Some title*

Extended Abstract†

Maximilian Bachl
Institute of Telecommunications
Gußhausstraße 25
1040, Vienna, Austria
maximilian.bachl@tuwien.ac.at

Tanja Zseby
Institute of Telecommunications
Gußhausstraße 25
1040, Vienna, Austria
tanja.zseby@tuwien.ac.at

Joachim Fabini
Institute of Telecommunications
Gußhausstraße 25
1040, Vienna, Austria
joachim.fabini@tuwien.ac.at

ABSTRACT

This paper proposes a novel method of Congestion Control that uses on-line Reinforcement Learning (RL) to maintain the optimum congestion window with respect to a given utility function.

To this end we use an Artificial Neural Network (ANN) based approach that can be initialized with a previously trained neural network or with random weights (without any prior knowledge). As in congestion control the rewards used in the learning process naturally come delayed and bit by bit, we propose a novel formulation of reinforcement learning that allows us to deal with delayed and partial rewards.

We show that our method converges to a stable, close-to-optimum solution within the order of minutes and then commonly outperforms existing congestion control algorithms in typical networks. Thus, for the first time, we demonstrate that ANN based Reinforcement Learning without any preknowledge can feasibly be done on-line and can compete with hand-crafted solutions given long enough episodes.

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:
Maximilian Bachl, Tanja Zseby, and Joachim Fabini. 2017. Some title. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97),* 7 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Recent advances in Artificial Neural Network (ANN) based Reinforcement Learning (RL) for the purpose of playing video games raise the question if it is also possible to formulate sending Internet data in a fashion similar to a video game. The rules of the "Game of Congestion Control" can be summarized approximately as follows: Each player wants to send (1) as many packets as possible (2) with as little delay as possible (3) while losing as few packets as possible. 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 participating players play the game in a way that does not give them an unfair advantage over others. When several players (from now on called senders) share an Internet link under the same conditions, they should receive an equal share of the reward.

When performing Congestion Control (CG) TODO: What should be capitalized and what shouldn't? Capitalize RL, CG, ANN?, each sender uses a set of observed environment conditions to determine what 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.

From this formulation of Congestion Control, one can observe a couple of unique features of the problem that seem to prohibit the use of RL:

- 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 mean time. Thus, contrary to classical RL, **action and rewards are out of sync**.
- **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 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 assemble the overall reward of the corresponding action and can perform an update of the underlying RL logic.

*Produces the permission block, and copyright information
†The full version of the author's guide is available as `acmart.pdf` document

WOODSTOCK'97, El Paso, Texas USA

DOI: 10.475/123_4

To address these problems, we develop a new formulation of RL 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 RL.

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* [10]. The only seizable 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 [3]) we see that our machine learning based approach can learn congestion control that maximizes this objective given a couple of minutes of time (without any pre-training) 1, which – to our knowledge – is the first demonstration that neural network based on-line reinforcement learning is possible.

After having learned how to perform congestion control in a certain network scenario, our approach can compete with established algorithms and achieve superior throughput with packet loss at an average of less than 5%, while still gradually improving and adapting its way of performing congestion control.

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 [7]. 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, successful transmission and acknowledgement of data leads to the congestion window being increased while packet loss (or an increase in delay as in the Vegas algorithm [1] TODO: How much detail is necessary in the related work section?) lead to the congestion window being lowered according to a set of fixed rules. Algorithms such as Cubic and Compound [6, 9] improve congestion control specifically with respect to so-called long fat networks with a high round-trip time and bandwidth. Proportional rate reduction [4] 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 [2] 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 [3], 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).

