ReCoCo: Reinforcement learning-based Congestion control for Real-time applications

ReCoCo: Reinforcement learning-based Congestion control for Real-time applications

1st Dena Markudova
Politecnico di Torino
dena.markudova@polito.it

2nd Michela Meo
Politecnico di Torino
michela.meo@polito.it

***Abstract*—Real-time communication (RTC) platforms have seen a considerable surge in popularity in recent years, largely due to the COVID-19 pandemic which facilitated remote work. To ensure adequate Quality of Experience (QoE) for users, a good congestion control algorithm is needed. RTC applications use UDP, so congestion control is done on the application layer, leaving way for advanced algorithms. In this paper, we propose *ReCoCo*, a solution for congestion control in RTC applications based on Reinforcement learning (RL). *ReCoCo* gains information about the network conditions at the receiver-side, such as receiving rate, one-way delay and loss ratio and predicts the available bandwidth in the next time bin. We train *ReCoCo* on 9 bandwidth trace files that cover a vast array of network types. We try different algorithms, states and parameters, training both specific and general models. We find that *ReCoCo* outperforms the de-facto standard heuristic algorithm *GCC* in both specialized and general models. We also make observations on the difficulty of generalization when using RL.**

***Index Terms*—networking, reinforcement learning, congestion control, rate adaptation, real-time communications**

I. Introduction

Internet Real-time Communication (RTC) platforms, such as video-conferencing applications (VCAs) and cloud gaming have been on the rise in recent years. VCAs are the main enabler of remote working, which has become the de-facto standard way of work for many companies [1]. Nowadays there are countless VCA applications on the market [2], most of which use the Real-time Protocol (RTP) [3]. In web browsers, RTC is enabled by the open-source WebRTC framework[1], on top of RTP. In this context, it is becoming increasingly important to maximize the Quality of Experience (QoE) of users of RTC applications.

One way to improve QoE is through good congestion control (CC). RTC applications use RTP mostly over UDP [2], so they are not subject to TCP congestion control protocols. Instead, CC is implemented via rate adaptation at the application layer, by using a feedback mechanism between the sender and receiver, that relies on the Real-time Control Protocol (RTCP). The biggest challenge for CC lies in the low-latency requirement of RTC applications. Thus, the goal of CC algorithms is to produce a sending rate as close as possible to the available end-to-end bandwidth, while maintaining the queue occupancy as low as possible [4]. The sending rate directly affects the packet delay, losses and throughput, which are the main drivers of network QoE [5]. The algorithms in use by RTC applications today are heuristic schemes that make decisions on increasing or decreasing the sending rate, based on the one-way queuing delay and loss ratio [4]. The most notable open-source algorithm is Google's GCC [6], which measures the delay variation and compares it with a dynamic threshold. However, in a complicated network scenario, such as wireless network links with very variable bandwidth, it is hard to optimize all network metrics with a heuristic scheme. To combat these limitations, we propose a novel rate adaptation scheme, based on Reinforcement learning (RL). Using RL for congestion control in RTC has been somewhat explored in the literature [7]–[9]. We elaborate on the limitations of these works and differences with respect to ours in Section II.

In this paper, we propose *ReCoCo*, a fully-RL based solution for congestion control in real-time applications. To create our system and experiments, we build upon the open-source framework OpenNetLab [10]. This framework was first built to serve the *MMSys2021 Grand challenge* [11], which called for a novel bandwidth estimation scheme for RTC. We thus use some of the performance metrics defined by this challenge to evaluate our approach. We assess 3 different RL algorithms in a number of parameter configurations, on 9 different bandwidth trace files that comprise of wired, 4G and 5G channels covering a vast array of bandwidth levels. We train both specific and general models to evaluate the difficulty of generalizing RL algorithms. We find that, when trained on each trace file separately, with specialized configuration, *ReCoCo* outperforms *GCC* for every trace, by 8.95 QoE units on average, especially for traces with high bandwidth. When training a single model for all network conditions, the best way is to use curriculum training, ordering the environments easiest to hardest based on improvement over a heuristic baseline (gap-to-baseline). In this case we observe a performance penalty of 8.76 QoE units on average over the specialized model, but still outperform *GCC* by 0.2 QoE units.

To make our research reproducible, we disclose our codebase and trained models[2].

