# TCP Rax: Deep Reinforcement Learning for Congestion Control


## ABSTRACT

A major challenge in TCP 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 and based on current network conditions. 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 *Partial Action Learning* (PAL), a specific formulation of neural network based reinforcement learning that supports the 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 online and can compete with human-designed congestion-control schemes such as New Reno and Cubic.



## KEYWORDS

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


## 1 INTRODUCTION

Recent advances in neural network based reinforcement learning (RL) that for the first time have the ability to learn complex tasks such as video games, raise the question if it is possible to use these frameworks to model end-to-end congestion control: As in RL, in congestion control each sender has a an observable state, performs actions and receives rewards for its actions according to the sender's objective function.

In general, in congestion control each sender aims to maximize an objective function that usually comprises one or more of the following subgoals: (1) Send 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. Besides aiming to fulfill these subgoals, it is also desirable from an overall point of view that all senders use available resources in a way that is fair to others: When several senders share an Internet link under the same conditions, they should receive an equal share of the available resources. To achieve fairness, it is necessary that senders only use utility functions that (1) do not give an unfair advantage over others (2) and converge to a stable equilibrium.

In a machine learning based congestion control method, each sender uses a set of observed environment conditions to determine which action it should take to maximize its reward in the future. The environment conditions are 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. or combinations thereof. An action is a change to the sending rate: For example, if a sender perceives an increase in the loss rate it might decide 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. As the operating system (OS) has to schedule different applications and share the network interfaces between them, maintaining a congestion window is appealing, because one cannot be sure that the OS can maintain a specific constant sending rate for each application on one machine.

While congestion control seems to be a well-suited task for reinforcement learning, there are a couple of unique aspects and differences that distinguish it from most other reinforcement learning problems:

- **Delayed rewards**: In congestion control, rewards are always delayed by one round-trip time. In most other areas where reinforcement learning is applied, if an action is performed, the consequences of that action become apparent immediately. Also, due to the delay, by the time an action receives its reward, probably other actions have already been performed. Thus, contrary to most RL applications, actions 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 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 assemble the overall reward of the corresponding action and can perform an update of the underlying reinforcement learning logic.
- **Variable number of actions and rewards**: 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. With the larger window more training data arrives and it is thus going to be overemphasized.

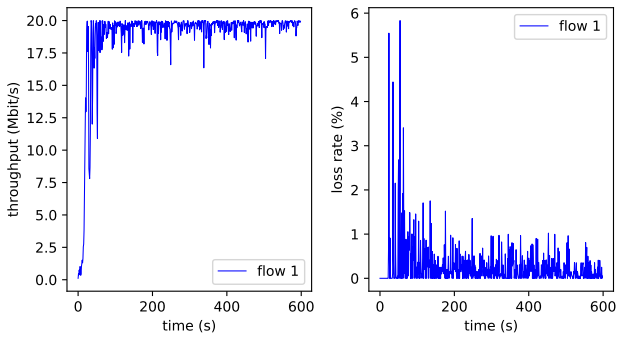
To address these problems, we develop a new formulation of reinforcement learning called *Partial Action Learning* (PAL). PAL is a superset of reinforcement learning: If one uses PAL for a learning problem without delay, asynchronicity and partial actions/partial rewards, one gets classical reinforcement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

While using PAL to train an optimum congestion control for a specific range of network scenarios in an offline fashion is possible, offline learning of an optimum TCP congestion control is something that has already been done previously in an approach called *Remy* [17]. The only potential advantage that PAL can provide over prior solutions 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 online and without any preknowledge about the network environment.

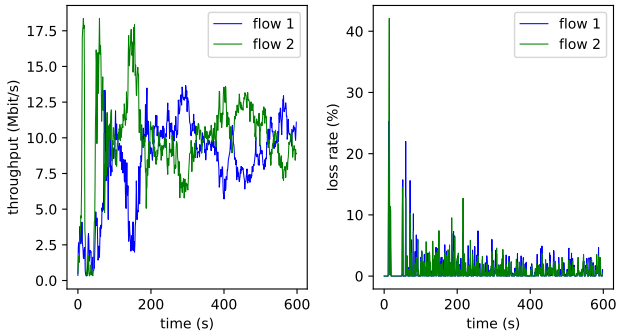
Figure 1 shows that using a utility function that encourages high throughput and discourages packet loss (adapted from [4]) our machine learning approach can learn congestion control that maximizes this objective given a couple of minutes of time (without any pre-training).

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 an average packet loss of less than 1%, while still gradually improving and adapting its way of performing congestion control.

Thus we make three contributions:




**(a) One sender**




**(b) Two senders**

**Figure 1: Throughput and 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 one (1a) and two (1b) senders . The time needed for convergence increases with more senders, which is caused by the fact that the more senders use the link, the fewer packets each sender receives per unit of time for training and because of the increasing complexity of the environment. In case of two senders, both flows complement each other and quickly adapt to the sending rate of the other flow.**

- TCP Rax: A reinforcement learning based TCP congestion control algorithm that helps senders to learn a good congestion control strategy autonomously. Our method acts on a per acknowledgement basis and can compete with existing congestion control algorithms.
- The ability to learn online, offline or a combination of both: offline pre-training with online learning.
- Partial Action Learning (PAL): A new method for deep reinforcement learning that allows dealing with delayed and partial rewards.

