivace [10]. The intuition behind using "goodness" is to make the reward function more general (and less specific) to the network environments, as discussed in the previous section.

Besides, our state-space in GOLD was simple. It was the tuple of the sending rate, loss rate and RTT gradient (the change in RTT over the step size) in the past four time steps.

$$r_t = \text{goodness}^a - b \times \text{goodness} \times \frac{\mathrm{d}RTT}{\mathrm{d}T} - c \times \text{goodness} \times L_t \tag{3}$$

$$u_t = x_t^a - b \times x_t \times \frac{\mathrm{d}RTT}{\mathrm{d}T} - c \times x_t \times L_t \tag{4}$$

where a, b and c are constant that we set to $a = 0.9, b = 0.5$ and $c = 0.5$, x_t is the sending rate at time step t in Mbps,

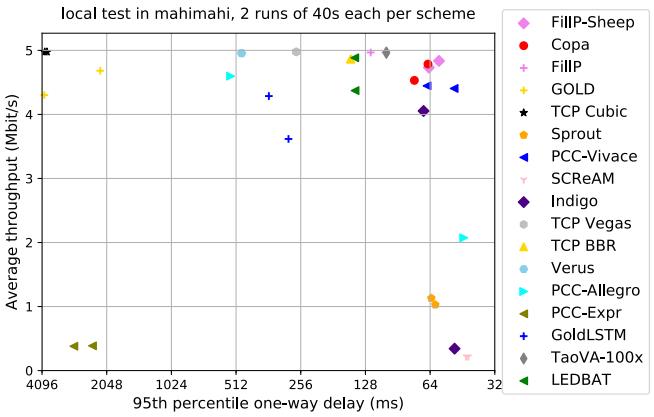


Fig. 2. The performance of DRL agents in our first cut: GOLD and GoldLSTM.

$\frac{\mathrm{d}RTT}{\mathrm{d}T}$ is the RTT gradient in milliseconds, and L_t is the loss rate ratio at time step t .

To show an illustrative example of the performance of GOLD, we can see in Fig. 2 the performance of GOLD as compared to other congestion control algorithms. The experiment is carried out using a 5 Mbps emulated link, and a one-way delay of 40 ms. It achieves high throughput, but at the cost of high delays. After a detailed post-mortem analysis of the logs, we found that the main reasons leading to this behaviour are as follows:

- ♢ *An overly aggressive action space*: The multiplicative numbers in our action space, especially the factor of 2.89, caused the sending rate to elevate significantly, causing a significant amount of queuing delay, which requires a much longer time to drain if no complementary immediate action is taken.
- ♢ *Not considering delays*: Not including delays in our reward function made us perform poorly in terms of delays. We initially conceptualized that the RTT gradient would help provide a good reward if we decrease the sending rate when we have a high delay since the rate at which the delay will increase is lower according to [10]. However, if we drop the sending rate from a value that is already above the optimum to an amount that is still above the optimum, the queue will continue to build up, and the rate at which we increase RTT continues to grow.
- ♢ *Not considering a sequence of events*: As mentioned in the previous section, we need to maintain a history of events and learn the best possible actions over all possible states. Here, we only consider that the history of the past four steps is estenial, and the rest is not. For this purpose, RNN models provide better flexibility in acquiring the traffic pattern, because it shares parameters across several time steps.

To focus more on delays, we designed and implemented an improved version of GOLD, referred to as *GoldLSTM*, where we changed the model from a simple neural network to an LSTM. We also used a less aggressive action space where we increase or decrease congestion window size by a factor of 1.75, 1.5, 1.05 or do nothing. At the same time, we also improved the reward function to focus more on delays when we have a high queuing delay. The utility function in GoldLSTM is defined in Eqn. 5, and its reward function is simply the change of the utility function as in Eqn. (6):

$$u_t = \begin{cases} \text{goodness}^a & \text{delay} = 0 \\ \text{goodness}^a - b \times \text{goodness} \times \text{delay} & \\ -c \times \text{goodness} \times L_t & \text{delay} > 0 \end{cases} \tag{5}$$

$$r_t = \Delta u_t. \tag{6}$$

We may observe in Fig. 2 that, the performance of GoldLSTM improved by a remarkable margin compared to GOLD in terms of delays. However, after another round of log analysis, we found that the main issue in GoldLSTM was mainly in the phase after draining the queue. GoldLSTM would prefer to continue probing up aggressively as it did in the startup phase, where there was no delay. The two main actions that it was using is multiplying cwnd by 1.75 or divide it by 1.5. This is what GoldLSTM converged to, which is not the optimum performance.

