TCP Rax: Deep Reinforcement Learning for TCP Congestion Control*

ABSTRACT

A major challenge in congestion control is to find a set of rules that dynamically adapt the congestion window to current and anticipated future network conditions to maximize a specified utility function, e.g. maximum throughput or minimum packet loss.

This paper proposes Reactive Adaptive eXperience based congestion control (TCP Rax), a new method of congestion control that uses online reinforcement learning to maintain an optimum congestion window with respect to a given utility function. We use a neural network based approach that can be initialized either with random weights (without prior knowledge) or with a previously trained neural network to improve stability and convergence time. The processing of rewards in congestion control depends on the arrival of acknowledgments, inducing loop delays that may lead to oscillations. As a remedy we propose a specific formulation of neural network based reinforcement learning that supports the proactive processing of delayed and partial rewards.

We show that our method converges to a stable, close-to-optimum solution within the order of minutes and outperforms existing congestion control algorithms in typical networks. Thus, this paper demonstrates that neural network based reinforcement learning without any prior knowledge can feasibly be done on-line and can compete with human-designed congestion-control schemes such as New Reno and Cubic.

CCS CONCEPTS

•Networks →Transport protocols; •Theory of computation →Multi-agent reinforcement learning; •Computer systems organization →*Neural networks;*

KEYWORDS

congestion control, machine learning, reinforcement learning, artificial neural networks

ACM Reference format:

. 2017. TCP Rax: Deep Reinforcement Learning for TCP Congestion Control. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* 10 pages.

DOI: 10.475/123_4

1 INTRODUCTION

Recent advances in neural network based reinforcement learning (RL) raise the question if it is possible to use these frameworks to model end-to-end congestion control. In general, in congestion control each sender aims to maximize an objective function that usually comprises of (1) sending as many packets as possible (2) with as little delay as possible (3) while minimizing the amount of lost packets. As, for instance, for some applications overall throughput is more important than little packet loss, each application can define its preference using a custom utility function. Clearly, it is also desirable from an overall point of view that all senders use available resources in a way that is fair to the others: When several senders share an Internet link under the same conditions, they should receive an equal share of the reward. To achieve fairness, it is necessary that senders only use utility functions that (1) converge to a stable equilibrium and (2) provide each sender with a fair share of his subgoals.

In an ideal machine learning based congestion control method, each sender uses a set of observed environment conditions to determine which action he should take to maximize his reward in the future. The environment conditions can be any metrics that the sender can obtain from the network, for example the mean round-trip time of the last received packets, the packet loss rate etc. An action is a change to the sending rate: For example, if a sender perceives an increase in the loss rate it might be advisable to lower the sending rate. To modulate the sending rate one can either (1) change the sending rate itself at certain points in time or (2) maintain a congestion window that indicates the number of packets that can be in flight at a certain point in time. It seems preferable to maintain a congestion window, as one cannot be certain that the operating system (OS) can maintain a certain constant sending rate for each application on one machine, as the OS has to schedule different applications and share the network interfaces between them.

While congestion control seems to be a well suited problem for reinforcement learning, there are a couple of challenges and differences from most other reinforcement learning problems:

- For instance, when playing a video game, if one pushes a button, one can see the consequences of an action immediately. However, in congestion control, the rewards are always **delayed** by one round-trip time.
- Due to the delay, by the time an action receives its reward, probably other actions have already been carried out in the meantime. Thus, contrary to classical RL, **action and rewards are not synchronized**.
- **Actions and rewards are not atomic**: If an action causes three packets to be sent, we consider these three packets being sent three *partial actions*. In case these packets are transmitted correctly, the receiver will send back three acknowledgements. Each of these acknowledgements is a *partial reward* that already unveils new information on the current state of the network and enables the sender to take an action based on this new information. However, only if all partial rewards have been received, the sender can

*Produces the permission block, and copyright information

WOODSTOCK'97, El Paso, Texas USA

DOI: 10.475/123_4

assemble the overall reward of the corresponding action and can perform an update of the underlying reinforcement learning logic.

- The **number of actions and rewards** per unit of time is **variable**: If the congestion window is larger, more packets are sent and more acknowledgements are received, which means that the reinforcement learning algorithm gets more training data the larger the window is. This is problematic as it means that – without proper adjustments – the experience made at a larger congestion window will "overwrite" the experience made at a smaller window as with the larger window simply more training data arrives and it is thus going to be overemphasized.

To address these problems, we develop a new formulation of reinforcement learning called *Partial Action Learning* (PAL). PAL is a superset of reinforcement learning: If one uses PAL for a learning problem without delay, asynchronicity and partial actions/partial rewards, one gets classical reinforcement learning.

While using PAL to train an optimum congestion control for a specific range of network scenarios in an off-line fashion is possible, it is something that has already been done previously in an approach called *Remy* [13]. The only potential advantage that PAL could provide here is increased training speed. Thus, we want to show that it is not only possible to learn congestion control by using machine learning but that it is even possible to do so on-line and without any preknowledge about the network environment.

Using a utility function that encourages high throughput and discourages packet loss (adapted from [4]) we see that our machine learning approach can learn congestion control that maximizes this objective given a couple of minutes of time (without any pre-training) 1.

After spending up to a few minutes with suboptimal performance during initial exploration (if no pre-training is used), our approach can compete with established algorithms and achieve superior throughput and packet loss of less that 1%, while still gradually improving and adapting its way of performing congestion control.