# 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 [10]. These approaches as well as most others maintain a congestion window which stands for the maximum amount of data that are allowed to be in flight in the network without having been acknowledged by the receiver. In most TCP congestion control algorithms, 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 [1]) leads to the congestion window being lowered according to a set of fixed rules. Algorithms such as Cubic and Compound [9, 16] 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. A different approach is taken by BBR [2], which does not use a set of fixed rules like previous congestion control algorithms. Instead it 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. [8] train a classifier offline to determine whether packet loss is caused by congestion or by the link layer in case of wireless links. 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. Shaio et al. [15] use online reinforcement learning to choose a sending rate for multimedia streaming. Winstein and Balakrishnan [17] 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 10-20 Mbit/s. After a lengthy training procedure of several CPU weeks, Remy finds an optimum congestion control algorithm for the specified networks on a per acknowledgement basis (updating the window upon receiving an acknowledgement like 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 online. Such a solution could be used in a purely online fashion, in a offline 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 online 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 [12], which has been demonstrated to be able to learn to play a wide range of video games and commonly outperform human players. Its use of a neural network enables it to even learn complex tasks. 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 [11].

## 2.2 Actor Critic Learning

Our proposed framework *Partial Action Learning* (PAL) (see 3.1) is based on the Actor Critic framework for neural networks proposed by Mnih et al. [12], 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 predicts what long-term reward can be expected in this state. So an action is considered good if it achieved an empirically experienced long-term reward that is higher than the predicted 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 calculated 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 a prediction of the long-term reward $v_t = V(s_t; \theta)$ given a state $s_t$ at time step $t$ and the parameters (neural network weights) of the critic network $\theta$.

With $r_t$ being the reward that was received at time $t$ and $\gamma$ being a factor 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 $R_t$ at time step $t$ as

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

To compute $R_t$ 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 $R_{t+1}$ relies on knowing $R_{t+1}$. Thus, the algorithm takes $t_{\max}$ steps (for example 20), waits for all the rewards and then calculates $R_t$ for each step and uses an estimate of $R_{t_{\max}+1}$ provided by the critic network as the continuation in the future. The only exception from this is when the task to execute ends 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 or alternatively let the moving average end without continuation i.e. ignore the term weighted with $(\gamma - 1)$ at the last step.

The loss function[1] (as defined by [12]), which the critic network tries to minimize at each time step, is the square of the difference of the actual experienced long-term reward received and the predicted long-term reward $l_{\mathrm{v},t}$ as

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

*2.2.2 Actor Network.* The Actor Network outputs a normal distribution of possible values for the action parametrized by a mean $\mu$ and a standard deviation $\sigma$. From this distribution each action $a_t$ at time step $t$ is randomly sampled. The main idea is that the neural network learns to output the right mean of the actions at the right time step to maximize future rewards 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 $v_t = V(s_t;\theta)$, given the current neural network weights $\theta$ and the current state $s_t$. The loss function of the actor network can be described as follows: How likely was it to take action $a_t$ given the probability distribution ($\pi$) at time $t$ and how beneficial was this action when comparing it to the critic's estimation? For example, if an action was very successful and highly unlikely, the actor network's weights are updated to make actions like this more likely in the future.

The actor also aims at increasing the probability density function's entropy ($H(\pi(s_t;\theta)) = \frac{1}{2}\log(2\pi e\sigma^2)$) to encourage exploration: Otherwise the actor could always perform similar actions but never actually take the action that yields the best performance. The actor network will aim to maximize the entropy and at the same time output the best possible action: If it realizes that the entropy is too large and bad actions are taken, it learns to lower the entropy. We designate the entropy as $H$ and introduce a parameter $\beta$ which specifies the magnitude of the influence of the entropy. The loss function of the actor network is defined as follows [12]:

$$\begin{aligned} l_{\mathrm{a},t} = &-\log\left(\pi\left(a_t \mid s_t;\theta\right)\right)\left(R_t - V(s_t;\theta)\right) \\ &-\beta H\left(\pi\left(s_t;\theta\right)\right). \end{aligned} \tag{3}$$

[1] not in the sense of packet loss but in the sense of the loss function of a machine learning problem, which one aims to minimize

# 3 METHOD

## 3.1 Partial Action Learning

The key difference between our approach PAL and previous approaches to RL is that in classical RL an action is always followed by a reward and a reward is always followed by an action. In our proposed concept, however, it is possible to take new actions while previous actions haven't been rewarded yet.