II. Related Work

Congestion control for RTC. CC in employed RTC ap-

This work has been supported by the SmartData@PoliTO center on Big Data and Data Science

[1]https://webrtc.org/

[2]https://github.com/denama/ReCoCo

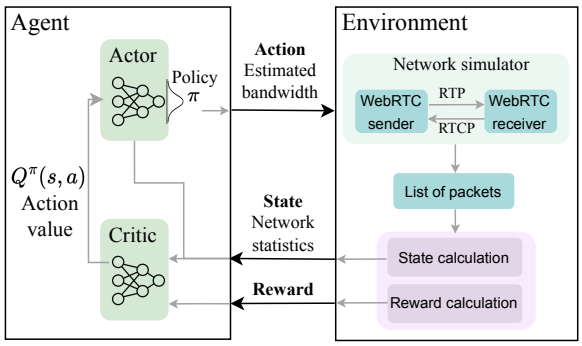


Fig. 1: Overview of the system

plications today is mostly heuristic-based. There are three main standardized algorithms: NADA [12], SCReAM [13] and GCC [6]. They all employ delay-based mechanisms to detect congestion and loss-based mechanisms as fall-back when there is buffer overflow. The most employed implementation is *GCC*, used by the popular VCA, *Google Meet*, and by any RTC application that runs in the browser (e.g. the cloud gaming platform *GeForceNow*). Thus we use it as our baseline. *GCC* suffers from a few limitations: it only responds to latency variation, so it does not note an increase in the absolute value of RTT, introducing delays in the conversation. It also becomes too conservative in the event of losses [5]. *GCC* is slow to respond to an increased bandwidth, since it increases the sending rate by just 5% every second. This holds true also for some RTC applications that use proprietary congestion control protocols [14]. With *ReCoCo* we aim to overcome these limitations by setting the reward to optimize for all three metrics: delay, losses and bandwidth utilization, at any given time, making it faster to adapt to varying network conditions.

Reinforcement learning in RTC Congestion control. There are three previous works that tackle the problem of CC for RTC with RL: HRCC, CLCC and R3Net. HRCC [8] is a hybrid receiver-side scheme, that uses *GCC* as the main algorithm and occasionally tunes the bandwidth estimate with a gain coefficient generated by an RL agent. R3Net [7] is a fully RL-based solution, that uses the PPO algorithm. Its state consists of receiving rate, loss rate, average RTT and average packet inter-arrival time. The reward function is a very simple linear combination of the receiving rate, delay and loss rate. CLCC [9] uses both packet-level and frame-level statistics of traffic as states and in the reward, arguing that frame-level information is vital for RTC. However, this requires adding an additional exclusive channel to RTP, to send frame-level information, which is a non-trivial change to the protocol.

Albeit having the same end goal, our work contrasts these works in a few important ways. First, we explore many different configurations of the problem formulation. By trying different algorithms, we find that PPO, which is used by both R3Net and CLCC does not perform well on our traces. We meticulously design the reward function, outlining separate functions for all network metrics we wish to optimize, based on network standards and recommendations. We provide details on the traces we use, to better understand what kind of environments are hard for RL algorithms to solve, while previous works do not elaborate on the dataset. We also present both specialized and general models, discussing some challenges in generalizing RL algorithms for networking problems.

III. Background: Deep RL basics

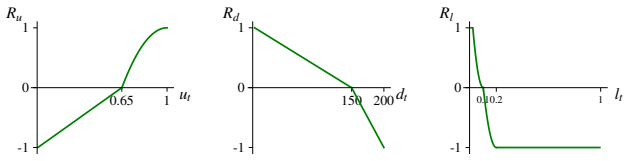
The setting of RL [15] consists of an *agent* that interacts with an *environment*, in a discrete time stochastic control process. At every time step t, the agent finds itself in a state $\mathbf{S_t}$ and takes an action a_t. This action brings the agent a reward R_t and transitions it to the next state $\mathbf{S_{t+1}}$. Which action an agent takes at any given time step from any given state is defined by its policy π. The agent's goal is to learn a policy π that maximizes the reward in the long run. In fact, it aims to maximize the *discounted return* $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$, where γ is a discount factor decreasing the weight of past rewards. In many real-life tasks, the state space is arbitrarily large and often continuous. Here the agent learns a policy π through a function approximator - usually a neural network (Deep RL). The algorithms we use in this paper use the Actor-critic architecture [16] for Deep RL. The actor-critic framework constitutes two neural networks, an actor and a critic. The actor learns the policy π and decides the action to take at every time step. The critic evaluates how good that action was compared to the average for that state and informs the actor. The actor then changes the weights of the policy function accordingly, to adjust the probability of that action being taken. In formal terms, the critic learns an *action value function* $Q^{\pi}(s,a)$.