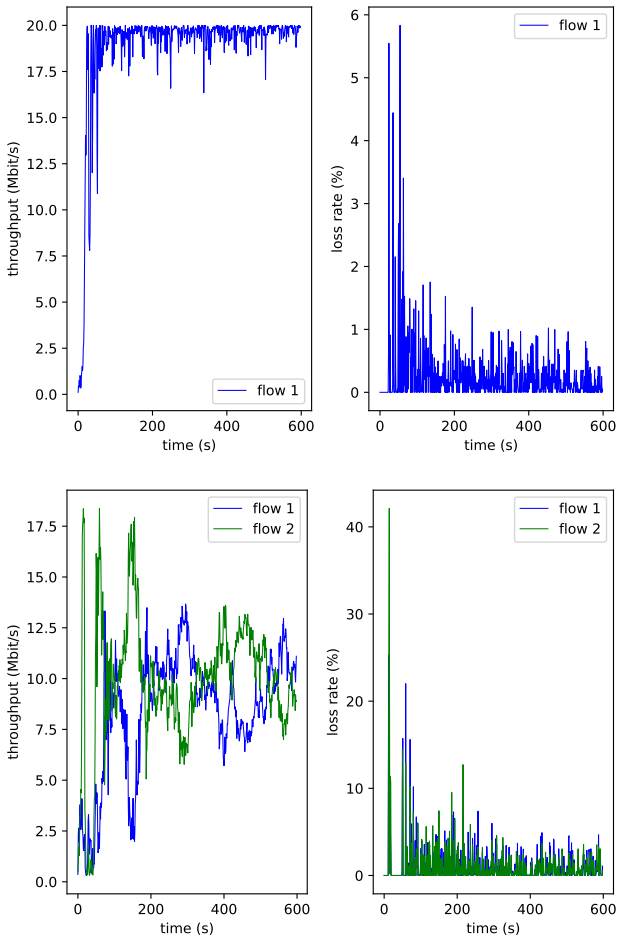

Figure 1: The throughput and the loss rate when sharing a bottleneck link of 20 Mbit/s with a two-way end-to-end delay of 50 ms and a very small buffer of $\frac{1}{10}$ bandwidth delay product between 1 and 2 senders (from top to bottom). The time needed for convergence increases with more senders, which is caused by the fact that the more senders use the link, the fewer because each sender receives fewer packets per unit of time for training and because the environment gets more complex. It is interesting to note that in the case of two senders, both flows complement each other and quickly adapt to the sending rate of the other flow.

2 RELATED WORK

2.1 Congestion Control

Congestion control has been implemented in TCP since the 1980s as a series of congestion collapses dramatically decreased applications' throughput in the Internet, which lead to the development of the Tahoe and Reno algorithms [8]. These approaches as well as most others maintain a congestion window which stands for the maximum amount of data that are allowed to be traveling in the network without having been acknowledged by the receiver. In general, receiving acknowledgements of data that were successfully transmitted leads to the congestion window being increased while packet loss (or an increase in delay as in the Vegas algorithm [2]) lead to the congestion window being lowered according to a set of fixed rules. Algorithms such as Cubic and Compound [7, 12] improve congestion control specifically with respect to so-called long fat networks with a high round-trip time and bandwidth. Proportional rate reduction [5] aims to make the reduction of the rate that occurs in case of congestion more smooth and steady in time to avoid the bursty behavior that occurred upon loss in previous TCP congestion control variants. On the other hand, BBR [3] does not use a set of fixed rules such as previous congestion control algorithms but instead estimates a model of the network path by using measurements of RTT and throughput and then adjusts its sending rate so that it uses the maximum bandwidth according to its network model while trying not to fill up queues. A similar approach is PCC [4], which also uses measurements to find an optimum sending rate with respect to a defined utility function (e.g. maximizing throughput while minimizing packet loss). However,

PCC doesn't maintain a congestion window but instead modulates the sending rate, using one constant sending rate for a certain time span.

Besides the aforementioned human-designed algorithms there have been attempts to use machine learning to improve congestion control. Geurts et al. [6] train a classifier off-line to determine whether packet loss is caused by congestion or by the link layer. If it is caused by the link layer, TCP does not alter the window. Otherwise – if a packet was lost due to congestion according to the classifier – it uses a traditional TCP congestion control algorithm. Winstein and Balakrishnan [13] train a machine learning solution called *Remy* that finds an optimum congestion control for a given range of network parameters. For instance, one could find an optimum congestion control for networks with an RTT of 50-100 ms and link speeds of 100-500 Mbit/s. After a lengthy training procedure Remy finds an optimum congestion control algorithm for the specified networks on a per acknowledgement basis (updating the window upon receiving an acknowledgement alike traditional congestion control algorithms).

Our goal is to find an optimum congestion control algorithm on a per acknowledgement basis similar to Remy that, however, can be trained on-line. Such a solution could be used in a purely on-line fashion, in a off-line fashion like Remy or a combination of both: One could pre-train a generic congestion control algorithm that works reasonably well for every network that then gets refined during on-line training according to the current network circumstances.

To this end we propose Partial Action Learning which is based on the Asynchronous Advantage Actor Critic framework [10], which has been demonstrated to be able to learn to play a wide range of video games and commonly outperform human players. In particular, it is a good choice for congestion control as it has been proven to deliver good performance and high training speed for playing video games and also for networking related tasks such as Adaptive Bitrate Algorithms [9] TODO: This motivation doesn't sound convincing yet, does it?. Furthermore, as we will show, it is conceptually possible to use it for on-line training although – to our knowledge – this has not been done until now.

2.2 Actor Critic Learning

The *Partial Action Learning* (PAL) (see 3.1) framework is based on the Actor Critic framework for neural networks proposed by Mnih et al. [10], which we outline (with a focus on intuition over rigor) in this section.

There are two neural networks, the Actor Network and the Critic Network. Given a state, the Actor Network outputs what it deems to be the optimum action to perform in that certain state. The Critic Network estimates what long-term reward can be expected in this state. So an action is considered good if it achieved a long-term reward that is higher than the long-term reward expected by the Critic Network and it is considered bad if the reward was lower than expected. The long-term reward is implemented as an exponentially weighted moving average of future rewards. So if a high reward can be achieved right now this is more favorable than if it can be achieved in the future. However it can also be beneficial to get a low reward now and instead get a very large one in the future.