Another major difference in PAL is that one action generates a number of partial actions ($\geq 0$) (see Figure 2). Each partial action generates feedback upon interacting with the environment. Upon receiving feedback for a partial action, the agent determines the current state and performs 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 offline training [11, 12]) 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 Application to Congestion Control

To use reinforcement learning for congestion control in general and our modified approach PAL specifically we first have to define the correct semantics for this specific use case and define what makes up the state and the reward. Furthermore, we have to define how the actor and critic network explicitly 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, the time when the first packet is sent, corresponds to time step 0.

**$s_t$** The state $s_t$ describes the current "congestion state" defined by:

- the time between the last two acknowledgment that were received
- the RTT of the last received packet
- a loss indicator that shows whether the last packet was lost, where 1 denotes that the last packet was lost while 0 indicates that it was received correctly
- the current congestion window (as a real number, as actions by output by the actor network can be

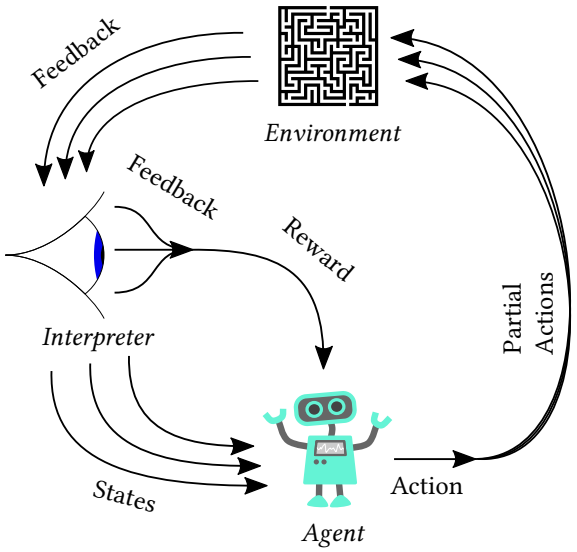



**Figure 2: Partial Action Learning: An action consists of zero or more partial actions which trigger feedback upon interacting with the environment. Each feedback updates the state. The value and actor networks are updated upon receiving all feedback of one action.**[a]

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

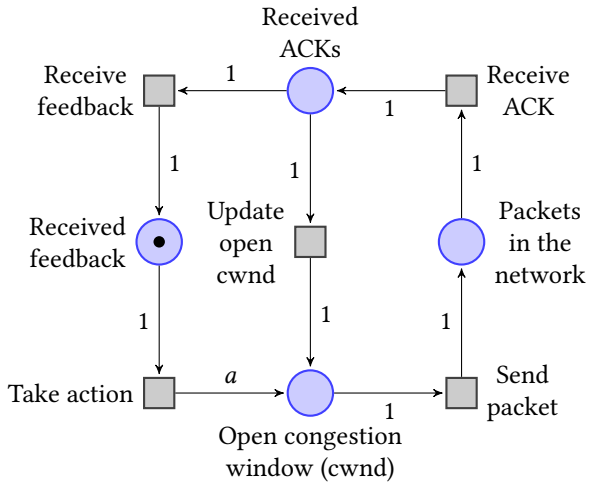



**Figure 3: A petri net describing the congestion control mechanism. It starts with one token in the state *Received feedback*, which means that it can take an action right in the beginning. The concept is the following: If there is at least one token in the *Open congestion window* it sends a packet. For the sake of simplicity in this example we assume that the congestion window is in packets and not bytes. The *Open congestion window* designates the congestion window minus the packets that are currently unacknowledged. The token now is passed 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 it receives an ACK, NACK (or timeout) it 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). Therefore, in each state there can 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.**

very small and thus the congestion window is often changed by minuscule amounts)

These features are not used themselves but instead of each of these features an exponentially weighted moving average (ewma) with a factor $\alpha$ of $\frac{1}{8}$ and one with an $\alpha$ of $\frac{1}{256}$ is computed, which makes a total of 8 features. The concept of using ewmas with these specific values of $\alpha$ was first introduced in *Remy* [17]). Each time an acknowledgement is received, the state is updated and the actor network is asked for the next action.

**$a_t$** The action $a_t$ is a real number that represents the change to the congestion window. It is computed based on a given state and the history of previous states. The history of previous states as considered as our neural network uses Gated Recurrent Units (GRUs) [3].

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

**$r_t$** The reward $r_t$ is a tuple of three reward metrics (defined in the next paragraph). 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 $v_t$ is a tuple of the expected average reward estimated by the critic network (see subsubsection 3.2.1) for each of the reward metrics. Also for the value $v_t$ the current state and the history of previous states is considered.

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 by the receiver). Example: During time step $t$, 2500 bytes are sent, of which 1250 are lost and 1250 are acknowledged. Then $r_{\text{received},t}$ is 1250 at time step $t$. However, $r_{\text{received},t}$ can only be determined at some point in time in the future, when the sender will have received the acknowledgements for all the bytes that he sent. Thus one has to wait one round-trip time until all rewards were received.

$\boldsymbol{r_{\text{duration},t}}$ is the time between receiving the last acknowledgement and receiving this packet ("inter acknowledgement 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 at some point (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.