IV. System overview

In this section, we describe our system, depicted on Figure 1. Mapping the RL framework to our problem, we get the following scenario: The agent is an RL algorithm that, at every time interval Δt, predicts the available bandwidth and sends this information to the environment as an action. The environment runs a network simulation of RTP traffic between a sender and a receiver, given a bandwidth trace file, and based on the action, adjusts the sending rate. Then the simulation runs with that sending rate for a time interval Δt and spits out a list of packets. From these packets we calculate a set of network statistics (such as average delay, loss ratio etc.) - the state. Based on the state we also calculate an appropriate reward. For instance, if the loss ratio is high, the reward is very low. The state and reward are sent to the agent, that, based on them, adjusts its policy.

For the network simulation and the agent's interaction with it, we use the OpenNetLab [10] framework, which provides a plug-and-play *gym*[3] environment for training RL algorithms to the task of congestion control in RTC, using an *ns-3* event-driven network simulator and Chrome's WebRTC. We make some changes to the environment to account for our states and rewards.

[3]https://github.com/OpenNetLab/gym



(a) R_u (b) R_d (c) R_l

Fig. 2: Functions describing the reward

A. State space

The states are histories of network statistics. In our implementation, we use a statistics vector $\overrightarrow{v_t}$, which has four components, calculated at time t, for the duration of Δt: (i) Receiving rate r_t, (ii) Average delay d_t, (iii) Loss ratio l_t and (iv) Last action taken a_{t-1}. We then use multiple past instances of $\overrightarrow{v_t}$ as our state $\boldsymbol{S_t}$ at time t:

$$\overrightarrow{v_t} = (r_t, d_t, l_t, a_{t-1}), \boldsymbol{S_t} = (\overrightarrow{v_t}, \overrightarrow{v_{t-1}}, ...\overrightarrow{v_{t-x}}) \quad (1)$$

where x is the number of time intervals Δt we consider in the past. In our experiments, we fix Δt to $200ms$ and we vary the parameter x, by setting it to 5 (delayed states) or setting it to 0 (non-delayed states). In the latter case, $\boldsymbol{S_t} = \overrightarrow{v_t}$. We normalize all state values in the interval [0,1], as we find this is vital for many RL algorithms to learn.

B. Reward design

The reward is the most important driver of RL algorithms, so we take careful consideration in designing it. Our reward function incorporates three components: bandwidth utilization, delay and loss ratio. The reward is always normalized to the interval [-1,1]. The functions describing the different components of the reward are depicted on Figure 2. We start by drawing these functions using domain knowledge and then translate them to corresponding equations.

Bandwidth utilization reward component (R_u). It expresses how well the algorithm is using the available bandwidth. Let u_t be the bandwidth utilization at time t:

$$u_t = r_t / B_t, u_t \in [0, 1] \quad (2)$$

where r_t is the receiving rate and B_t is the bandwidth in the time bin ending at time t. Then R_u is defined as:

$$R_u = \begin{cases} 1.538u_t - 1, & \text{if } 0 < u_t \leq u_{thres} \\ -8.2(u_t - 1)^2 + 1, & \text{if } u_{thres} < u_t \leq 1. \end{cases} \quad (3)$$

where $u_{thres} = 0.65$.

R_u is depicted on Figure 2a. The closer the bandwidth utilization u_t is to 1, the higher the reward. u_{thres} is a threshold that distinguishes between a negative and positive reward. It means that the minimum bandwidth utilization we consider acceptable is 65%. Note that if u_t is higher than 1, then the receiving rate is higher than the available bandwidth (over-utilization). In that case we force the whole reward R_t to -1.