2.2.1 Critic Network. The Critic Network outputs the expected long-term reward $V(s_t;\theta_\mathrm{v})$ given a state s_t at time step t and the parameters (neural network weights) of the critic network θ_v.

With r_t being the reward that was received at time t and γ being a factor with $0 < \gamma \leq 1$, which stands for the influence that future reward has on the moving average, we define the expected long-term reward at time step t as

$$R_t = \gamma r_t + (1-\gamma) R_{t+1},$$

One can see that to compute R_t at time step t one has to look infinitely far into the future to get all future rewards needed to compute the exponentially weighted moving average at time step t. Thus, one usually only carries out t_max steps (for example 20) and then uses an estimation of $R_{t_\mathrm{max}+1}$ provided by the critic network as the continuation TODO: Clearer than before?. The only exception from this is when the task to carry out is over before reaching t_max (e.g. the TCP flow is over). In this case, one can either use a prediction from the critic network as a continuation as usually or simply let the moving average end without continuation (ignoring the term weighted with $(\gamma - 1)$ at the last step).

The loss function (not in the sense of packet loss but in the sense of the loss of a machine learning problem), which the critic network tries to minimize at each time step, is the square of the difference of the actual long-term reward received and the expected long-term reward

$$l_{\mathrm{v},t} = \left(R_t - V(s_t;\theta)\right)^2.$$

2.2.2 Actor Network. The Actor Network outputs a probability distribution from which the action a_t at time step t is randomly sampled. We use the mean μ and the standard deviation σ to parametrize a normal distribution. The main idea is that the network learns to output the right mean at the right time step to maximize the future reward and that it uses the standard deviation to try out new actions, which could yield a better than expected reward.

Each time we take an action a_t we also let the critic network make an estimation $V(s_t;\theta)$, which we abbreviate as v_t, given the current neural network weights θ and the current state s_t. The loss function of the actor network can now be described as this: How likely was it (on a logarithmic scale) to take action a_t given the probability distribution at time t and how beneficial was this action when comparing to the critic's estimation? For example, if an action was very successful and highly unlikely, the actor network's are updated to make actions like this more likely in the future.

The actor also aims at increasing the entropy of the probability density function $(\frac{1}{2}\log\left(2\pi e\sigma^2\right))$ to encourage exploration: Otherwise the actor could always output similar actions but never actually take the action that yields the best performance. We designate the entropy as H and introduce a parameter β which specifies magnitude of the influence of the entropy.

$$\begin{aligned} l_{\mathrm{a},t} = &-\log\left(\pi\left(a_t \mid s_t;\theta\right)\right)\left(R_t - v_t\right) \\ &-\beta H\left(\pi\left(s_t;\theta\right)\right). \end{aligned}$$

3 METHOD

3.1 Partial Action Learning

The key difference between PAL and previous approaches to RL is that in classical RL an action is always followed by a reward and a

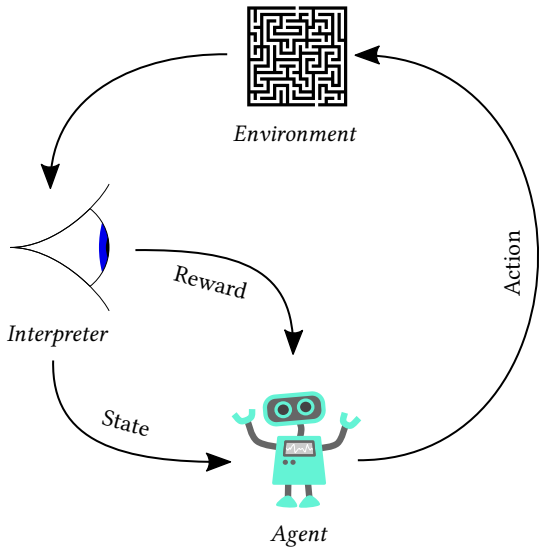

Figure 2: The classical reinforcement learning approach.[a]

[a]adapted from https://upload.wikimedia.org/wikipedia/commons/1/1b/Reinforcement_learning_diagram.svg

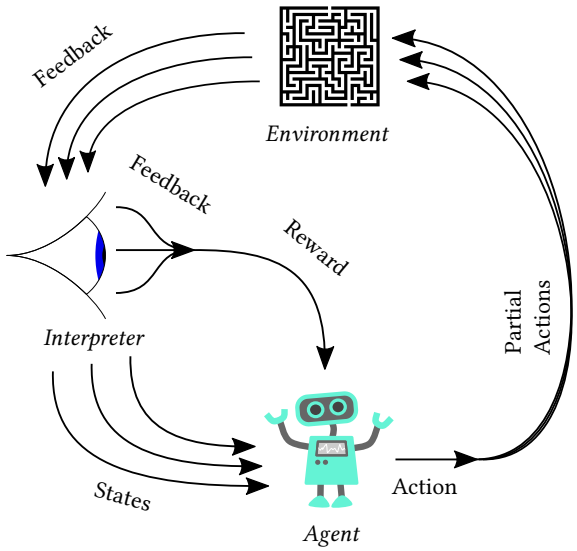

Figure 3: Partial Action Learning: An action consists of zero or more partial actions which trigger feedback upon interacting with the environment. Each feedback updates the state. The value and actor networks are updated upon receiving all feedback of one action.[a]

[a]adapted from https://upload.wikimedia.org/wikipedia/commons/1/1b/Reinforcement_learning_diagram.svg

reward is always followed by an action. In our proposed concept, however, it is possible to take new actions while previous actions haven't received their rewards yet.