The overall structure of the neural network for both the value and the actor network is depicted in Figure 4. 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 case of congestion control, the loss function $l_{v,t}$ of the critic network is actually 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:

$$\begin{aligned} l_{v,t} &= \left(R_{\text{received},t} - V_{\text{received}}(s_t;\theta)\right)^2 \\ &+ \left(R_{\text{sent},t} - V_{\text{sent}}(s_t;\theta)\right)^2 \\ &+ \left(R_{\text{duration},t} - V_{\text{duration}}(s_t;\theta)\right)^2 \end{aligned} \tag{4}$$

One apparent issue of using reinforcement learning for congestion control is that in RL one usually uses a fixed parameter $\gamma$ 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, the use of a single fixed parameter $\gamma$ incurs the problem that larger congestion windows result in shorter future intervals to be considered for congestion control. The reason is that the larger the window, the 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 $\gamma$ 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).

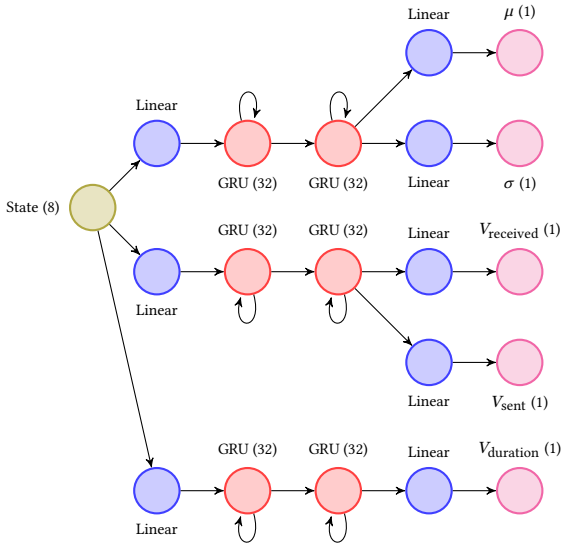



**Figure 4: An overview of the complete neural network being used in our experiments. Ocher stands for the input, blue for linear layers, red for GRU layers and purple for the outputs of the neural network. The numbers in parentheses next to each label stand for the width of that layer (i.e. the number of neurons except for the inputs and outputs). Parts of the neural network are shared between $V_{\text{received}}$ and $V_{\text{sent}}$ as well as $\mu$ and $\sigma$ as they output quantities of similar magnitude [a] 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.**

[a] The expected number of bytes sent and the expected number of bytes received are usually of similar size.

An ewma with a $\gamma$ of $\frac{2}{n+1}$ has similar characteristics as 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 [13].

Thus, the idea is to make $n$ the number of packets in the current window and so we define the factor $\gamma_t$ 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 follows:

$$\gamma_t = \frac{2}{1 + \max(\frac{w_t}{V_{\text{sent}}(s_t;\theta)}, 10)} \tag{5}$$

We use 10 as a minimum (as proposed by [4]) because when $\gamma$ becomes too large, future rewards are not considered sufficiently anymore and as the ensuing variability causes instability.

*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 $\mu$ and $\sigma$. 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 aims to minimize the loss function

$$\begin{aligned} l_{\text{a},t} = &- \log\left(\pi\left(a_t \mid s_t;\theta_{\text{a}}\right)\right)\left(U_{\text{measured},t} - U_{\text{expected},t}\right) \\ &- \beta H\left(\pi\left(s_t;\theta_a\right)\right) \end{aligned} \tag{6}$$

where $U$ can be any utility 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 utility 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].

$$U_t = \text{throughput} - \alpha \times \text{lost throughput} \tag{7}$$

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

$$U_{\text{measured},t} = \frac{R_{\text{received},t}}{R_{\text{duration},t}} - \alpha\frac{R_{\text{sent},t} - R_{\text{received},t}}{R_{\text{duration},t}} \tag{8}$$

The corresponding expected utility $U_{\text{expected},t}$ 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 [6] and Remy [17]. In the following evaluation we use a classical dumbbell network topology with one receiver and one or more senders and a shared bottleneck link. As our evaluation focuses on bulk transfers we use equally sized packets of 1250 bytes (including TCP and IP headers). In all figures in the evaluation we use data from at least 50 simulations unless stated otherwise. In case of multiple senders per simulation we randomly sample one sender from each simulation and leave out the others. The reason is that in each simulation the behavior of the senders is correlated.

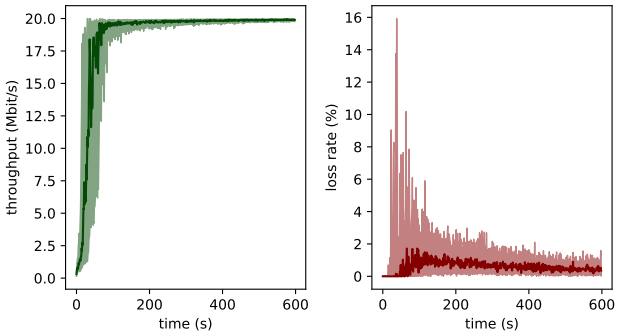
## 4.1 Tradeoff between throughput and packet loss

The tunable tradeoff between bandwidth and packet loss rate ($\alpha$) 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 (see Figure 5). However, the stronger packet loss is punished, the longer it takes for the sender to reach the maximum bandwidth on the link.