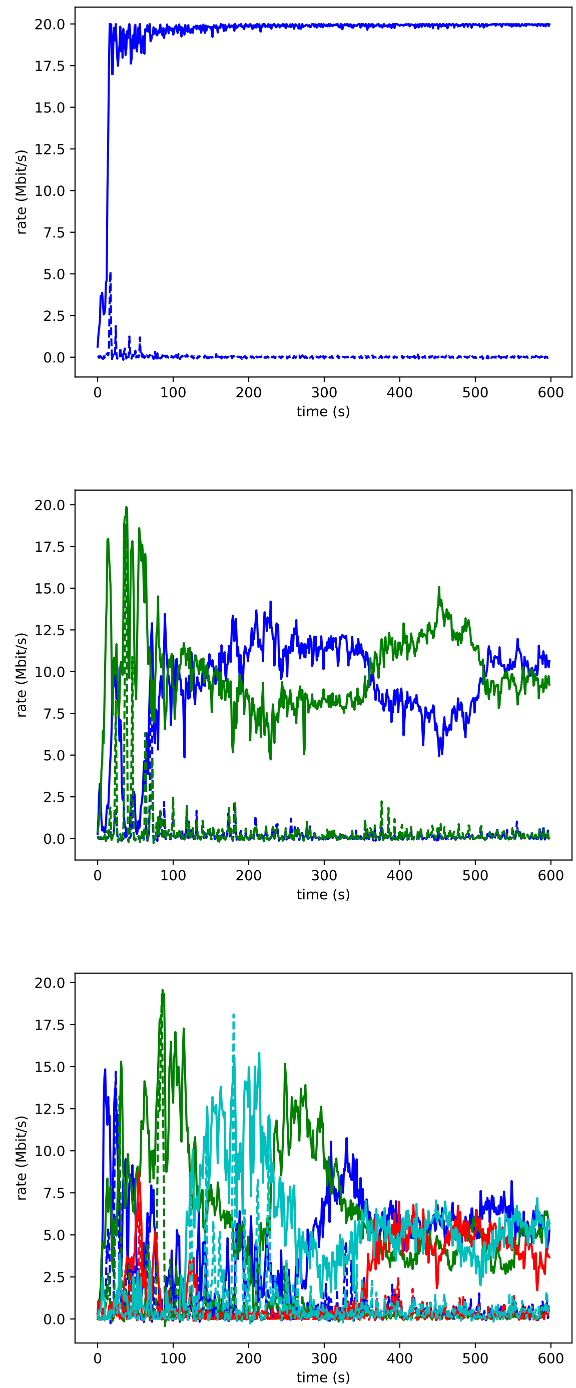

Figure 1: The throughput (solid line) and the loss rate (dashed line) when sharing a bottleneck link of 20 Mbit/s with a two-way end-to-end delay of 50 ms and a very small buffer of $\frac{1}{10}$ bandwidth delay product between 1, 2 and 4 senders (from top to bottom). One can see that the time to convergence increases as the number of senders increases, 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. Also, the complexity of the learning problem increases with more senders as the environment becomes more dynamic and unpredictable.

Besides the aforementioned human-designed algorithms there have been attempts to use machine learning to improve congestion control. Geurts et al. [5] 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 [10] train a machine learning solution called *Remy* that finds an optimum congestion control for a given range of a range of network parameters. For instance, one could find an optimum congestion control for networks with a 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.

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 pre-trains 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 use Partial Action Learning which is based on the Asynchronous Advantage Actor Critic framework [8], which has been demonstrated to be able to learn a wide range of video games and commonly outperform human players. In particular it is a good choice for congestion control as it can be easily adapted to a wide range of problems and has been proven to deliver good performance. 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. [8], which we outline in this section.

There are two Artificial Neural Networks (ANNs), the Actor Network and the Value Network (the Critic part in the abbreviation stands for the Value Network). Given a state, the Actor Network outputs what it deems to be the optimum action to perform in that certain state. The Value 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 Value 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 Value Network. The Value 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 value network θ_v.

With r_t being the reward that was received at time t and γ being the roll-off factor, which stands for the influence that future reward has on the moving average (commonly set to 0.99), we define the expected long-term reward as

$$R_t = \left(\left(\sum_{i=0}^{k-1} \gamma^i r_{t+i}\right) + \gamma^k V(s_{t+k};\theta_\mathrm{v})\right)(1-\gamma)\,,$$

where k is upper-bounded by t_max (t_max is a fixed hyperparameter that indicates how many rewards should be received before updating the neural network). So R_t is simply a moving average of rewards at time step t. However, usual moving averages take into account values from the past while this one uses values from the future. In practice this means that we collect t_max rewards, compute the expected long-term reward starting at each time step, update the neural network and start the same procedure again.

