Rax: Deep Reinforcement Learning for Congestion Control

1st Maximilian Bachl
Institute of Telecommunications
TU Wien
Vienna, Austria
maximilian.bachl@tuwien.ac.at

2nd Tanja Zseby
Institute of Telecommunications
TU Wien
Vienna, Austria
tanja.zseby@tuwien.ac.at

3rd Joachim Fabini
Institute of Telecommunications
TU Wien
Vienna, Austria
joachim.fabini@tuwien.ac.at

***Abstract*—This paper proposes *Reactive Adaptive eXperience* based congestion control (Rax), a new method of congestion control (CC) that uses online reinforcement learning (RL) to maintain an optimum congestion window with respect to a given reward function and based on current network conditions. We use a neural network based approach that can be initialized either with random weights or with a previously trained neural network to improve stability and convergence time. As the processing of rewards in CC depends on the arrival of acknowledgements, which are delayed and received one by one, the problem is not suitable for current implementations of Deep RL. As a remedy we propose *Partial Action Learning*, a formulation of Deep RL that supports delayed and partial rewards. We show that our method converges to a stable, close-to-optimum solution within minutes and outperforms existing CC algorithms in typical networks. Thus, this paper demonstrates that Deep RL can be done online and can compete with classic CC schemes such as Cubic.**

I. Introduction

Congestion control (CC) has been a long-standing problem in networking. In the past decades various attempts have been made to devise a set of rules that perform optimum CC. Recently, researchers have realized that such a set of rules might be impossible to find due to the complexity of the current Internet (caused by various types of middleboxes, different CC algorithms and Active Queue Management) and the fast-paced evolution of Internet technologies [10]. Two principal solutions have been proposed recently: (1) Remy, a machine learning approach that is trained offline in a simulator and then statically deployed [10], and (2) PCC, which learns online but forgets all it learned after each flow [2]. We argue that the solution lies in a hybrid approach: A machine learning based CC that receives some initial pre-training and then uses online learning to constantly refine and optimize its behavior.

Recent advances in neural network based reinforcement learning (RL), which for the first time have the ability to learn complex tasks such as video games, seem to be an ideal method to model end-to-end CC: As in RL, in CC each sender has an observable state, performs actions and receives rewards for its actions according to the sender's objective function.

While CC seems to be a well-suited task for RL, there are a couple of unique aspects and differences that distinguish it from most other RL problems: (1) **Delayed rewards**: In CC, rewards are always delayed by one round-trip time.

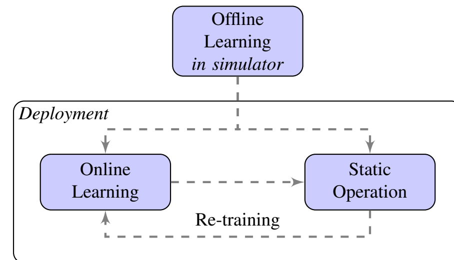


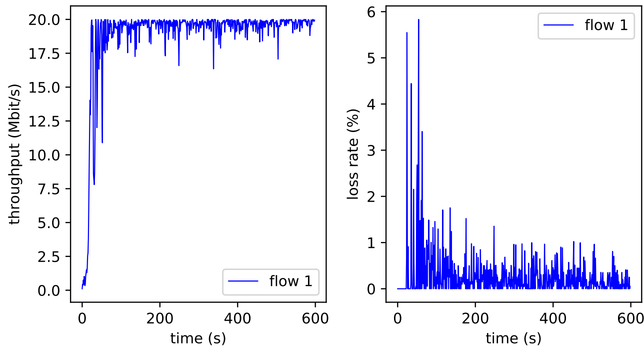
Figure 1: Hybrid learning approach to Congestion Control

(2) **Non-Atomic Actions and Rewards**: We do not have atomic actions and rewards because an action typically consists of multiple packets being sent and thus we need to introduce the concept of *partial actions*. (3) **Variable number of actions and rewards**: If the congestion window (CW) is larger, more packets are sent and acknowledged, which means that the RL algorithm gets more training data with a large window.