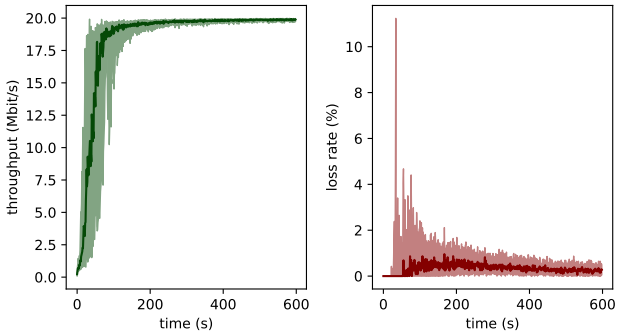
The apparent spike of packet loss in the beginning can be explained as follows: In the very beginning the reinforcement learning learns that increasing the window is always good (as it is still below the bottleneck bandwidth). This leads to exponential growth of the congestion window in the beginning and as soon as the bottleneck bandwidth is reached, the reinforcement learning needs to correct its behavior to realize that increasing the congestion window is not always the appropriate action to perform. Potential countermeasures for this are to either use pre-training (see subsection 4.3) or to use a utility function that heavily penalizes excessive packet loss.

## 4.2 Number of senders

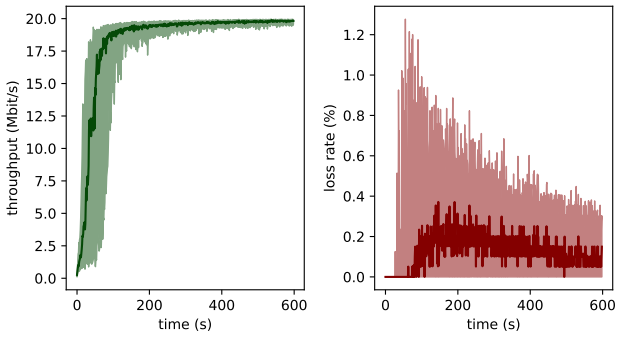
While TCP Rax generally does not exhibit much variance after a few minutes of training when it is the only flow on a link, fluctuations and training time needed to achieve good performance increase when having two or four concurrent senders (see Figure 6). We attribute this to the fact that on a shared link with $n$ senders each can only get $\frac{1}{n}$ of the bandwidth on average and thus each sender receives fewer packets which it can use for training and therefore training takes longer. 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.




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

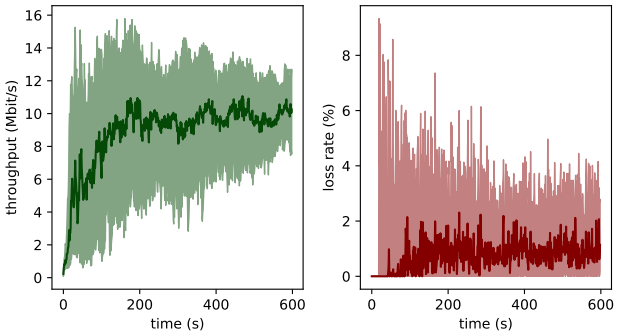



(b) One sender, throughput/packet loss tradeoff of 2

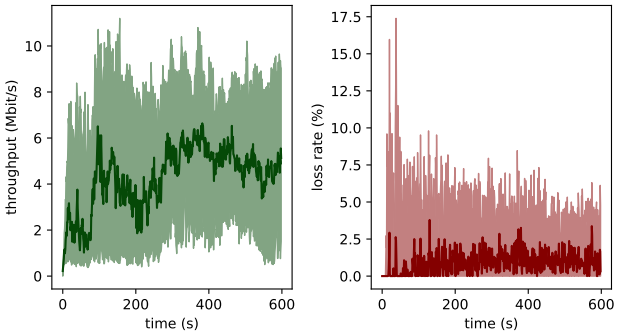



(c) One sender, throughput/packet loss tradeoff of 4

**Figure 5: One sender with varying throughput/packet loss tradeoff on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. Median in dark color and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in a light color.**