The loss function, which the value 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

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

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.

With v_t being the value that the value network estimated as the future reward given the current state s_t at time step t (v_t is just an abbreviation for $V(s_t;\theta_\mathrm{v})$) and θ_a the parameters of the actor network and with β being a factor that specifies the importance of the entropy H, π designating the probability density function, meaning that $\pi\,(a_t \mid s_t;\theta_\mathrm{a})$ is the value of the probability density function of taking action a_t in state s_t with the current weights of the actor network θ_a, we define the loss that the actor network aims to minimize as follows:

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

3 METHOD

3.1 Partial Action Learning

The key difference between PAL and previous approaches to Reinforcement Learning is that in classical Reinforcement Learning, an action is always followed by a reward and a reward is always followed by a 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 value and actor networks.

In Algorithm 1 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 weights but one can also use separate weights for each agent,

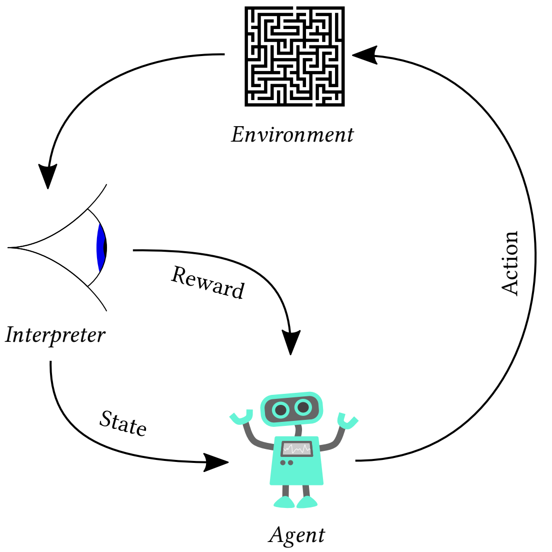

Figure 2: The classical Reinforcement Learning approach.[a]

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

which is more realistic in case of congestion control in the Internet, as different senders cannot easily 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). 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 two acknowledgements directly after one other. For each of these two acknowledgements an action has to be performed but it does not seem feasible for the second action to wait until the first action has received a reward as it takes one RTT until an acknowledgement (a reward) is received. Thus the Asynchronous Actor Critic framework as described by [8] 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 value 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.

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

- the round-trip time of the last packet
- the current congestion window
- the time between the last two packets that were sent

Algorithm 1 Partial Action Learning – pseudocode for each agent. It is possible that the agents share the global weights $\theta_{\mathrm{a,g}}$ and $\theta_{\mathrm{v,g}}$, which can be reasonable when learning off-line. Otherwise an individual copy of them is kept by each agent. All weights are initialized randomly in the beginning. LSTM cells are used in both the actor and the value network.