Another major difference in PAL is that one action generates a number of partial actions (≥ 0) (see Figure 3). Each partial action generates feedback upon interacting with the environment. Upon receiving feedback for a partial action, the agent determines the current state and triggers a new action. When all feedbacks of one action were received, the agent combines them to form the reward and updates the critic and actor networks.

In Algorithm 1 (in the appendix) we show the code that runs in each of the agents (in the congestion control scenario, one agent corresponds to a sender). It is possible to have several agents which share a set of neural weights (which is usually done when performing off-line training [9, 10]) but one can also use separate weights for each agent, which is more realistic in case of congestion control in the Internet, as it is not sensible that different senders share a set of neural network weights over the Internet.

3.2 Congestion Control Specifics

The motivation for PAL is that classical reinforcement learning assumes that a reward follows an action and vice-versa (see Figure 2). TODO: Too much repetition of this? However, in the case of congestion control, it is desirable to perform a new action without having received a reward for the previous action. For example, imagine that we receive an acknowledgement. Now we have to sample an action and wait for the reward, and it takes one RTT until the first acknowledgements (which make up the reward) are received. If we now receive another acknowledgement before receiving the reward of the previous action, classical RL cannot handle this situation because of the asynchronicity of actions and rewards. Thus the Asynchronous Actor Critic framework as described by [10] cannot be applied to congestion control as it assumes that actions and rewards are synchronized and so we have to use Partial Action Learning (see subsection 3.1). We describe the overall workings of PAL for congestion control using a petri net (see Figure 4).

To use PAL for congestion control we first have to define the correct semantics for this specific use case and we have to explicitly state how the state, reward etc. are defined. Furthermore, we have to define how the actor and critic network explicitely work in case of congestion control. In the following a time step t corresponds to the reception of an acknowledgement. The beginning of the flow, before any packet is sent, corresponds to time step 0.

$\mathbf{s_t}$ The state describes the current "congestion state". The following features are included in it:

- the time between the last two packets that were sent
- the time between the last two packets that were received
- the RTT of the last received packet
- whether the last packet was lost where 1 signifies that the last packet was lost while 0 indicates that it was received correctly
- the current congestion window as a real number

Of each of these features a exponentially weighted moving average with a factor α of $\frac{1}{8}$ and one with an α of $\frac{1}{256}$ (the concept of using ewmas with these specific values of α was first introduced in *Remy* [13]), which makes a total of 10 features. Each time an acknowledgement is received, the

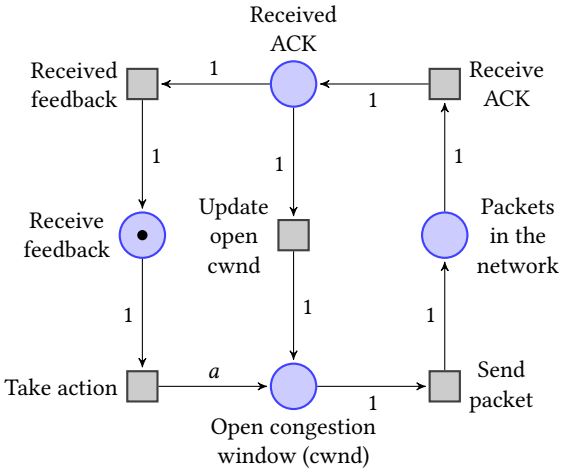

Figure 4: A petri net describing the congestion control mechanism. We start with one token in the state *Received feedback*, which means that we can take an action right in the beginning. The concept is the following: If there is at least one token in the *Open congestion window* send a packet (for the sake of simplicity in this example we assume that the congestion window is in packets and not bytes). The *Open congestion window* signifies the congestion window minus the packets that are currently unacknowledged. The token now goes to the network and after some time the acknowledgement (ACK) (or negative-acknowledgement (NACK) or timeout if the packet gets lost) for the packet is received in the *Received ACK* state. From here one token goes to the *Open congestion window* meaning that when we get an ACK, NACK (or timeout) we can send another packet. Furthermore the state *Received feedback* receives a token upon receiving an ACK/NACK/timeout. *Take action* adds a tokens to the *Open congestion window*, where a is a real number (possibly also negative); thus in each state there can also be a real number of tokens (e.g. 2.34 tokens are possible in the *Open congestion window*). However, we define that the congestion window can never be smaller than 1, which means that the sum of all tokens in the *Open congestion window.*, *Received ACK* and *Packets in the network* states can never be smaller than 1.

state is updated and the actor network is asked for the next action.

$\boldsymbol{a_t}$ Based on a given state and the history of previous states (because we use Gated Recurrent Units (GRUs) and thus can also consider previous states), the actor network returns an action a_t, which is a real number that stands for the change of the congestion window.

$\boldsymbol{r_t}$ The reward is a tuple of five reward metrics. For each reward metric there is also an output of the critic network that predicts the expected long-term average of this reward metric given the current state.

$\boldsymbol{v_t}$ The value is a tuple of the expected average reward estimated by the critic network (see subsubsection 3.2.1) for each of the reward metrics.

We use the following reward metrics:

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

$\boldsymbol{r_{\text{received},t}}$ is 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 at some point by the receiver).

$\boldsymbol{r_{\text{duration},t}}$ is the time between receiving the last packet and receiving this packet ("inter acknowledgement reception time") summed over all packets that the sender sent and that were not lost. Example: The sender sent three packets, of which two were not lost. The time between receiving the ACK of the first packet that wasn't lost and the previous one is 5 ms and the time between receiving the ACK of the second packet that wasn't lost and the previous ACK is 5 ms. So for this time step $r_{\text{duration},t}$ is 11 ms.

The overall structure of the neural network for both the value and the actor network is depicted in Figure 5. Having these reward metrics one can compose a variety of functions with their long-term averages. How these reward metrics can be used in reinforcement learning is described in subsection 3.3.