(a) Two senders, throughput/packet loss tradeoff of 2

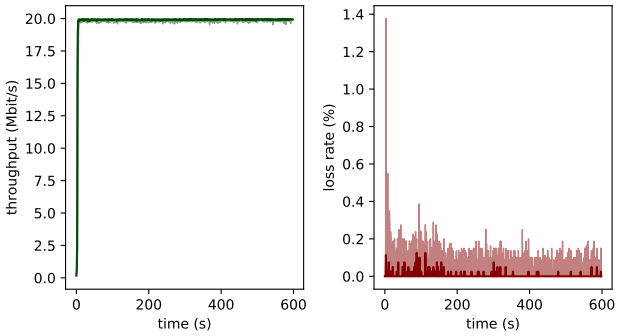



(b) Four senders, throughput/packet loss tradeoff of 2

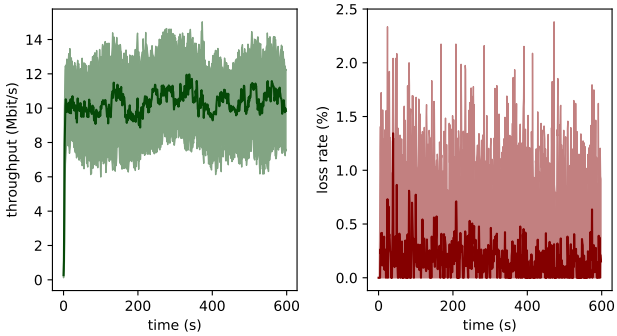
**Figure 6: Two (6a) and four (6b) senders with a bandwidth/packet loss tradeoff of 2 on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. Median in dark color and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in a light color.**

## 4.3 Pre-training

To verify whether 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 (see 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 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 in a completely different state than the one it was used to for hundreds of seconds before. For two senders pre-training also greatly speeds up the initial training procedure and can avoid the spike of packet loss in the beginning that always occurs to some extent without pre-training (see Figure 5 and Figure 6). Pre-training also works for more than one flow and although TCP Rax learns to minimize packet

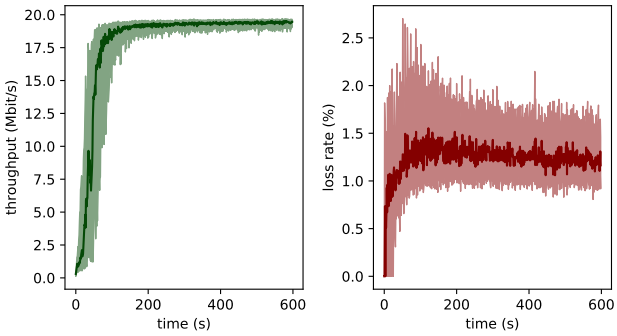



**(a) One sender initialized with a neural network from a previous flow with a throughput/packet loss tradeoff of 4**

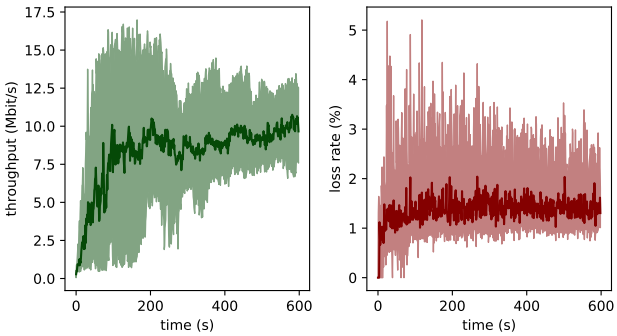



**(b) Two senders initialized with a neural network from a previous flow with a throughput/packet loss tradeoff of 4**

**Figure 7: One (7a) and two (7b) senders that were trained 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 dark color and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in a light color.**




**(a) One sender on a link with a stochastic loss rate of 1% with a throughput/packet loss tradeoff of 4**




**(b) Two senders sharing a link with a stochastic loss rate of 1% with a throughput/packet loss tradeoff of 4**

**Figure 8: One (8a) and two (8b) senders on a link with a stochastic loss rate of 1%, 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 dark color and $\frac{1}{4}$ and $\frac{3}{4}$ quantile band in a light color. Apparently there is no stochastic loss in the very beginning of the flows, however, this is an artifact of the binning procedure (we use bins of one second): As the window is very small in the beginning, there are many flows in which no packet loss occurs whatsoever in the first bins and thus the median is distorted towards 0.**

loss even further (see Figure 6b) some variability in throughput remains. We argue that this is caused by the stochastic behavior of the reinforcement learning: While each sender aims to maximize its utility, at the same time it also tries out new actions to check whether they could improve its utility even further and this behavior results in fluctuations.

## 4.4 Stochastic Loss

We also verify whether stochastic packet loss on the link has an influence on TCP Rax. For this, we randomly dropped 1% of packets on the link and it seems as if stochastic loss does not substantially affect TCP Rax. This can be attributed to the fact that the reinforcement learning realizes that a certain minimum packet loss rate is "normal" and cannot be avoided and thus TCP 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).

## 4.5 Comparison with other TCP congestion control variants

For reasons of comparability we take the same scenario of a 20 Mbit/s shared bottleneck with a small buffer of $\frac{1}{10}$ bandwidth delay product to create a challenging scenario. Surprisingly, New Reno performs by far better than more modern

TCP congestion control algorithms (see 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 [14]. TCP Rax achieves a significantly larger total throughput on average though with a couple of outliers which become more significant with two concurrent senders (see 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.