Delay reward component (R_d). It expresses how acceptable the one-way delay between the sender and the receiver is. Let d_t be the average one-way delay in the time bin Δt. Then R_d is defined as:

$$R_d = \begin{cases} -0.00667d_t + 1, & \text{if } 0 < d_t \leq 150 \\ -0.02d_t + 3, & \text{if } 150 < d_t \leq 200. \end{cases} \quad (4)$$

The equation is depicted on Figure 2b. We design R_d according to the G.114 recommendation for one-way transmission time [17], which states that if delays were kept below 150 ms, then most real-time applications would not be significantly affected. If the delay is more than 200ms, it gets a reward of -1, since we aim for a low-latency algorithm.

Loss ratio reward component (R_l). It expresses how well the algorithm is doing in terms of losses. Let l_t be the loss rate in the time bin ending at time t. Then R_l is defined as:

$$R_l = \begin{cases} 1, & \text{if } 0 \leq l_t \leq 0.02 \\ 156(l_t - 0.1)^2, & \text{if } 0.02 < l_t \leq 0.1 \\ 100(l_t - 0.2)^2 - 1, & \text{if } 0.1 < l_t \leq 0.2 \\ -1, & \text{if } 0.2 < l_t \leq 1. \end{cases} \quad (5)$$

The equation is depicted on Figure 2c. To design the first two thresholds in (5) we rely on the GCC thresholds for acceptable loss rate [18].

Final reward equation (R_t). Combining all the reward components together, the final reward at a time step ending at time t is:

$$R_t = \begin{cases} 0.333R_u + 0.333R_d + 0.333R_l, & \text{if } l_t > 0 \\ 0.4R_u + 0.4R_d + 0.2R_l, & \text{otherwise.} \end{cases} \quad (6)$$

Since losses are a rare event, we want to mitigate the effect of a positive reward from the loss component. Thus, we decrease the weight of R_l if the loss ratio is 0. In addition, we force R_t to 1 if all these conditions apply: the loss ratio is below 0.02, the delay is below 30 and the bandwidth utilization is higher than 0.9.

V. Experimental setup

In this section, we outline all the experiments we conduct to train *ReCoCo*, starting from the traffic traces we use, the process of training, the configuration parameters we try and finally the QoE performance metrics we use for evaluation.

A. Dataset

To train the algorithm we use 9 trace files that specify the channel bandwidth in time. The trace files are open data by OpenNetLab [10]. Their distributions are depicted on Figure 3. We use traces from a wired channel (green), a 4G cellular channel (blue), a 5G channel (red) and one trace with stable bandwidth at 300 kbps. The traces have different duration, from a minimum of $60s$ to a maximum of $223s$, with a mean duration of $88.5s$. While training, the simulator goes through the traces many times.

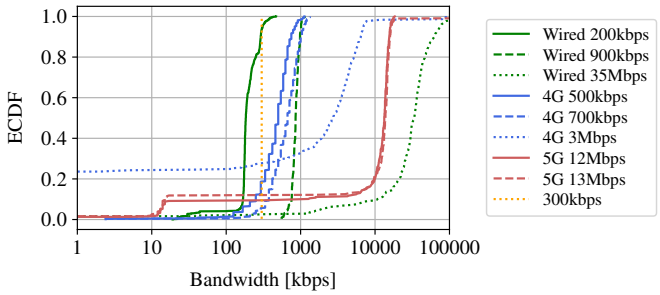


Fig. 3: Bandwidth distribution of different trace files

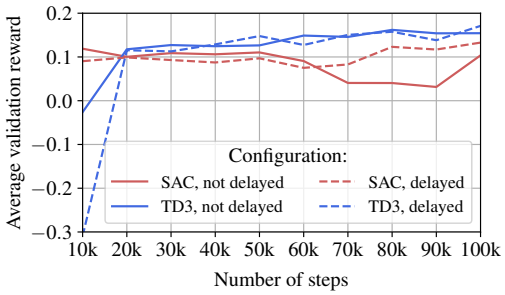


Fig. 4: Validation during training of some experiment configurations on *Wired 900kbps*

B. Employed algorithms

We employ three different algorithms: (i) Soft Actor-Critic (SAC) [19], (ii) Twin-delayed DDPG (TD3) [20] and (iii) Proximal Policy Optimization (PPO) [21]. They are all Deep RL algorithms, using the actor-critic architecture, (see Section III and Figure 1) and employing different optimizations. Indeed, as stated in Section II, algorithms based on the actor-critic paradigm have proven to be successful for congestion control tasks. SAC is an Off-policy Maximum Entropy algorithm with a Stochastic Actor. It is trained to maximize a trade-off between expected return and entropy. TD3 is an Off-policy algorithm that uses clipped double Q-learning (two critic networks), a delayed policy update and target policy smoothing. PPO uses clipping to avoid large updates between the old policy and the new policy. We use the implementation of these algorithms in the Python library *Stable Baselines 3*[4].