3.2.1 Critic Network. In the case of congestion control, the loss function $l_{\text{v},t}$ of the critic network is actually the sum of the squares of the difference for each of the expected long-term averages and the empirically found averages for each reward metric:

$$\begin{aligned} l_{\text{v},t} &= \left(R_{\text{received},t} - V_{\text{received}}(s_t;\theta_{\text{v}})\right)^2 \\ &+ \left(R_{\text{sent},t} - V_{\text{sent}}(s_t;\theta_{\text{v}})\right)^2 \\ &+ \left(R_{\text{duration},t} - V_{\text{duration}}(s_t;\theta_{\text{v}})\right)^2 \end{aligned}$$

One apparent issue of using reinforcement learning for congestion control is that in RL one usually uses a fixed parameter γ that determines the influence of future rewards. However, in congestion control the larger the window is, the more packets are sent and received per unit of time. Thus, if one used a single fixed parameter γ, one would have the problem that one would look into the future for a shorter time the larger the window is, as 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 no matter 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.

A ewma with a γ of $\frac{2}{n+1}$ has roughly the same distribution of forecast error as a regular moving average with a window size of n [11]. Furthermore, with sufficiently large n this means that the first n data account for $\approx 86\%$ of the total weight in the ewma [1].

Thus, the idea is to make n the number of packets in the current window and so we define the factor γ as a function of the current window size and the expected amount of bytes to be sent per time step as follows:

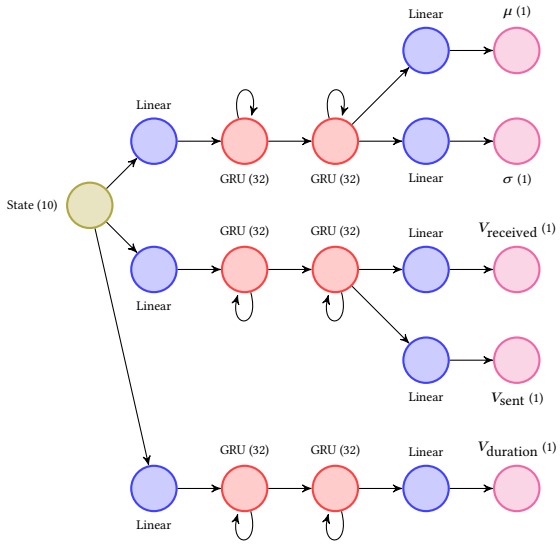

Figure 5: An overview of the complete neural network being used. Ocher stands for the input, blue for linear layers, red for GRU layers and purple for the outputs of the neural network. The numbers in parentheses next to each label stand for the width of that layer (i.e. the number of neurons except 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 similar quantities 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.

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

We use 10 as a minimum because when γ becomes too big, future rewards is not considered sufficiently anymore and as the variability increases instability ensues.

3.2.2 Actor Network. Each time an action a_t is requested, a change to the congestion window is sampled from the current normal distribution defined by the parameters μ and σ. However, we define that the window can never fall 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 minimizes the loss function

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

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

3.3 Utility function

We use a reward function that is similar to the one used by PCC [4].

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

α is a parameter that determines how strongly packet loss is discouraged. We can define this utility function as follows using the previously defined reward metrics:

$$U_t = \frac{R_{\text{received},t}}{R_{\text{duration},t}} - \alpha \frac{R_{\text{sent},t} - R_{\text{received},t}}{R_{\text{duration},t}}$$

The equivalent utility is also defined using the estimates of the value networks and it can be determined whether an action achieved a performance that was higher or lower than expected.

4 EVALUATION

We implemented TCP Rax as an extension to both the NS-2 network simulator [**?**] and as well as Remy [13]. In the following evaluation we use a classical dumbbell network topology with one receiver and one or more senders and a shared bottleneck link.

4.1 Tradeoff between throughput and packet loss

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 6. However, the stronger packet loss is punished, the longer it takes for the sender to reach the maximum bandwidth on the link.

4.2 Number of senders

While TCP Rax generally does not exhibit much variance after a few minutes of training when it is the only flow on a link, fluctuations clearly increase when having two or four concurrent senders Figure 7. We attribute this to the fact that on a shared link n can on average only get $\frac{1}{n}$ of the bandwidth and thus each sender receives fewer packets which he can use for training. Furthermore, as the environment becomes more dynamic with more senders, each sender has a harder time learning whether the reward changed due to his own actions or due to actions that were caused by other senders: If there is only one flow on a link, the flow "can be sure" that all changes in rewards were actually caused by his own actions.

4.3 Pre-training

To see whether pre-training is generally feasible, we used the neural networks that were produced after each 600 seconds flow shown in figure Figure 7 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. This does not seem like a remarkable finding, however, this is unexpected: For instance, in