loop
  $l_{\mathrm{actions}} \leftarrow []$
  $l_{\mathrm{states}} \leftarrow []$
  $l_{\mathrm{values}} \leftarrow []$
  $l_{\mathrm{rewards}} \leftarrow []$
  $l_{\mathrm{estimatedValues}} \leftarrow []$
  $l_{\mathrm{snapshots}} \leftarrow []$
  $l_{\mathrm{hiddenStates}} \leftarrow []$
  $t \leftarrow 0$
  $s_0 \leftarrow \mathrm{initialState}()$
  $h_{\mathrm{a}} \leftarrow \mathrm{initialHiddenState}()$
  $h_{\mathrm{v}} \leftarrow \mathrm{initialHiddenState}()$
  $\theta_{\mathrm{a}} \leftarrow \theta_{\mathrm{a,g}}$
  $\theta_{\mathrm{v}} \leftarrow \theta_{\mathrm{v,g}}$
  repeat
    $l_{\mathrm{estimatedValues}}.\mathrm{append}(V(s_t;\theta_{\mathrm{v}}))$
    if $t \bmod t_{\mathrm{max}} = 0$ **then**
      $\theta_{\mathrm{a}} \leftarrow \theta_{\mathrm{a,g}}$
      $\theta_{\mathrm{v}} \leftarrow \theta_{\mathrm{v,g}}$
      $l_{\mathrm{snapshots}}.\mathrm{append}((\theta_{\mathrm{a}}, \theta_{\mathrm{v}}))$
      $l_{\mathrm{hiddenStates}}.\mathrm{append}((h_{\mathrm{a}}, h_{\mathrm{v}}))$
    end if
    $l_{\mathrm{states}}.\mathrm{append}(s_t)$
    Sample a_t from the
      Actor Network's probability distribution
    $l_{\mathrm{actions}}.\mathrm{append}(a_t)$
    $l_{\mathrm{values}}.\mathrm{append}(V(s_t;\theta_{\mathrm{v}}))$
    $l_{\mathrm{partialActions},t} \leftarrow \mathrm{partialActions}(a_t)$
    for all partial actions $a_{\mathrm{p},i,t}$ in $l_{\mathrm{partialActions},t}$ **do**
      Take partial action $a_{\mathrm{p},i,t}$
    end for
    Wait for the next feedback $r_{\mathrm{p},j,t'}$
      where $t' \leq t$ and $0 \leq j \leq \#(l_{\mathrm{partialActions},t'})$
    if all feedback of $a_{t'}$ was received **then**
      $r_{t'} \leftarrow$ reward w.r.t all feedback $r_{\mathrm{p},k,t'}$
        where $0 \leq k \leq \#(l_{\mathrm{partialActions},t'})$
      $l_{\mathrm{rewards}}.\mathrm{append}(r_{t'})$
      if $\#(l_{\mathrm{rewards}}) > t_{\mathrm{max}}$ or the episode is over **then**
        ComputeGradients
      end if
    end if
    Generate s_{t+1} using $r_{\mathrm{p}_{t'}}$
    $t \leftarrow t + 1$
  until reaching the end the episode
end loop

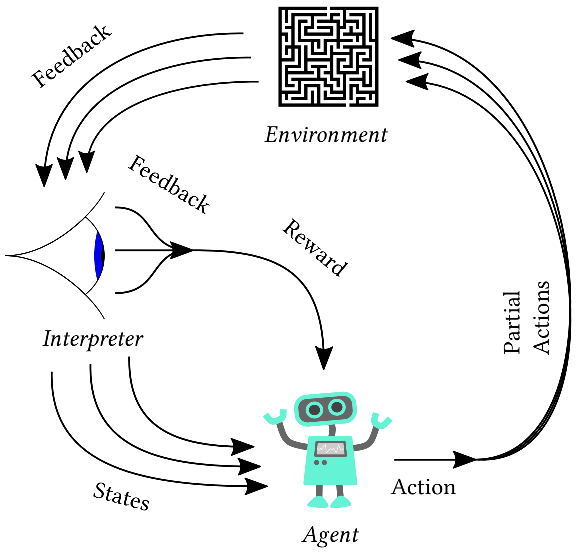

Figure 3: Partial Action Learning: An action consists of zero or more partial actions which trigger feedback upon interacting with the environment. Each feedback updates the state. The value and actor networks are updated upon receiving all feedback of one action.[a]

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

- the time between the last two packets that were received
- the number of packets that were lost since the last acknowledgement was received

Each time an acknowledgement is received, the state is updated and the actor network is asked for the next action.

a_t Based on a given state and the history of previous states (because we use LSTM cells 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.

r_t The reward is a tuple of at least one reward metric. For each reward metric there is also an output of the value network that predicts the expected long-term average of this reward metric given the current state.

We actually use the following types of reward:

$r_{\text{packet},t}$ is the number of the packets that the sender sent during time step t and that were not lost (so they were acknowledged at some point by the receiver).