The intuitive way of addressing this issue is to add another case to the reward function, where we reward an increase in the sending rate as long we have not experienced any delay before (*i.e.*, when we are in the startup phase). In contrast, after entering the queue draining phase, we penalize the reward significantly for any small increases in delays.

In both GOLD and GoldLSTM, our DRL agent did not generalize well to other network environments. Also, a randomly initialized model can perform extremely bad actions in the beginning, causing very high delays. Since our step size depends on RTTs, bad actions can increase the step size to hours, leading to very long training times.

B. Eagle: Design Choices and Rationale

To overcome all the issues encountered in GOLD and GoldLSTM, we are now ready to discuss the choices we made in our new DRL-based algorithm, *Eagle*, with respect to our model parameters and the DRL algorithm.

Neural Network Model. There are several different neural network models that prove their applicability in different applications. However, given our problem, which is based on sequential decision making to reach an optimum utilization of a network, we need a model that remembers the past states well and act accordingly. Therefore, we use Long-short term memory (LSTM) as our neural network model to train. It is a two layered LSTM with 64 hidden units.

Actions. Actions are discrete changes to the sending rate and the size of the congestion window (cwnd). In our formalization, our agent resides at the sender, and it interacts with the environment by adjusting its sending rate by a_t.

Typically, we want cwnd to be equal to the bottleneck bandwidth-delay product (BDP) to maximize network utilization. Intuitively, this would be equivalent to having a sending

rate equal to the bottleneck bandwidth for fully utilizing the network. As the agent adjusts its sending rate, it changes its cwnd to a small multiple of its estimate of the BDP. The BDP estimate is the product of the highest delivery rate and minimum round-trip time. As inspired from BBR, the sender can increase or decrease its sending rate by a factor of 2.89 or 1.25 or do nothing. With a 2.89 or 1.25 times increase or decrease in sending rate, cwnd is increased or reduced by a factor of 2 or 1.25, respectively. We use the same numbers used by BBR since it has been proved to converge quickly in practice.

Time is divided into unequal non-overlapping intervals, as time steps t in reinforcement learning. The sender adjusts its sending rate given the observed network state at the beginning of each time step t. Each time step lasts for three RTTs. This is to allow enough time for the network conditions to settle, and for the newly adjusted sending rate of the agent to take effect. Though pre-defined step sizes can also be considered, it is always advisable to have a small action space when possible to help in fast convergence.

States. Information related to the observed network conditions are measured by the agent. Aurora [9] mentioned that the more past observations you take into consideration, the better performance will be. However, there is always a tradeoff between converging to an optimum policy quickly and having a large representative state space. To take the best of both worlds, we choose a state s_t to be a summary of the past four observation steps as inspired by the DRL applied to Atari games in [11]. Observations in each step are:

- ♢ a binary number indicating whether the agent experienced delay before;
- ♢ if we experience a long delay, the difference between the number of times of decreases and increases in the sending rate, starting from the point the queuing delay increases;
- ♢ an exponentially weighted moving average (EWMA) of the queuing delay, calculated as the difference between the currently measured RTT and the minimum RTT observed in the past 50 steps;
- ♢ an EWMA of the loss rate;
- ♢ the ratio of the change in the delivery rate.

Reward Function. Learning from our experiences with GOLD and GoldLSTM, we train *Eagle* with different reward functions for different cases, which we classify into phases: the *startup*, *queue draining*, and *bandwidth probing* phase.

Startup. This is the initial phase in which the sender has not experienced any delay yet. Initially, as the flow starts, it begins with an initial sending rate of 300 Kbps and an initial cwnd of 10. After the first two steps, the round-trip propagation time can be estimated. The product of the newly estimated round-trip time and the next chosen sending rate multiplied by a small multiple is then set to be the cwnd. This is to ensure that we are effectively sending at the sending rate selected.