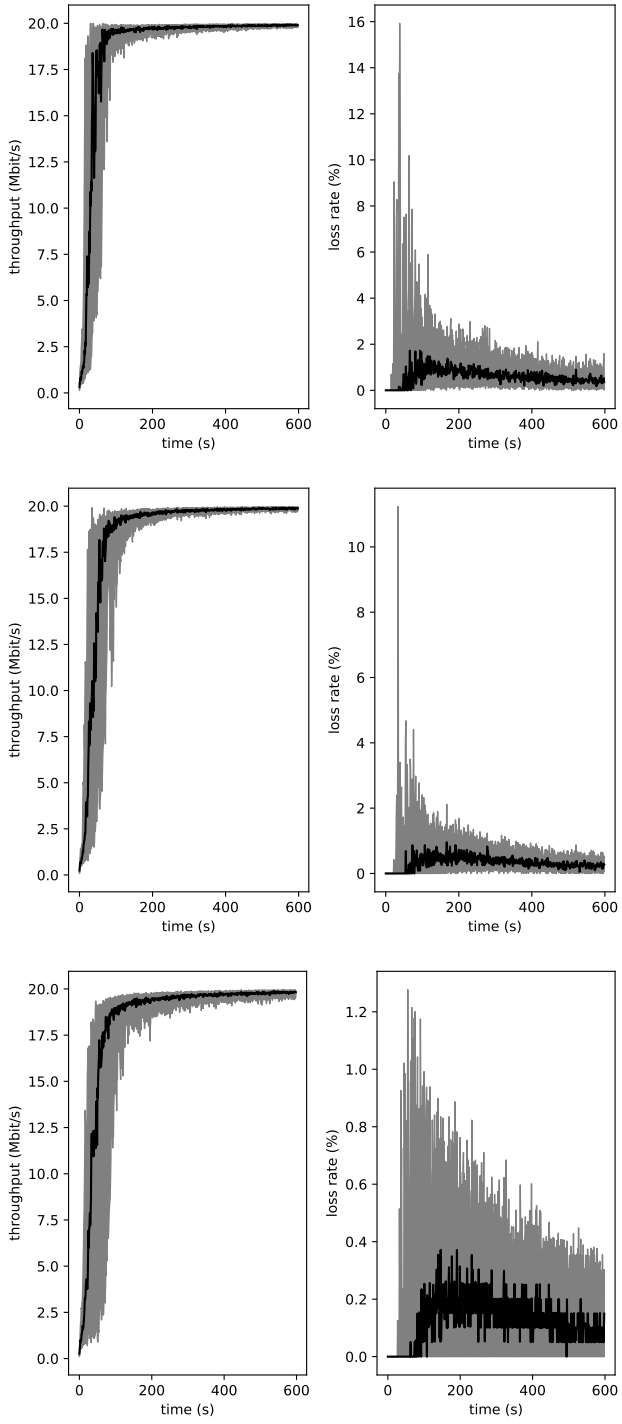

Figure 6: One sender with a bandwidth/packet loss tradeoff of 1, 2 and 4 (from top to bottom) on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. Median in black and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in gray.

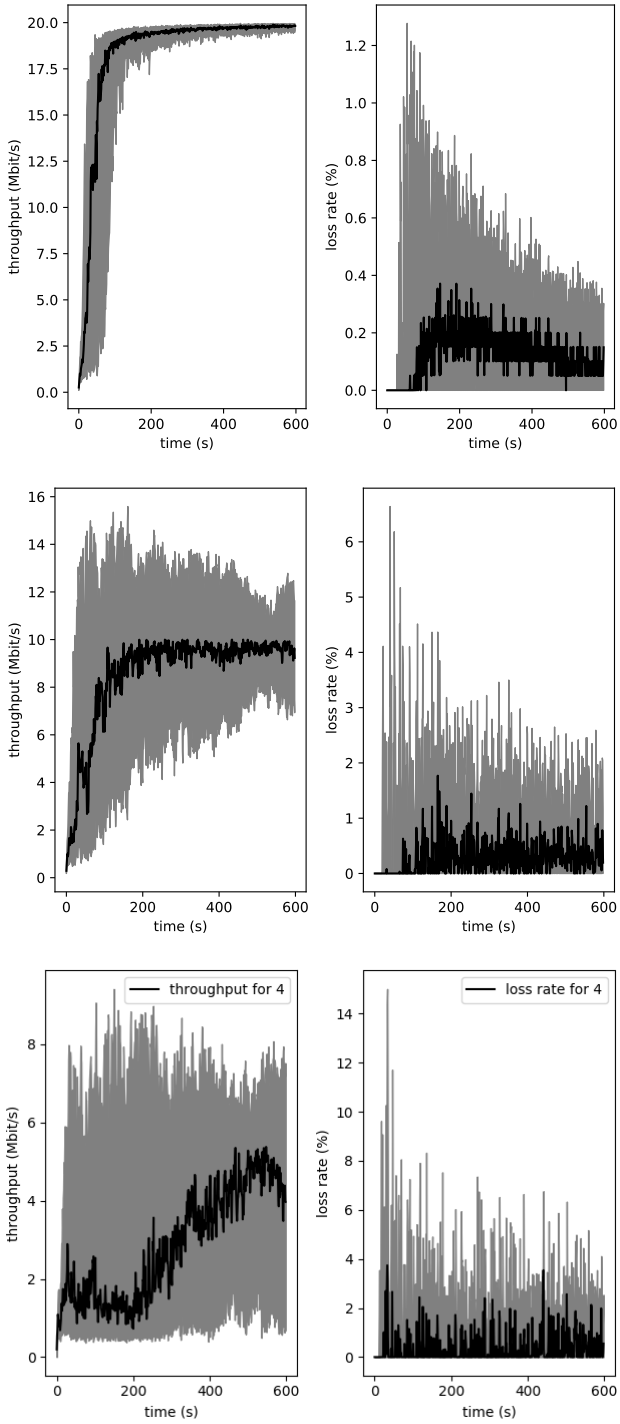

Figure 7: One, two and four senders (from top to bottom) with a bandwidth/packet loss tradeoff of 4 on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. Median in black and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in gray.