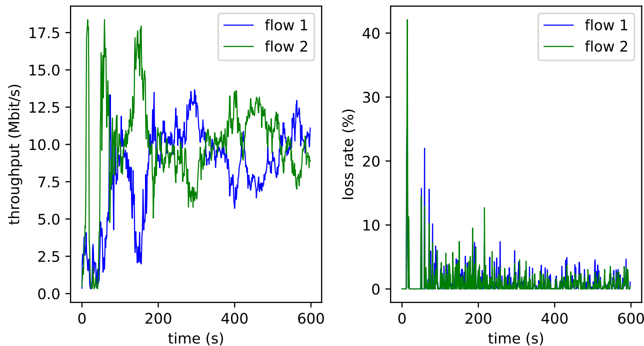
To address these three problems we develop a new variant of Deep RL called *Partial Action Learning* (PAL) which we base on the Asynchronous Advantage Actor Critic framework (A3C) [7]. PAL is well suited to train an optimum CC for a specific range of network scenarios in an offline fashion. Offline learning of an optimum CC has already been addressed by a previous approach called *Remy* [10]. Nevertheless, in general PAL achieves a much higher training speed than Remy, which makes PAL not only faster usable for deployment but also applicable for online training. Thus, additionally, PAL can be trained online without any pre-knowledge about the network environment (Figure 2). Another possibility is to combine both options: To use offline learning to generate a basic model of CC for a wide range of networks and then use online learning to continuously refine it (Figure 1).

II. Related Work

Additional to the two aforementioned works, Remy [10] and PCC [2], [3], several other proposals have been made to improve CC with the help of machine learning: [4] predict if an event of packet loss is caused by congestion or a transmission error of the physical layer. More recently [6] use RL to implement a CC, using the method of *Q-learning* [9]. [5]



(a) One sender



(b) Two senders

Figure 2: One (2a) and two (2b) Rax senders learning online without pre-training on a bottleneck link of 20 Mbit/s with a two-way end-to-end delay of 50 ms and a very small buffer of $\frac{1}{10}$ bandwidth delay product between two senders.

implement two CC algorithms using *SARSA* [5]. Contrary to these two approaches, we use Deep RL.

[11] use Deep Learning for CC but do not rely on RL but instead use *Imitation Learning*. For this, they construct a *Congestion Oracle* which knows the optimal Congestion Control policy for a large number of network conditions and let a neural network learn this policy. While their approach shows promising results, unlike our proposal, they cannot continue learning *in the wild* after deployment of their method.

III. Proposed Method

In general, in RL there is an agent which observes a *state* from the environment and then performs the *action*, which the agent estimates will yield the highest *reward*. We base our RL framework on the Asynchronous Advantage Actor Critic framework [7], which uses two underlying neural networks: (1) the *critic network*, which estimates the reward that it expects to receive given the current state and the (2) *actor network*, which tries to perform an action which results in a reward that is higher than what the critic network predicted.

A. Partial Action Learning

The key difference between our approach PAL and previous Deep RL frameworks is that PAL can handle the delay of rewards. Another major difference in PAL is that one action generates several partial actions (≥ 0). Each partial action generates partial rewards upon interacting with the environment. Furthermore, upon receiving a partial reward, the agent determines the current state and performs a new action. When all partial rewards of one action were received, the agent combines them to form the reward, which is used for the RL (Figure 3).