As long as the sender did not measure any increase in queuing delay, the reward is measured as the ratio of change in the delivery rate as ratio Δd as shown in Eqn. 7. Since we want good actions in different phases to have reasonably similar rewards, the constant five used is to help set an upper bound of the reward, comparable to the upper bounds of the reward functions in other phases. The higher the change in the delivery rate, the higher will the reward be. For example, Fig. 3 shows an example of the decisions of an agent during the early iterations of the training phase, where the bottleneck bandwidth is 10 Mbps. The startup phase observed from time step 0 to 3. Fig. 3a shows the chosen sending rate, Fig. 3b shows the queuing delay measured, and Fig. 3c shows the reward corresponding to each time step. From time step 0 to 3, the sending rate is increasing, and the delay is negligible; therefore, the reward can be seen to increase.

$$\text{reward} = 5 \times \text{ratio } \Delta\text{d} \tag{7}$$

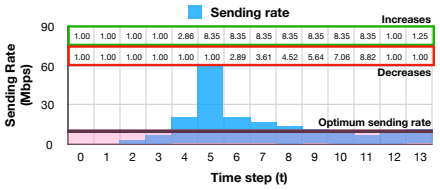
Queue draining. The agent enters into this phase when it experiences its first increase in delay. As the queuing delay is experienced for the first time, we start to accumulate the actual time of increases and decreases in the sending rate. For example, Fig. 3a shows some example decisions and how the times of increases and decreases vary as these decisions change. In time step 4, the sending rate increases by 2.89 and in time step 5, it increases again by 2.89, making the cumulative increase times 8.35. As observed, when the difference between the times of decreases and increases goes up, the reward decreases since this is causing an increase in queuing delay. However, step 4 caused a small increase in the delivery rate, so the reward of that action in step 4 is not too high as it was still in the startup phase. As the times of decreases approach the times of increases, we wish to drain the queue and reward that positively. This can be seen in steps 6 to 8 in Fig. 3. The exact algorithm for computing the reward used in the queue draining phase is shown in Algorithm 1.

Algorithm 1 Calculating the reward in the queue draining phase

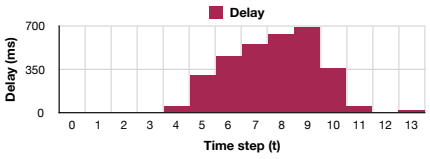
```
1: if did not experienced delay before then
2:     reward = ratio Δd × 5 − 0.5 × queuing delay
3: if experienced delay before then
4:     reward = 5 × % of decreases in (increases − decreases)
5:     if (increases ≤ decreases) then
6:         reward = % decrease in delay
```

Bandwidth probing. After the queue is drained, and the delay goes to 0, the cumulative times of increases and decreases are reset back to 1. This starts from time step 12 in Fig. 3. However, the sender still needs to probe up for the sending rate to fully utilize the link capacity. Here, we wish to reward any increase in ratio Δd, but penalize any increase in queuing delays (as opposed to the startup phase) as shown in Eqn. 8. If we again encounter an increase in queuing delays, we reward actions as in the queue draining phase.

$$\text{reward} = \text{ratio } \Delta\text{d} \times 5 - 0.5 \times \text{queuing delay} \tag{8}$$



(a) Sending Rate



(b) Delay

(c) Reward

Fig. 3. All possible phases of a flow: an example

C. Eagle: Learning from Expert Demonstrations

To overcome the challenges above and successfully design a DRL agent for congestion control that can outperform the state-of-the-art congestion control protocols, we seek help from an expert congestion control algorithm: BBR.

Just like humans, when they learn a task, trial-and-error may not always be their best technique; instead, it is still better to watch a teacher do a job first. As humans start to master the task, they could be as good as the teacher or become even better if they practice more. In the context of congestion control, our teacher is BBR, and our pre-trained model is the student. The heuristic algorithm in BBR may not always be optimal or perfect. Still, as it is proven to outperform TCP CUBIC, we believe it is an all-round performer and an excellent choice as a teacher. This can also guarantee that we do not converge to an unsatisfactory policy in our agent since the right actions of BBR will help us get out of the local optimum.