$r_{\text{delay},t}$ is the sum of the round trip times of the packets that the sender sent and that were not lost.

$r_{\text{duration},t}$ is the sum of the time between receiving the last packet and receiving this packet ("inter-receive time") for the packets that the sender sent and that were not lost.

$r_{\text{lost},t}$ is the sum of packets that were lost between receiving the last packet and receiving this packet.

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

```
function computeGradients
    t_end = min(t_max, #(l_rewards))
    θ_backup ← (θ_a, θ_v)
    h_backup ← (h_a, h_v)
    θ_a, θ_v ← l_snapshots[0]
    h_a, h_v ← l_hiddenStates[0]
    R_{i+1} ← last element of l_estimatedValues
    for i ← t_end − 1, 0 do
        R_i ← (r_i + γR_{i+1})(1 − γ)
        a ← l_actions[i]
        s ← l_states[i]
        v ← l_values[i]
        dθ_a ← −∂log(π(a | s;θ_a))(R_i − v)/∂θ_a
               − ∂βH(π(s;θ_a))/∂θ_a
        dθ_v ← ∂(R_i − v)/∂θ_v
        θ_{a,g} ← θ_{a,g} + dθ_a
        θ_{v,g} ← θ_{v,g} + dθ_v
    end for
    Remove first t_end elements from
        l_actions, l_states, l_values, l_rewards, l_estimatedValues
    Remove the first element from l_snapshots
    Remove the first element from l_hiddenStates
    θ_a, θ_v = θ_backup
    h_a, h_v = h_backup
end function
```

The overall structure of the neural network for both the value and the actor network is depicted in Figure 5.

3.2.1 Value Network. In the case of congestion control, the loss function $l_{\text{v},t}$ of the value 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:

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

3.2.2 Actor Network. The actor network outputs two parameters: The mean of a normal distribution μ and its standard deviation σ.

Each time an action a_t is requested, it is sampled from the current normal distribution defined by the parameters μ and σ.

Then the window is incremented by a_t; however, the window can never be smaller than 1. At the beginning of a flow the window starts with 1 as well.

With H being the entropy, the actor network minimizes the loss

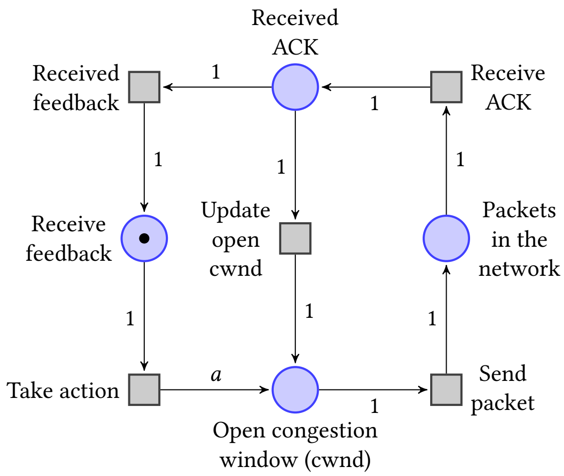

Figure 4: A petri net describing the congestion control mechanism. We start with one token in the state *Received feedback*, which means that we can take an action right in the beginning. The concept is the following: If there is at least one token in the *Open congestion window*, send a packet. The token goes to the network and after some time the acknowledgement (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 (or timeout) we can send another packet (The *Open congestion window* signifies the congestion window minus the packets that are currently unacknowledged.). Furthermore the state *Received feedback* receives a token upon receiving an ACK/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 (This 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.

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

where U can be any utility function defined based on some of the reward metrics. In the above formulation, one considers how much better the actual experienced Utility was compared to the expected one. The Utility is formed from the previously defined expected long-term values.