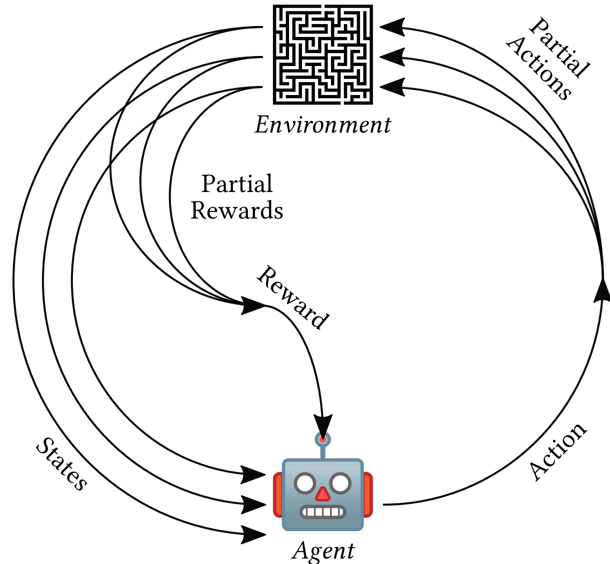


Figure 3: Partial Action Learning: An action consists of zero or more partial actions which generate partial rewards upon interacting with the environment. Furthermore the state is updated alongside each partial reward that is received.

B. Application to Congestion Control

To use RL for CC in general and our modified approach PAL specifically we first have to define the correct semantics for this specific use case and define the reward etc. Furthermore, we have to define how the actor and the critic network explicitly work in case of CC. In the following, a time step t ($t \in \mathbb{N}$) corresponds to the reception of an acknowledgement (ACK). The beginning of the flow, the time when the first packet is sent, corresponds to time step 0.

State: The state s_t (“congestion state”) consists of: (1) the time between the last two ACKs that were received (2) the RTT of the last received packet (3) a loss indicator that shows if the last packet was lost, where 1 denotes that the last packet was lost while 0 indicates that it was received correctly (4) the current CW (as a real number, as actions by output by the actor network can be very small (a fraction of a byte) and thus the CW is often changed by minuscule amounts) These features are not used themselves but instead of them an exponentially

weighted moving average (EWMA) with a factor α of $\frac{1}{8}$ and one with an α of $\frac{1}{256}$ is computed, which makes a total of 8 features. The concept of using EWMAs for smoothing with these specific values of α was first introduced by [10]. Each time an ACK is received, the state is updated and the actor network is asked for the next action.

Action: The action a_t is a real number that represents the change to the CW. It is computed based on a given state and the history of previous states, which is considered as our neural network uses Gated Recurrent Units (GRUs) [1].

It might seem simplistic to only use an additive increase and decrease to the CW and no more sophisticated mechanisms such as a combination of a multiplicative increase and an additive increase. However, GRUs have a nonlinear output function and thus they can learn to model complex relationships between the input and the output.

Reward: The reward is received from the environment at each time step t. In fact, we do not use just one reward but several *reward metrics*:

$r_{\text{sent},t}$: the sum of all bytes in all packets the sender sent during time step t.

$r_{\text{received},t}$: the sum of all bytes in all packets that the sender sent during time step t and that were not lost (so they were acknowledged by the receiver). Example: During time step t, 2500 bytes are sent, of which 1000 are lost and 1500 are acknowledged. Then $r_{\text{received},t}$ is 1500 at time step t. However, $r_{\text{received},t}$ can only be determined after the sender has received the ACKs for all the bytes that he sent. Thus one has to wait one round-trip time until all rewards can be calculated.

$r_{\text{duration},t}$: the time between receiving the last ACK and receiving the current ACK ("inter ACK reception time") summed over all packets that the sender sent and that were acknowledged. Example: The sender sent three packets, two of which were acknowledged (one was lost). The time between receiving the ACK of the first packet and the previous one (received at an earlier time step) is 5 ms and the time between receiving the ACK of the second packet and the previous ACK is 6 ms. So for this time step $r_{\text{duration},t}$ is 11 ms.

Experienced long-term reward: With the reward metrics $r_{x,t}$ ($x \in \{\text{sent, received, duration}\}$) and a factor γ with $0 < \gamma \leq 1$, which stands for the influence that future rewards have on the moving average, we define the actual experienced long-term reward $R_{x,t}$ at time step t as $R_{x,t} = \gamma r_{x,t} + (1-\gamma) R_{x,t+1}$. To compute $R_{x,t}$ at time step t it is necessary to look infinitely far into the future to get all future rewards as in each step t computing $R_{x,t}$ relies on knowing $R_{x,t+1}$. Thus, the algorithm takes t_{max} steps, waits for all the rewards and then calculates $R_{x,t}$ for each step and uses an estimate of $R_{x,t_{\text{max}}+1}$ provided by the critic network as the continuation in the future. For Rax we set t_{max} to be one full CW of packets: For example, if the CW is 15003 bytes, then we wait until we receive the next t_{max} packets that contain at least 15003 bytes together. If the next 16 packets contain 15040 bytes, then t_{max} is 16. We update the neural network for these 16 steps, look at the current CW and repeat the procedure to obtain the next t_{max}.