C. Configuration parameters

Since Deep RL is very dependent on parameters and results can change considerably, we try a myriad of different configurations. One parameter we vary is x from Equation 1. We either set it to 0 (non-delayed states) or to 5 (delayed states by five Δt). We try two versions of the algorithm hyperparameters - one is the default suggested by *Stable baselines 3* and another is tuned hyperparameters for an RL environment similar to ours (the Cartpole environment). The tuning is provided by *Stable Baselines Zoo*[5]. We call these configurations *not tuned* and *tuned*, respectively.

D. Training and validation strategy

For each configuration *(trace, algorithm, delayed states/not, tuned hyperparameters/not)*, we employ training of $100k$ steps. One step is equal to Δt in simulation time. Every $10k$ steps, we save the model and perform validation on the same environment (same trace file), by observing the average reward collected on the whole trace file. This procedure helps us choose the best configuration for each trace. Namely, the configuration that performs better has a higher average reward in the final few tests. When two or more configurations perform similarly enough in terms of average reward, we evaluate them using our QoE metrics defined in Section V-E.

[4]https://stable-baselines3.readthedocs.io/

[5]https://github.com/DLR-RM/rl-baselines3-zoo

An example of validation during training of a few configurations is shown on Figure 4. Here the environment is the trace *Wired 900kbps*. The lines are an average of 3 runs with different random seeds. All configurations on the plot consider the algorithms with tuned parameters. We notice that the average reward grows as training progresses. Notice that the first point is already after $10k$ steps of training. Since at least three options show similar performance, we choose the best one in terms of QoE metrics, which turns out to be *(SAC, delayed)* - red dashed line. Training $1M$ steps of TD3 takes 4 hours on an Nvidia v100 Tensor Core GPU. Applying the model at runtime is instantaneous.

E. Performance metrics

To evaluate the goodness of our models on the network traces, we employ the Quality of Experience (QoE) scores defined by OpenNetLab [10]. The QoE score is composed of three components: (i) Receiving rate QoE, (ii) Delay QoE and (iii) Loss QoE. The Receiving rate QoE is given by:

$$QoE_{rr} = 100 \times U \tag{7}$$

where U is the median bandwidth utilization in the trace (a median of all u_t from Equation 2). We clip all $u_t > 1$ to 1, since it would skew the QoE_{rr} towards a good score, while the agent is sending at a higher rate than the available bandwidth, thus introducing considerable delay or losses.

The delay QoE is defined as:

$$QoE_{delay} = 100 \times \frac{d_{max} - d_{95th}}{d_{max} - d_{min}} \tag{8}$$

where d_{max}, d_{min} and d_{95th} are taken from the distribution of the delay (d_t) throughout the whole trace. Note that this score takes into account only delay variation and not absolute values. However, we mitigate high delays using the reward.

The loss QoE equation is the following:

$$QoE_{loss} = 100 \times (1 - L) \tag{9}$$

where L is the mean of all l_t in a trace. Note that even when this score is 80, which seems high, it means 20% of losses, which is a low score. The final QoE metric is a weighted average of all QoE components:

$$QoE = 0.33QoE_{rr} + 0.33QoE_{delay} + 0.33QoE_{loss} \tag{10}$$

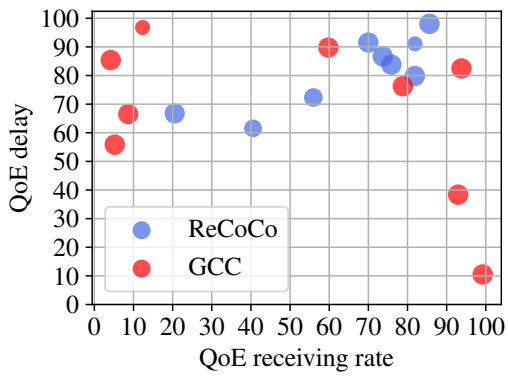


Fig. 5: Comparison of QoE between *ReCoCo* and *GCC* for each trace. The size of the circle indicates the loss QoE.