In addition, we also consider BBR perfect in *emergency* bad situations, where the delay is notably high, and the model is immature and does not know how to reduce it. This can help us overcome long training times due to large step sizes since step sizes depend on RTT. This is also important after convergence if the model starts to perform poorly in unseen network conditions, there will always be an expert to help it adapt to newly encountered network conditions.

Though the idea of learning from an expert is appealing, it is non-trivial to design Eagle to implement it. As an example of the challenges involved, BBR takes sequential decisions based on its *own* previous *good actions* that it made in different working modes (startup, queue draining and bandwidth probing). It will not be straightforward to use BBR after our

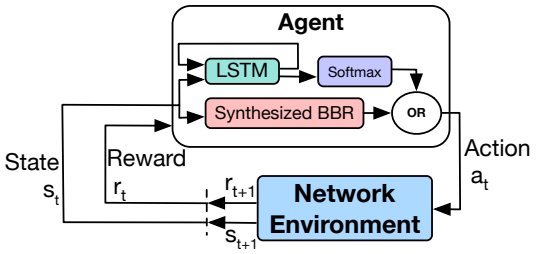


Fig. 4. The framework of Eagle

model performs some *bad actions*. This is because BBR is not designed for that and will not perform well. It was difficult for us to have BBR show some expert demonstrations after our untrained model performs some good and bad actions.

Our solution addresses these challenges using a synthesized BBR algorithm that imitates its behaviour, rather than directly using BBR's real-world implementation. We wish to not only ensure that the performance of our synthesized BBR is as good as BBR's real-world implementation, but also prepare our synthesized BBR for deplorable network conditions (that will never be experienced by the actual BBR), where the delay or loss rate is too high.

Fig. 4 helps in visualizing Eagle's final design framework, covering all the details we highlighted in this section.

D. Training Eagle

We train the DRL agent in Eagle using a custom made `OpenAI Gym` environment, in which we run mahimahi shells [12]. Mahimahi in Linux is a container-based network emulator that uses the real Linux stack across virtual network interfaces to send and receive packets. These emulated networks have five parameters to emulate in the network: (1) the fixed round-trip propagation time, (2) the fixed or variable bandwidth capacity, (3) the queue size, (4) queuing disciplines such as Drop Tail FIFO, and (5) the stochastic packet loss rate. We used a wide variety of fixed and stochastic bandwidth values to train our model (20 – 120 Mbps), fixed round-trip propagation times (10 – 400 ms), as well as different queue sizes and loss rates.

In the emulated network for training, we consider an episode spanning 50 steps, where each step is 3 RTTs. Every time the environment resets, a new flow starts with new network parameters. Changing network parameters helps us avoid overfitting our model to specific network conditions. In other words, since we are not training our model over just a few tasks, the shared network patterns among different conditions are learned. Therefore, if our agent is to "play" or send over a network that it has not seen before, it will still perform reasonably well.

We train Eagle using the cross-entropy method, in which we play a certain number of episodes and calculate the total reward for each episode, as the sum of the rewards of the 50 steps in the episode. We then choose the elite percentile of these episodes (3 out of 5 in a batch) with the best total

reward, labelling the actions taken in these episodes as desired actions. We repeat this process until we converge to a policy that stops changing.

In noisy (non-deterministic) environments, such as the Internet environment, we know very little about the convergence properties of cross-entropy, and this is the main reason we use our synthesized BBR as a teacher. However, our experiences have shown that cross-entropy has a faster convergence rate as compared to other policy gradient methods.

Since we wish our agent to outperform synthesized BBR, we do not wish to have many actions played by synthesized BBR. Therefore, our training process starts in the first batch with having synthesized BBR play only two episodes out of five, then slowly decrease this number to one.

In addition, in the episodes where the model takes actions, we periodically play some actions by synthesized BBR. The number of such actions decays with time to allow Eagle to explore the state space more – more precisely, "the most likely to see" state space. Seeking help from BBR eliminated the very corrupt observations in the state space and did not limit Eagle's exploration. In cases where the observed state shows a high delay, this is an emergency, and the model actions may make it worse if it has not experienced such states before. In such emergencies, synthesized BBR intervenes to act and quickly get out of these corrupt states. Complete details of our training algorithm are presented in Algorithm 2.