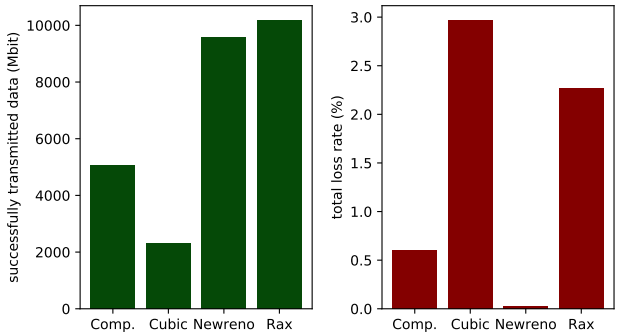
## 5 DISCUSSION

Our results demonstrate that our proposed concept can perform online learning of congestion control and can outperform existing congestion control algorithms in certain scenarios. Our results also show that with an increasing number of senders the reinforcement learning becomes significantly more challenging and time consuming. Regarding online learning, during the implementation of TCP Rax it became apparent that a neural network consisting of LSTM cells [7] is not feasible. LSTM cells keep an internal state that gets unstable when it is used for long training episodes, as for each received acknowledgement the LSTM state has to be updated. Previous work using deep reinforcement learning uses up to 50 million training steps for offline learning [12] while we generally use significantly less (a 20 Mbit/s link can carry a maximum of up to one 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 (see Figure 6b) one sender can receive a maximum of 600 000 packets during a ten minute flow for training. We argue that the reason for TCP Rax needing fewer training samples than other reinforcement learning tasks is that in congestion control 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 whether one won or lost the game).

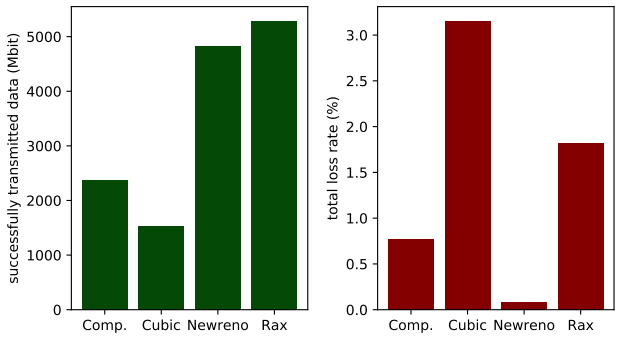
## 6 CONCLUSION

In this paper we show that it is possible to use deep reinforcement learning to learn a per acknowledgement congestion control strategy online. In order to achieve this we had to modify existing reinforcement learning algorithms to accommodate the specific needs of congestion control such as the inherent delay and proposed Partial Action Learning (PAL), a new method that allows deep reinforcement learning with delayed and partial rewards.

We developed TCP Rax and the results of our evaluation show that congestion control can be learned online and that it can outperform standard TCP variants for large flows. With




**(a) TCP Compound, Cubic and New Reno's and Rax's total throughput and loss rate for each sender with one senders in a simulation of 600 s**



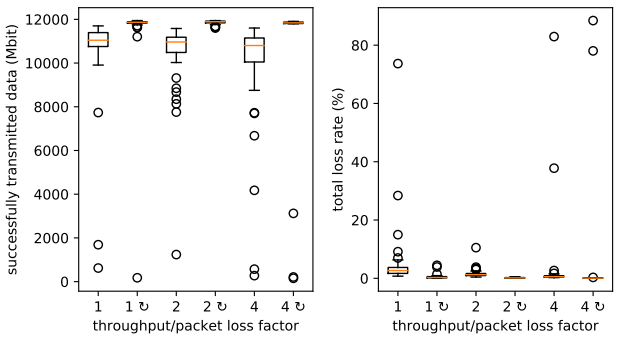

**(b) TCP Compound, Cubic and New Reno and Rax's total throughput and loss rate for each sender with two senders in a simulation of 600 s**

**Figure 9: Total throughput and loss rate ($\frac{\text{all packets lost}}{\text{all packets sent}}$) for one (9a) and two (9b) senders of TCP Cubic/Compound/New Reno and Rax with a throughput/packet loss tradeoff of 4 without pre-training on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. For a flow of 600 s the maximum possible throughput is 12 000 Mbit. For TCP Rax we use the mean over 50 simulation runs, while for the other TCP congestion control variants we use only one run because we use a deterministic simulator.**

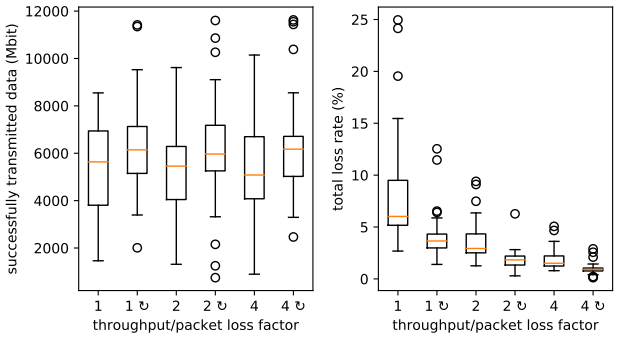
TCP Rax it is possible to pre-train the neural network used in the reinforcement learning and this improves performance especially in the beginning of the flow. However, TCP Rax can also compete with other congestion control variants after an initial exploration phase without pre-training.