Having these three long-term reward metrics, one can compose a variety of functions with them.

Estimated long-term reward: The value $v_{x,t}$ ($x \in \{\text{sent}, \text{received}, \text{duration}\}$) is the long-term reward that is estimated by the critic network. For each of the three previously defined experienced long-term reward metrics the critic has one output. The overall structure of the neural network for both the critic and the actor network is depicted in Figure 4. Both networks are updated after t_{max} steps.

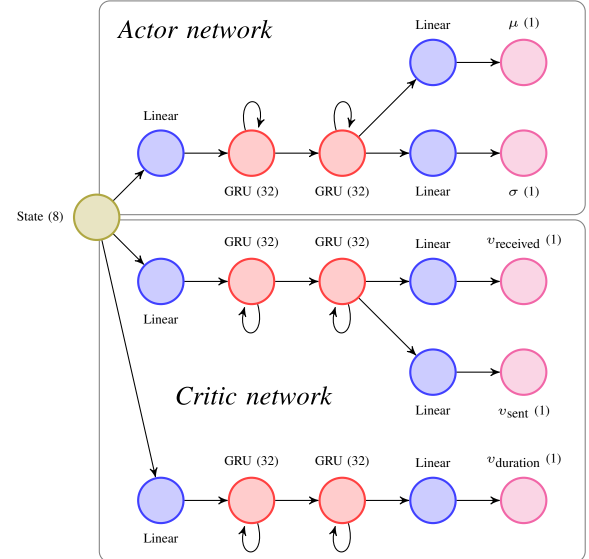


Figure 4: An overview of the complete neural network being used in our experiments. Ocher stands for the input, blue for linear layers, red for GRU layers and purple for the outputs of the neural network. The numbers in parentheses next to each label stand for the width of that layer (i.e. the number of neurons except for the inputs and outputs). Parts of the neural network are shared between v_{received} and v_{sent} as well as μ and σ as they output quantities of similar magnitude (The expected number of bytes sent and the expected number of bytes received are usually of similar size.) and thus the gradients that are computed when updating the neural network are of the same order of magnitude. In our experiments we saw that if one uses shared weights between very different outputs, the outputs which produce significantly larger gradients would "overwrite" the gradients of the smaller ones.

1) Critic Network: In case of CC, the aim of the critic network is to minimize the sum of the squares of the difference for each of the predicted long-term averages and the empirically found averages for each reward metric (to learn to make accurate predictions about the future reward):

$$\begin{aligned} l_{\text{v},t} = & (R_{\text{received},t} - v_{\text{received}})^2 + (R_{\text{sent},t} - v_{\text{sent}})^2 \\ & + (R_{\text{duration},t} - v_{\text{duration}})^2 \end{aligned} \tag{1}$$

One apparent issue of using RL for CC is that in RL one usually uses a fixed parameter γ that determines the influence of future rewards. However, in CC the larger the window is,

the more packets are sent and received per unit of time. Thus, the use of a single fixed parameter γ incurs the problem that larger CWs result in shorter future intervals to be considered for CC. The reason is that with a large window more packets and thus more actions and rewards are handled per unit of time. Our aim is to look into the future for a more-or-less constant time span independent of the current size of the window. In other words, we want to define γ so that it reflects the number of packets in the current window, so that it always looks in the future for approximately one round-trip time (as one full window corresponds to one round-trip time).