Algorithm 2 Training Eagle by Learning from Synthesized BBR

```
procedure TrainEagle
    Initialize network parameters randomly
    i ← 0
    for all batches do
        Play a decreasing small portion of the batch using BBR
        for all episodes ∈ batch do
            if emergency then
                i ← i + 1
            else i ← 0
            if i > emergencies to ignore then
                Play a synthesized BBR action
            if time to play BBR action then
                Play a synthesized BBR action
            else Play Eagle agent action
        Train on elite episodes
        Decrease the ratio of BBR:Eagle actions in an episode
        if episodicRewardMean > 200 then
            Increment number of emergencies to ignore
            Reduce frequency of BBR actions to 5%
```

E. Testing and Deployment

After Eagle is fully trained, it is deployed in emulated networks for testing. At this point, the definition of episode changes. The size of the flow is now the size of the episode. In order not to favour longer flows in the online learning process, we decide to weigh the episodic return of each episode in the batch with a ratio between the total number of steps in the batch and the number of steps in this episode.

Algorithm 3 shows how we continue to train Eagle online after deployment. We still have some flows played by synthesized BBR, as we do not wish Eagle to deviate much from BBR's performance. This is to make sure Eagle does not deviate towards worse performance. From our observations, the fully trained Eagle takes at most 2 – 4 steps to drain a queue. This is why we choose to have synthesized BBR intervene when the delay remains high for more than 4 steps.

Finally, even in deployment, we find that having synthesized BBR playing some steps in an episode gives us a better performance. This is because synthesized BBR in the bandwidth probing phase, takes the "do nothing" action; however, Eagle is more aggressive towards exploring the environment quickly. By mixing the teacher and the "student," we take the best of both worlds.

Algorithm 3 Training Eagle Online after Deployment

```
procedure DeployEagle
    Use pre-trained network parameters
    i ← 0
    for all batches do
        Play a small portion of the batch using BBR
        for all flows ∈ batch do
            if emergency then
                i ← i + 1
            else i ← 0
            if i > emergency situations to ignore then
                Play a synthesized BBR action
            if time to play BBR action then
                Play a synthesized BBR action
            else Play Eagle agent action
        Train on elite flows
```

V. Performance Evaluation

To evaluate Eagle's performance, we use an accessible standard testbed for congestion control schemes: Pantheon [4]. Results from Pantheon are reproducible, and closely approximate real-world results. This is because Pantheon uses mahimahi shells to emulate the network accurately.

We choose to present test scenarios that are different from the training environment, and that involve highly dynamic links such as LTE cellular links. It is worth noting that Eagle always performs as well as alternative schemes in network environments it was never trained on. One of the main design objectives of Eagle that it should generalize well to different network scenarios, and this is shown by performing well in network environments it was never trained on. It is important to note here that our model updates its weights only after a batch has been played, hence there were no gradient updates