## APPENDIX

The algorithm for Partial Action Learning is described in Algorithm 1 in a generic formulation that can also be used for




**(a) TCP Rax's total throughput for one sender showing different throughput/packet loss tradeoffs**




**(b) TCP Rax's total throughput for two senders showing different throughput/packet loss tradeoffs**

**Figure 10: Total throughput and loss rate ($\frac{\text{all packets lost}}{\text{all packets sent}}$) for one (10a) and two senders (10b) for TCP Rax on a link with 20 Mbit/s bottleneck speed and 50 ms RTT and a $\frac{1}{10}$ bandwidth delay buffer. 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.**

other reinforcement learning problems with similar characteristics such as delay and asynchronous actions and rewards. We use custom functions to determine when to update the neural network weights and when to compute the gradients and apply them in Algorithm 3 and Algorithm 4. Intuitively, these functions ensure that the neural network gets updated approximately every round-trip time. For other reinforcement learning problems it is also possible to simply use a fixed step size such as proposed by [12].

## REFERENCES

[1] 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.

**Algorithm 1** Partial Action Learning – pseudocode for each agent. The global weights $\theta_g$ can be shared by multiple agents, which is reasonable when learning offline. Otherwise an individual copy of them is kept by each agent. By default, all weights are initialized randomly 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**
  $l_{\text{actions}}, l_{\text{states}}, l_{\text{values}}, l_{\text{rewards}}, l_{\text{values'}} \leftarrow []$
  $l_{\text{snapshots}}, l_{\text{hiddenStates}} \leftarrow []$
  $t \leftarrow 0$
  $s_0 \leftarrow \text{initialState}()$
  $h \leftarrow \text{zeros}()$
  $\theta \leftarrow \theta_g$
  **repeat**
    $l_{\text{values'}}.\text{append}(V(s_t; \theta))$
    **if** updateWeights? **then**
      $\theta_a \leftarrow \theta_g$
      $l_{\text{snapshots}}.\text{append}(\theta)$
      $l_{\text{hiddenStates}}.\text{append}(h)$
    **end if**
    $l_{\text{states}}.\text{append}(s_t)$
    Sample $a_t$ from the
      Actor Network's probability distribution
    $l_{\text{actions}}.\text{append}(a_t)$
    $l_{\text{values}}.\text{append}(V(s_t; \theta))$ ▷ Possibly different from the value in $l_{\text{values'}}$ as the weights could have been updated in the meantime.
    $l_{\text{partialActions},t} \leftarrow \text{partialActions}(a_t)$
    **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 $0 \leq k \leq \#(l_{\text{partialActions},t'})$
      $l_{\text{rewards}}.\text{append}(r_{t'})$
      **if** gotEnoughRewards? **then**
        computeGradients
      **end if**
    **end if**
    Generate $s_{t+1}$ using $r_{p_{t'}}$
    $t \leftarrow t + 1$
  **until** reaching the end of 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 $t_{\text{end}}$ elements from $l_{\text{actions}}, l_{\text{states}}, l_{\text{values}}, l_{\text{rewards}}, l_{\text{values}'}$
  Remove the first element from $l_{\text{snapshots}}, l_{\text{hiddenStates}}$
  $\theta = \theta_{\text{backup}}$
  $h = h_{\text{backup}}$
**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 $l_{\text{windows}}$ and that the number of bytes sent in each time step is stored in $l_{\text{sent}}$. 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
        $i \leftarrow j$
        **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 steps $t$ is stored in a list $l_{\text{windows}}$ and that the number of bytes sent in each time step is stored in $l_{\text{sent}}$ analogously to the other metrics defined in Algorithm 1. 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?
  **if** the epsiode is over **then**
    **return true**
  **end if**
  $w \leftarrow \lfloor l_{\text{windows}}[0] + l_{\text{actions}}[0] \rfloor$
  **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
      **return true**
    **end if**
  **end for**
  **return false**
**end function**


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

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

[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] Kevin Fall and Kannan Varadhan. 2007. The network simulator (ns-2). *URL: http://www.isi.edu/nsnam/ns* (2007).

[7] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[8] 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.

[9] 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.

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

[11] 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

[12] 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.

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

[14] Ryo Oura and Saneyasu Yamaguchi. 2012. Fairness comparisons among modern TCP implementations. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*. IEEE, 909–914.

[15] Ming-Chang Shaio, Shun-Wen Tan, Kao-Shing Hwang, and Cheng-Shong Wu. 2005. A reinforcement learning approach to congestion control of high-speed multimedia networks. *Cybernetics and Systems: An International Journal* 36, 2 (2005), 181–202.

[16] 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*.

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