An EWMA with a γ of $\frac{2}{n+1}$ has similar characteristics to a regular moving average with a window size of n. Furthermore, with sufficiently large n this means that the first n data account for $\approx 86\%$ of the total weight in the EWMA. Thus, the idea is to make n the number of packets in the current window and so we define the factor γ_t as a function of the current window size w_t and the expected amount of bytes to be sent per time step (the window divided by the expected number of bytes per packet is the expected number of packets in this window) as

$$\gamma_t = \frac{2}{1 + \max(\frac{w_t}{V_{\text{sent}}(s_t;\theta)}, 10)} \tag{2}$$

We use 10 as a minimum (as proposed by [2]) because when γ becomes too large, future rewards are not considered sufficiently anymore and the ensuing variability causes instability.

2) Actor Network: Each time an action a_t is requested, a change to the CW is sampled from the current normal distribution defined by the parameters μ and σ. However, we define that the window can never decrease below 1: $w_{t+1} = \max(w_t + a_t, 1)$. At the beginning of a flow, the window starts as 1 too.

The actor network aims to minimize the loss function

$$\begin{aligned} l_{\text{a},t} = &- \log\left(\pi\left(a_t \mid s_t; \theta_\text{a}\right)\right)\left(U_{\text{measured},t} - U_{\text{expected},t}\right) \\ &- \beta H\left(\pi\left(s_t; \theta_a\right)\right) \end{aligned} \tag{3}$$

where U can be any reward function defined based on some of the (long term averages of the) reward metrics. In other words, the actor network considers if an action improved the actual experienced reward compared to the expected one and adjusts the neural network accordingly.

C. Reward function

We use a reward function that is similar to PCC's, whose convergence for multiple senders has been proven [2].

$$U_t = \text{throughput} - \alpha \times \text{lost throughput} \tag{4}$$

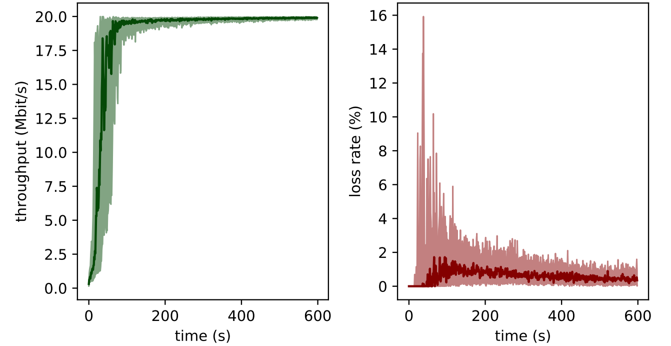
α is a parameter that determines how strongly packet loss is discouraged (throughput/packet loss tradeoff factor). We can define this reward function as follows using the previously defined reward metrics:

$$U_{\text{measured},t} = \frac{R_{\text{received},t}}{R_{\text{duration},t}} - \alpha\frac{R_{\text{sent},t} - R_{\text{received},t}}{R_{\text{duration},t}} \tag{5}$$

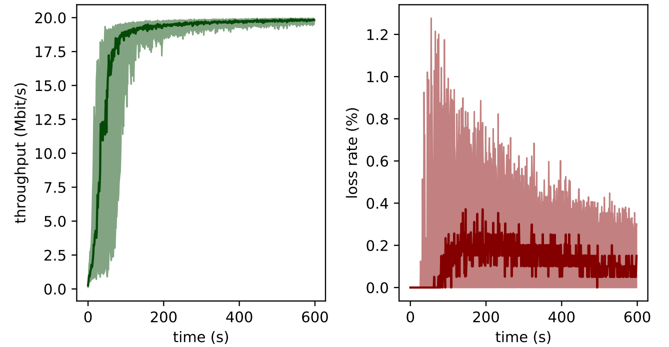
The corresponding expected reward $U_{\text{expected},t}$ is also defined using the estimates of the value networks and it can be determined if an action caused performance to be lower or higher than expected.

IV. Evaluation

We implemented Rax as an extension to both the ns-2 network simulator and Remy [10]. In the following evaluation we use a classical dumbbell network topology with one receiver and one or more senders and a shared bottleneck link of 20 Mbit/s with a buffer of $\frac{1}{10}$ bandwidth delay product and a round trip delay of 50 ms. In all figures in the evaluation we use data from at least 50 simulations. In the following figures we depict the median in dark color and 25th and 75th percentile band in a light color.