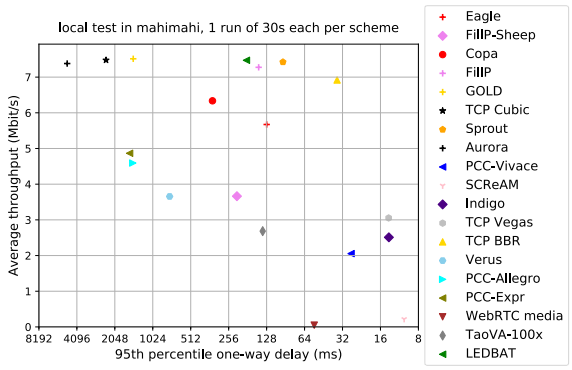


Fig. 5. Performance over an emulated Verizon LTE network link

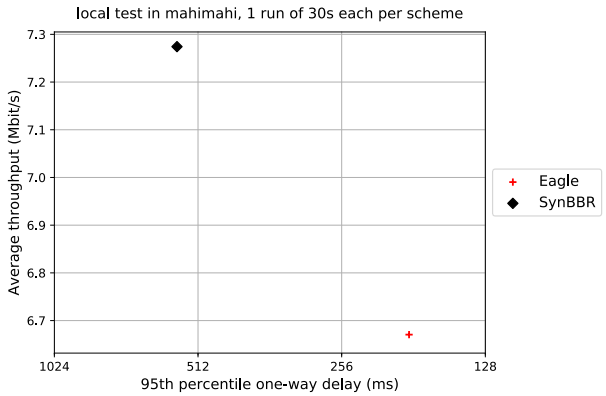


Fig. 6. Performance over a Verizon LTE link: Eagle Vs. Synthesized BBR

while running Eagle in these tests. In other words, current tests were only carried out on the pre-trained model of Eagle.

Fig. 5 shows Eagle tested with other congestion control algorithms in Pantheon. The test involves running a trace file emulating a Verizon LTE link provided by mahimahi, with a one-way delay of 10 ms. As depicted in Fig. 5, we can observe that Eagle enjoys a level of performance that is close to BBR and the top-3 congestion control algorithms. This illustrates two main properties in Eagle. First, it converges rapidly in both the startup and the queue draining phase. Second, although Eagle never saw such a network environment as it was trained on only fixed bandwidth links, its performance is close to BBR. It can even perform better if it is provided the opportunity to train more on the LTE link.

To better show the strength of Eagle, Indigo [4] is an example of a pre-trained model offline. As [4] mentioned, Indigo was trained in a wide range of settings to ensure it performs well in various environments. In contrast to such a claim, in this testing scenario, we have shown a failed example for Indigo, in which it did not generalize well to the LTE wireless network environment.

Our first cut, GOLD, performed better than Aurora, which is a pre-trained DRL agent [9]. It has 3.3× lower delay, and similar average throughput. In comparison, other online learning methods such as PCC and its extensions [10], [13] performed poorly, since they have lower convergence rates, which is unsuitable for the dynamically changing LTE link.

Another strength in Eagle is that it indeed was capable of outperforming its teacher, the synthesized BBR. Fig. 6 shows by how much is Eagle performing better than its teacher. The one-way delay is lower for Eagle by almost 5×, the loss rate is lower by 2×, and the throughput is comparable.

We also wanted to test Eagle in narrower fixed-bandwidth links outside its training range of bandwidths. For this purpose, we used an emulated link of 10 Mbps and RTT of 100 ms. In Fig. 7, we still see Eagle ranking among the best five schemes. Here, we see a generalized excellent performance as opposed to Indigo, PCC, PCC Vivace, Aurora and Sprout, which did not perform well in a cellular environment. Another essential trait that we see in Eagle is its stability over two different runs,

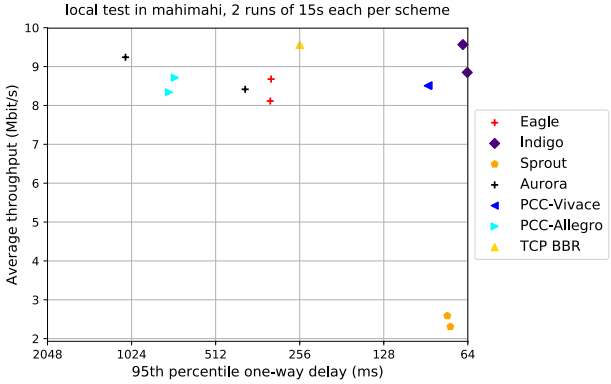


Fig. 7. 10 Mbps bandwidth and 100 ms RTT

as compared to the high variance we observed with Aurora.

We now move to show the convergence properties of Eagle in the startup phase to investigate what to expect of Eagle when it operates on a short flow. We decrease the time length of flows to 3 seconds with RTTs of 100 ms, it still ranks to be one of the best four protocols as shown in Fig. 8.