3.3 Utility function

As the utility function one can choose an arbitrary function based on the reward metrics previously defined. A safe choice is to use PCC's [3] reward function as its convergence has been previously proven. It is defined as follows:

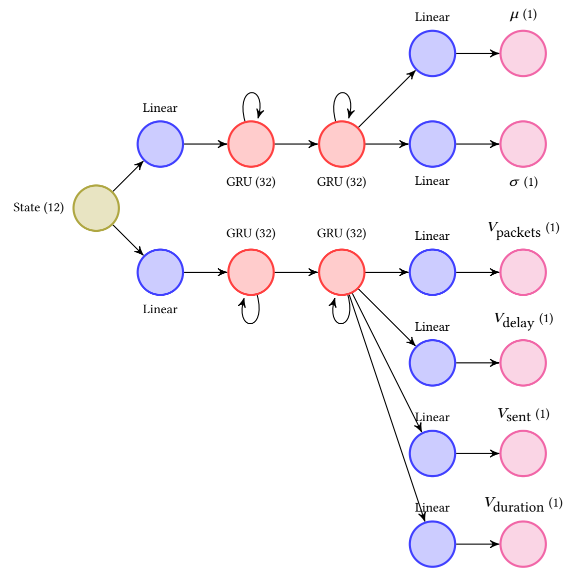

Figure 5: An overview of the complete neural network being used. Ocher stands for the input, blue for linear layers, red for LSTM layers, green for linear layers with a softplus activation function and purple for the outputs of the neural network. The numbers in parentheses next to each label stand for the number of width of that layer (i.e. the number of neurons except for the inputs and outputs).

$$U_t = r_{\mathrm{received},t}\,\mathrm{Sigmoid}_\alpha\left(\frac{r_{\mathrm{loss},t}}{r_{\mathrm{sent},t}} - 0.05\right) + r_{\mathrm{loss},t}$$

where

$$\mathrm{Sigmoid}_\alpha(x) = \frac{1}{1 + e^{\alpha x}},$$

r_{received} is the actual throughput (in packets per second), r_{loss} is the loss rate (in packets per second) and r_{sent} is the sending rate (in packets per second). Thus $\frac{r_{\mathrm{loss}}}{r_{\mathrm{sent}}}$ is the ratio of data getting lost compared to those being sent, which is a number between 0 and 1.

Intuitively it means that one takes the actual throughput (packets that get received by the receiver and do not get lost) times the sigmoid function of the loss ratio minus 0.05 plus the loss rate. The sigmoid function essentially acts as a cutoff threshold. As soon as the loss rate rises above 5%, the Sigmoid function decreases very quickly and thus diminishes the Utility.

It is also possible to set the cutoff threshold to 0, which means that losing packets is heavily punished; or one can increase the threshold to allow for more loss.

$r_{\mathrm{received},t}$ can be calculated as $\frac{R_{\mathrm{packet},t}}{R_{\mathrm{duration},t}}$, $r_{\mathrm{loss},t}$ can be calculated as $\frac{R_{\mathrm{lost},t}}{R_{\mathrm{duration},t}}$ and $r_{\mathrm{sent},t}$ can be calculated as $\frac{R_{\mathrm{packet},t}+R_{\mathrm{lost},t}}{R_{\mathrm{duration},t}}$.

Actually, we assume equally sized packets for all of the above methodology, however, it would also easily be possible to consider the number of bytes per packet as well: One would have to add an additional reward metric of the average number of bytes per packet and as well add one more output to the value network.

3.4 Example

In this section, an example of the overall procedure is outlined.

(1) Receive an ACK
(2) Update the internal state to reflect the information that this ACK provides (for example, the state consists of the round trip time experienced by this packet, the time since we receive the last ACK etc.)
(3) Sample an increase for the congestion window from the Actor Network (e.g. 0.24) and add it to the current congestion window.
(4) (a) If this was the last ACK, which completes a previous action (e.g. a previous action resulted in 3 packets being sent; so if this is the ACK for the third (and last) packet of this previous action, we got all the ACKs to compute the reward), combine all partial rewards to form the actual reward for that action.
    (b) If we now have t_{max} rewards (set to 20) then we also update the neural network.

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.

[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] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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*.

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