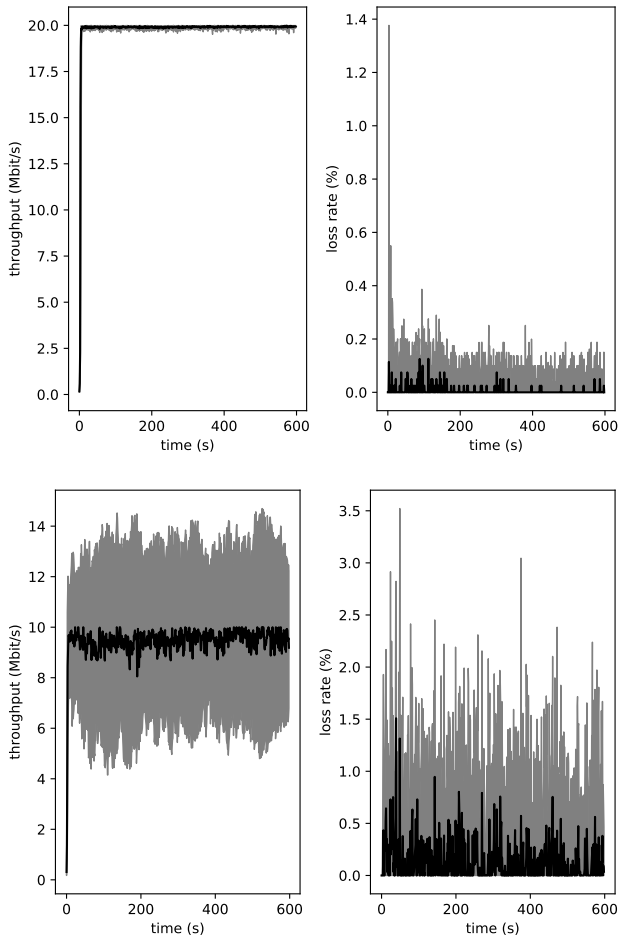

Figure 8: One and two senders (from top to bottom) that were trained with a bandwidth/packet loss tradeoff of 2 on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. Median in black and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in gray.

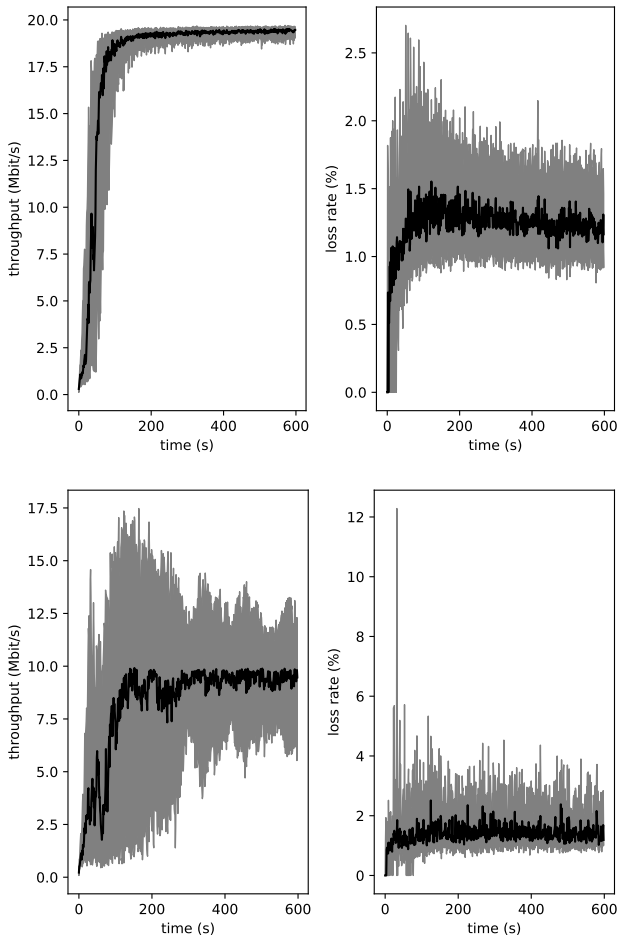

Figure 9: One and two senders (from top to bottom) on a link with a stochastic loss rate of 1%, a bandwidth/packet loss tradeoff of 2 on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. Median in black and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in gray.

the case of one sender on a link, the neural network starts with random weights and discovers that it achieves optimum utility 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. Thus it is interesting that after hundreds of seconds of only seeing one same state, the neural network still acts correctly when starting with a small congestion window again.

4.4 Stochastic Loss

To see whether pre-training is generally feasible, we used the neural networks that were produced after each 600 seconds flow shown in figure Figure 7 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. This does not seem like a remarkable finding, however, this is 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 utility 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. Thus it is interesting that after hundreds of seconds of only seeing one same state, the neural network still acts correctly when starting with a small congestion window again.

5 DISCUSSION

5.1 Best practices for on-line reinforcement learning

6 CONCLUSION

APPENDIX

Algorithm 1 Partial Action Learning – pseudocode for each agent. It is possible that the agents share the global weights θ_g, which is reasonable when learning off-line. Otherwise an individual copy of them is kept by each agent. By default, all weights are initialized randomly in the beginning but it is also possible to use a pre-trained neural network for initialization. As we use Gated Recurrent Units (GRUs) their output is stored in h and fed into them at the next time step.

loop
  $l_{\text{actions}} \leftarrow []$
  $l_{\text{states}} \leftarrow []$
  $l_{\text{values}} \leftarrow []$
  $l_{\text{rewards}} \leftarrow []$
  $l_{\text{values'}} \leftarrow []$
  $l_{\text{snapshots}} \leftarrow []$
  $l_{\text{hiddenStates}} \leftarrow []$
  $t \leftarrow 0$
  $s_0 \leftarrow$ initialState()
  $h \leftarrow$ zeros()
  $\theta \leftarrow \theta_g$
  repeat
    $l_{\text{values'}}$.append($V(s_t; \theta)$)
    if updateWeights? **then**
      $\theta_a \leftarrow \theta_g$
      $l_{\text{snapshots}}$.append(θ)
      $l_{\text{hiddenStates}}$.append(h)
    end if
    l_{states}.append(s_t)
    Sample a_t from the
      Actor Network's probability distribution
    l_{actions}.append(a_t)
    l_{values}.append($V(s_t; \theta_v)$) ▷ Possibly different from the value in $l_{\text{values'}}$ as the weights could have been updated in the meantime.
    $l_{\text{partialActions},t} \leftarrow$ partialActions(a_t)
    for all partial actions $a_{p,i,t}$ in $l_{\text{partialActions},t}$ **do**
      Take partial action $a_{p,i,t}$
    end for
    Wait for the next feedback $r_{p,j,t'}$
      where $t' \leq t$ and $0 \leq j \leq \#(l_{\text{partialActions},t'})$
    if all feedback of $a_{t'}$ was received **then**
      $r_{t'} \leftarrow$ reward w.r.t all feedback $r_{p,k,t'}$
        where $0 \leq k \leq \#(l_{\text{partialActions},t'})$
      l_{rewards}.append($r_{t'}$)
      if gotEnoughRewards? or the episode is over **then**
        computeGradients
      end if
    end if
    Generate s_{t+1} using $r_{p_{t'}}$
    $t \leftarrow t + 1$
  until reaching the end the episode
end loop