Now, we wish to take an in-depth look into the performance of Eagle and BBR at steady state, to examine how well Eagle performs. Fig. 9 and Fig. 10 show the performance of Eagle and BBR respectively for the same emulated link of 10 Mbps and 100 ms RTT. We see that BBR and Eagle have almost the

TABLE I
LOSS RATES IN DIFFERENT SETTINGS

Protocol	LTE	120 Mbps	10 Mbps
Aurora	11.45	9.81	4.45
BBR	0.07	0.37	0.90
Eagle	0.26	0.07	1.19
Indigo	0.02	0.13	0.42
PCC-Allegro	0.48	0.16	0.35
Sprout	0.04	0.35	0.29
PCC-Vivace	0.00	0.15	0.41

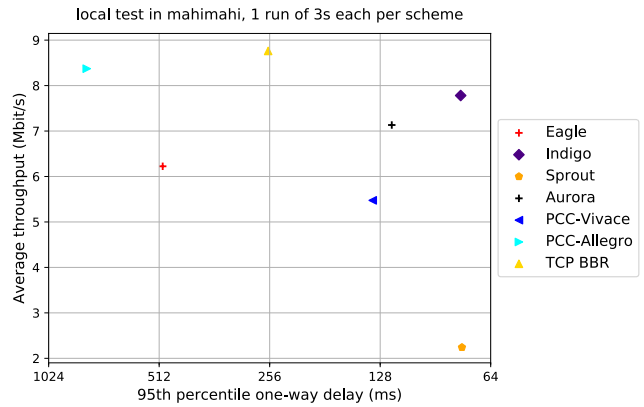


Fig. 8. Convergence properties of Eagle for shorter flows and longer RTTs

same dynamics at steady state; however, what may currently prevent Eagle from outperforming BBR is its slightly more aggressive behaviour during the startup phase.

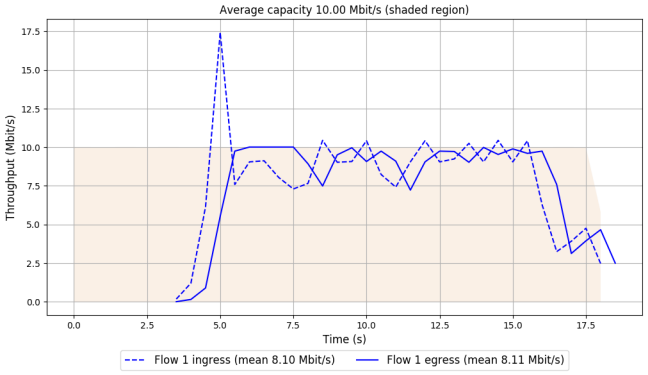


Fig. 9. Eagle's steady-state behaviour

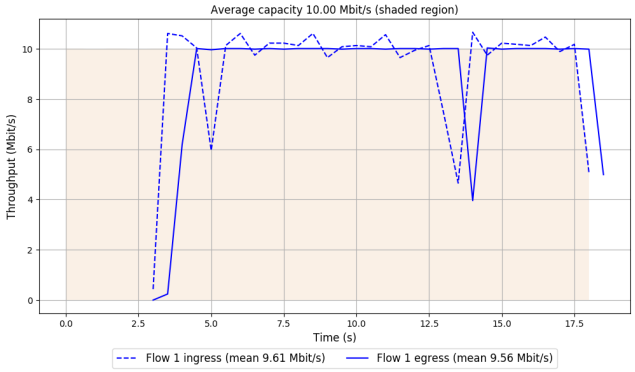


Fig. 10. BBR's steady-state behaviour

Table I shows the loss rate of Eagle and a few other well-performing protocols. We may observe that Aurora experienced very high loss rates. The other different protocols, such as PCC, could not be competitive with the best five protocols over LTE links, and their loss rates were high as well. Eagle, on the other hand, showed the lowest loss rate in the setting with a fixed bandwidth of 120 Mbps.

VI. Related Work

Online Learning. To avoid any hard wired mappings between states and actions, PCC [13] and PCC Vivace [10] proposed new Internet congestion control protocols based on live evidence. By trying marginally smaller and larger sending rates of the current sending rate, the sender increases or decreases their sending rates respectively in the direction of an increased utility function. Although both PCC [13] and PCC Vivace [10] are robust because they depend on live empirical evidence, their slow convergence time still serve as a critical limitation.