(a) One sender, throughput/packet loss tradeoff α of 1

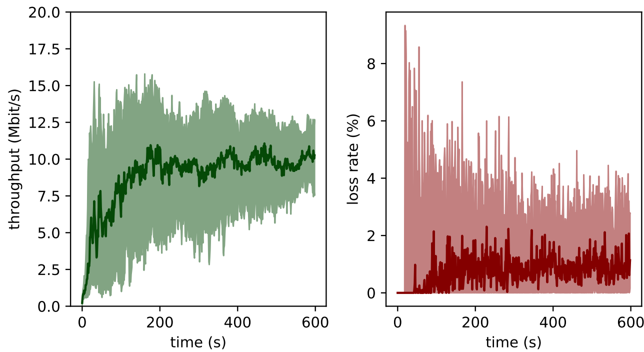


(b) One sender, throughput/packet loss tradeoff α of 4

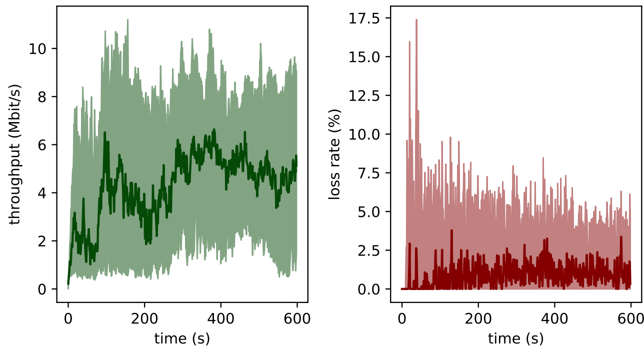
Figure 5: One sender with varying throughput/packet loss tradeoff α.

The tunable tradeoff between bandwidth and packet loss rate (α) has a clear impact on both the highest spike of packet loss that is produced in the very beginning of the online learning process as well as on the median packet loss rate later on in the learning process (Figure 5). However, the stronger packet loss is penalized, the longer it takes for the sender to reach the maximum bandwidth on the link.

While Rax generally does not exhibit much variance after a few minutes of training when it is the only flow on a link, fluctuations and training time needed to achieve good performance increase when having two or four concurrent senders (Figure 6). We attribute this to the fact that on a shared link with n senders each can only get $\frac{1}{n}$th of the bandwidth and to the increasingly dynamic environment.



(a) Two senders



(b) Four senders

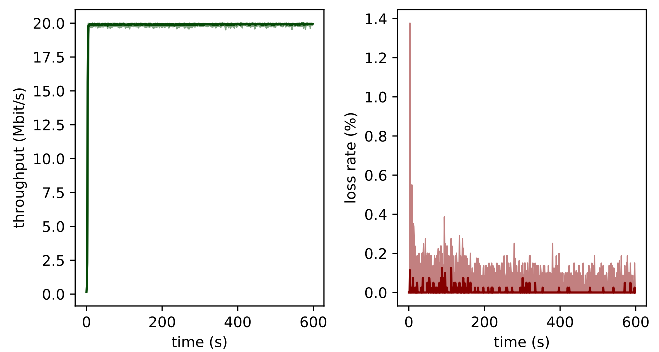
Figure 6: Without pre-training: Two (6a) and four (6b) senders with a bandwidth/packet loss tradeoff α of 2.

To verify if pre-training is generally feasible, we used the neural networks that were produced after each 600-second flow shown in Figure 6 and started flows of equal length with the pre-trained neural networks. The flows seem to simply resume where they ended training in the flow before (Figure 7). This does not seem like a remarkable finding, however, it is quite unexpected: For instance, in the case of one sender on a link, the neural network starts with random weights and discovers that it achieves optimum reward at a window that results in 20 Mbit/s of bandwidth. Then the sender constantly stays in this state and one would expect it to slowly "forget" what it learned in the very beginning when it started the flow.