Algorithm 2 Partial Action Learning – procedure which computes and applies the gradients.

function computeGradients
  $t_{\text{end}} = \#(l_{\text{rewards}})$
  $\theta_{\text{backup}} \leftarrow \theta$
  $h_{\text{backup}} \leftarrow h$
  $\theta \leftarrow l_{\text{snapshots}}[0]$
  $h \leftarrow l_{\text{hiddenStates}}[0]$
  $R_{i+1} \leftarrow$ last element of $l_{\text{values'}}$
  for $i \leftarrow t_{\text{end}} - 1..0$ **do**
    $R_i \leftarrow (r_i + \gamma R_{i+1})(1-\gamma)$
    $a \leftarrow l_{\text{actions}}[i]$
    $s \leftarrow l_{\text{states}}[i]$
    $v \leftarrow l_{\text{values}}[i]$
    $d\theta \leftarrow -\frac{\partial \log(\pi(a \mid s;\theta_a))(R_i - v)}{\partial \theta} - \frac{\partial \beta H(\pi(s;\theta_a))}{\partial \theta} + \frac{\partial (R_i - v)}{\partial \theta}$
    $\theta_g \leftarrow \theta_g + d\theta$
  end for
  Remove first t_{end} elements from $l_{\text{actions}}, l_{\text{states}}, l_{\text{values}}, l_{\text{rewards}}, l_{\text{values'}}$
  Remove the first element from $l_{\text{snapshots}}$
  Remove the first element from $l_{\text{hiddenStates}}$
  $\theta = \theta_{\text{backup}}$
  $h = h_{\text{backup}}$
end function

Algorithm 3 Procedure which determines when to synchronize the neural network weights from the global weights in case of congestion control. We assume that the window at time step t is stored in a list l_{windows} and that the number of bytes sent in each time step is stored in l_{sent}. Alternatively to this procedure one can simply synchronize the weights after each $t_{\max}$ actions where $t_{\max}$ is an arbitrarily chosen integer greater than 0.

function updateWeights?
  $i \leftarrow 0$
  while $i < \#(l_{\text{actions}})$ **do**
    $w \leftarrow \lfloor l_{\text{windows}}[i] + l_{\text{actions}}[i] \rfloor$
    for $j \leftarrow i..\#(l_{\text{actions}}) - 1$ **do**
      $w \leftarrow w - l_{\text{sent}}[j]$
      if $w \leq 0 \wedge i + 1 \geq 10$ **then** ▷ at least 10 rewards
        $i \leftarrow j$
        break
      end if
    end for
    if $i = \#(l_{\text{windows}}) - 1$ **then**
      return true
    end if
  end while
  return false
end function

Algorithm 4 Procedure which determines when to start computing the gradients in case of congestion control. We assume that the window at time step t is stored in a list l_{windows} and that the number of bytes sent in each time step is stored in l_{sent}. Alternatively to this procedure one can simply compute the gradients when $t_{\max}$ rewards have been received where $t_{\max}$ is an arbitrarily chosen integer greater than 0.

function gotEnoughRewards?
  $w \leftarrow \lfloor l_{\text{windows}}[0] + l_{\text{actions}}[0] \rfloor$
  for $i \leftarrow 0..\#(l_{\text{rewards}}) - 1$ **do**
    $w \leftarrow w - l_{\text{sent}}[i]$
    if $w \leq 0 \wedge i + 1 \geq 10$ **then** ▷ at least 10 rewards
      return true
    end if
  end for
  return false
end function

REFERENCES

[1] Johan Boissard. 2012. Applications and uses of digital filters in finance. *Master of Science in Management, Technology and Economics. Swiss, Zurich, Department of Management, Technology and Economics, Swiss Federal Institute of Technology* (2012).

[2] Lawrence S. Brakmo and Larry L. Peterson. 1995. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on selected Areas in communications* 13, 8 (1995), 1465–1480.

[3] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *Queue* 14, 5 (2016), 50.

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

[5] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. 2011. Proportional rate reduction for TCP. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 155–170.

[6] Pierre Geurts, Ibtissam El Khayat, and Guy Leduc. 2004. A machine learning approach to improve congestion control over wireless computer networks. In *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*. IEEE, 383–386.

[7] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74.

[8] Van Jacobson. 1988. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, Vol. 18. ACM, 314–329.

[9] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 197–210. DOI: http://dx.doi.org/10.1145/3098822.3098843

[10] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*. 1928–1937.

[11] Steven Nahmias. 2009. *Production and operations analysis* (6th ed ed.). McGraw-Hill/Irwin, New York, NY.

[12] Kun Tan, Jingmin Song, Qian Zhang, and Murad Sridharan. 2006. A compound TCP approach for high-speed and long distance networks. In *Proceedings-IEEE INFOCOM*.

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