Offline Learning. At this core, Indigo [4] is an offline-trained neural network model. Indigo is trained using imitation learning, where oracles serve as the expert. An LSTM saves the mappings between states and actions, and these mappings are never changed. Remy [3] is another offline optimization framework for congestion control. An obvious limitation for both techniques is that Indigo and Remy would have a near-optimum performance when it runs on links it was pre-trained on. However, as the network environment changes, the models may not generalize well.

RL Approaches. Iroko [14] develops a new RL-based congestion control protocol, but it is highly specific to datacenter networks only. With the assumption that nodes are all managed by the same organization, it is able to exploit global visibility of the network. Aurora [9] uses a simple neural network and an off-the-shelf RL algorithm for congestion control over the Internet. However, it does not perform well in our experiments, and since it is not designed for online training, it may not generalize and adapt to new network conditions. There are also some preliminary results by [15], [16].

VII. Concluding Remarks

In this paper, we presented *Eagle*, a new congestion control protocol powered by deep reinforcement learning, a synthesized BBR teacher, and a hands-on approach in its design philosophy. Our evaluation results show that Eagle is competitive with the state-of-the-art, and can generalize well to different network environments. Eagle even performed well in network environments it was never trained on, including dynamic networks such as LTE and stable ones with fixed bandwidth. Not only was Eagle was capable of beating its teacher, but its performance is very close to BBR in steady-state. We believe that *Eagle* represents a step forward towards realizing self-learning congestion control protocols that generalize well to a wide range of network environments.

In future work, we plan to extend our experimental setup to examine Eagle's behaviour thoroughly in the online-training phase – after deployment. This is to ensure that online learning doesn't make our model deviate much from its pre-trained model, and if it deviates it only deviates to a better performance. We also want to test the fairness among different Eagle flows, even after they change slightly during online training.

REFERENCES

[1] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[2] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS Operating Systems Review*, 2008.

[3] K. Winstein and H. Balakrishnan, "TCP ex Machina: computer-generated congestion control," in *Proc. ACM SIGCOMM*, 2013.

[4] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for Internet congestion-control research," in *Proc. USENIX Annual Technical Conference (ATC)*, 2018.

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

[6] R. Rubinstein and D. Kroese, *The Cross-Entropy Method*. Springer, 2004.

[7] D. Silver, J. Schrittwieser, K. Simonyan, and et al., "Mastering the game of Go without human knowledge," *Nature*, vol. 550, 2017.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[9] N. Jaya, N. Rotman, P. B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on Internet congestion control," in *Proc. 36th International Conference on Machine Learning (ICML)*, 2019.

[10] M. Dong, T. Meng, D. Zarchy, E. Arslan, and Y. Gilad, "PCC Vivace: Online-learning congestion control," in *Proc. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 343–356.

[11] T. Hester, M. Vecerik, and O. Pietquin, "Deep Q-Learning from demonstrations," in *Proc. 32nd AAAI Conference on Artificial Intelligence*, 2018.

[12] R. Netravali, A. Sivaraman, S. Das, and A. Goyal, "Mahimahi: Accurate record-and-replay for HTTP," in *Proc. USENIX Annual Technical Conference (ATC)*, 2015.

[13] 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 Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[14] F. Ruffy, M. Przystupa, and I. Beschastnikh, "Iroko: A framework to prototype reinforcement learning for data center traffic control," in *Proc. NeurIPS Workshop on Machine Learning for Systems*, 2018.

[15] J. Fang, M. Ellis, B. Li, S. Liu, Y. Hosseinkashi, M. Revow, A. Sadovnikov, Z. Liu, P. Cheng, S. Ashok, D. Zhao, R. Cutler, Y. Lu, and J. Gehrke, "Reinforcement learning for bandwidth estimation and congestion control in real-time communications," in *Proc. NeurIPS Workshop on Machine Learning for Systems*, 2019.

[16] V. Sivakumar, T. Rocktäschel, A. Miller, H. Küttler, N. Nardelli, M. Rabbat, J. Pineau, and S. Riedel, "MVFST-RL: An asynchronous rl framework for congestion control with delayed actions." in *Proc. NeurIPS Workshop on Machine Learning for Systems*, 2019.