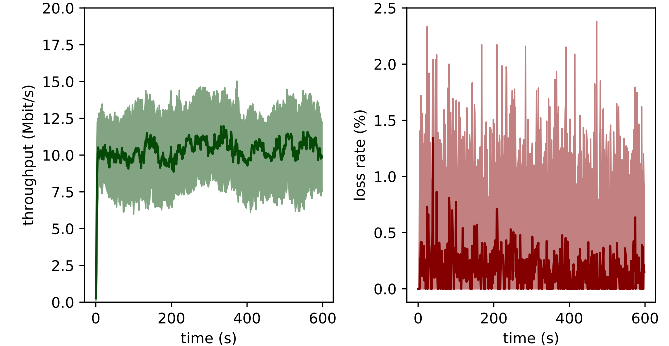
We also verify if stochastic packet loss on the link has an influence on Rax. For this, we randomly dropped 1% of packets on the link and it seems as if stochastic loss does not substantially affect Rax. This can be attributed to the fact that the RL realizes that a certain minimum packet loss rate is "normal" and cannot be avoided and thus Rax can learn to distinguish packet loss that is caused by congestion from packet loss that is purely stochastic (as for example caused by errors on the physical layer).

A. Comparison with other Congestion Control algorithms

For reasons of comparability we take the same scenario of a 20 Mbit/s shared bottleneck with a small buffer of $\frac{1}{10}$



(a) One sender



(b) Two senders

Figure 7: With pre-training: One (7a) and two (7b) senders initialized with a neural network from a previous flow with a throughput/packet loss tradeoff α of 4. Rax finds the correct bandwidth faster when comparing with the untrained case (6a).

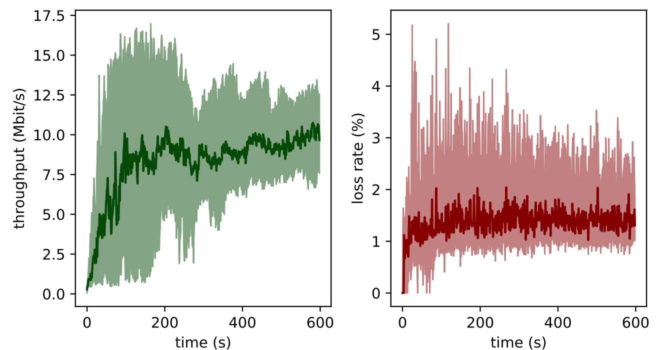


Figure 8: Stochastic packet loss of 1%: Two senders on a link with a bandwidth/packet loss tradeoff α of 4.

bandwidth delay product to create a challenging scenario. Surprisingly, New Reno performs far better than more modern CC algorithms (Figure 9). We attribute this to the fact that we chose a small buffer to create a challenging scenario and that Cubic and Compound are not optimized so well for this scenario but rather for long fat networks [8]. Rax achieves a significantly larger total throughput on average though with

a couple of outliers which become more significant with two concurrent senders (Figure 10); when inspecting the raw data, it becomes apparent that these outliers occur usually when the neural network diverges in the first seconds of a flow.

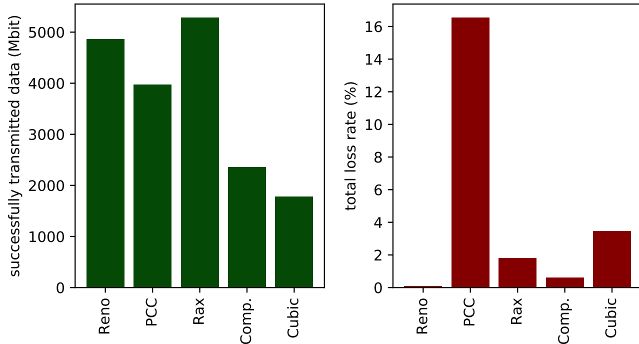


Figure 9: Total throughput and loss rate ($\frac{\text{all bytes lost}}{\text{all bytes sent}}$) for two senders of Cubic/Compound/New Reno, PCC and Rax with a throughput/packet loss tradeoff α of 4 without pre-training. For a flow of 600 s the maximum possible throughput is 12 000 Mbit. We do not show Remy as it easily exceeds all other CC variants in case it is trained for a specific scenario and performs rather badly in case it is not and thus it is not useful to be included in this comparison.

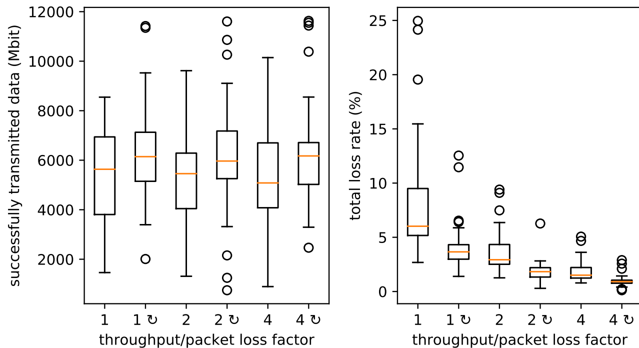


Figure 10: Total throughput and loss rate ($\frac{\text{all bytes lost}}{\text{all bytes sent}}$) for two senders for Rax. The arrow symbol indicates that the flows use a pre-trained neural network. For a flow of 600 s the maximum possible throughput is 12 000 Mbit.

V. Discussion

In this paper we show that it is possible to use Deep RL to learn a per ACK CC strategy online. In order to achieve this we had to modify existing RL algorithms to accommodate the specific needs of CC such as the inherent delay and proposed Partial Action Learning, a new method that allows Deep RL with delayed and partial rewards.

Our results demonstrate that our proposed concept can perform online learning of CC and can outperform existing CC algorithms in certain scenarios. We also show that with an increasing number of senders the RL becomes significantly more challenging and time consuming because the senders cannot be sure that the change in bandwidth/packet loss is actually caused by their own actions and not by other senders. Regarding online learning, during the implementation of Rax it became apparent that a neural network consisting of LSTM cells is not feasible but GRUs have to be used instead. LSTM cells keep an internal state that gets unstable when it is used for long training episodes, as for each received ACK the LSTM state has to be updated. Previous work using Deep RL uses up to 50 million training steps for offline learning [7] while we generally use significantly less (a 20 Mbit/s link can carry a maximum of up to 1.2 million packets of 1250 bytes size in 10 minutes). When sharing the link between several senders, each sender receives only a fraction of these packets for training. For example, with two senders on one link (Figure 6b) one sender can receive up to a maximum of 600 000 packets during a ten minute flow for training. We argue that the reason for Rax needing fewer training samples than other RL tasks is that in CC rewards are steadily coming in, while for other tasks, such as video games, commonly rewards are only received sparsely (e.g., one has to wait until the game is over to determine if one won or lost).

Acknowledgement

The Titan Xp used for this research was donated by the NVIDIA Corporation.

References

[1] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[2] Mo Dong, Qingxi Li, Doron Zarchy, Philip Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, pages 395–408, 2015.

[3] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *NSDI*, pages 343–356, 2018.

[4] Pierre Geurts, Ibtissam El Khayat, and Guy Leduc. A machine learning approach to improve congestion control over wireless computer networks. In *ICDM*, pages 383–386. IEEE, 2004.

[5] Yiming Kong, Hui Zang, and Xiaoli Ma. Improving TCP Congestion Control with Machine Intelligence. In *NetAI*, pages 60–66. ACM, 2018.

[6] W. Li, F. Zhou, K. R. Chowdhury, and W. M. Meleis. QTCP: Adaptive Congestion Control with Reinforcement Learning. *IEEE Transactions on Network Science and Engineering*, pages 1–1, 2018.

[7] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, pages 1928–1937, 2016.

[8] Ryo Oura and Saneyasu Yamaguchi. Fairness comparisons among modern TCP implementations. In *WAINA*, pages 909–914. IEEE, 2012.

[9] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[10] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 123–134. ACM, 2013.

[11] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for Internet congestion-control research. pages 731–743, 2018.