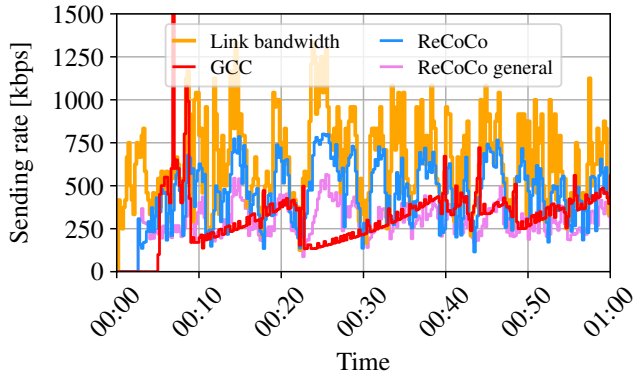


Fig. 6: Examples on *ReCoCo* v.s *GCC* sending rate on Trace *4G 700kbps*

All QoE components and the final score are on a scale [0,100].

VI. Experimental Results

In this section, we present the results of training *ReCoCo*. We first discuss the QoE when training a separate model for each trace, with their best configuration. Then we discuss the transferability of these models to other traces and the performance of a single model trained with curriculum learning.

A. Best configuration results

In Section V-C we outline all the different configuration combinations we try in our experiments. Here we present only the results of the best-performing configurations.

Table I shows the QoE of *ReCoCo* and *GCC*, for all traces. We find that SAC and TD3 exhibit much better performance than PPO in general, in every possible combination, thus PPO does not appear in the table. The algorithms prefer delayed states, which means that information on the network conditions in the near past proves useful. Out of 9 traces, *ReCoCo* exhibits better QoE_{rr} in 5, with a significant improvement for 5G traces with variable bandwidth. *ReCoCo* has a higher QoE_{delay} for 6 traces. Usually where one algorithm performs well on the receiving rate, it exhibits higher delay and vice-versa. Interestingly, *ReCoCo* shows much better results for *Wired 900kbps* and *300kbps*, which are traces with more stable bandwidth. As to QoE_{loss}, *GCC* exhibits slightly better results. *ReCoCo* prefers a slightly higher loss rate over a very high delay, which is not the case for *GCC*. Looking at overall QoE, where all components are given the same weight, *ReCoCo* outperforms *GCC* for all traces.

Figure 5 summarizes the results of Table I in a scatterplot. The x-axis is QoE_{rr}, the y-axis QoE_{delay} and the size of the circles represents the QoE_{loss}. The blue circles are *ReCoCo* and the red circles *GCC*. We see that *GCC* strongly favours either delay or receiving rate (red circles are found either on the top left corner or far right on the plot). It has decent QoE_{loss} in both cases, however it rarely optimizes for both metrics. This is expected, since it employs a controller based on delay variation and loss rate. Instead, *ReCoCo* shows many circles in the top right corner of the plot, with only some lingering with a slightly worse QoE_{rr}. This is where we can see the value of the reward function, which optimizes for all three of these metrics. Figure 6 shows an example of the sending rate vs. bandwidth in time, for the trace *4G 700kbps*. The trace bandwidth is the orange line, *GCC* is the red line and *ReCoCo* is the blue line. Another version of *ReCoCo*, discussed in Section VI-C, is the pink line. In the very variable bandwidth scenario in Trace *4G 700kbps*, we can definitely see *ReCoCo* prevail in bandwidth utilization, with very little penalty to the QoE_{delay} (just 3.09 units lower than *GCC*'s) and virtually no losses (0.16% in contrast to no loss for *GCC*).

B. Model cross-trace performance

In this subsection we discuss the performance of the best models for each trace on the other traces. This speaks to the transferability of the models. Figure 7 shows three heatmaps - one for each of the QoE components, of the QoE scores when trained on one trace and tested on another. The y-axis represents the training trace, while the x-axis the test trace. The diagonal holds the results already presented in section VI-A. Green indicates good QoE scores and red poor ones. We can see that in general, performance is bad when testing on high bandwidth and high variability traces (*Wired 35Mbps*, *5G 12Mbps* and *5G 13Mbps*), especially for QoE_{rr}. However, they show better performance when trained and tested among each other. Moreover, models trained on *Wired 900kbps*, *4G 500kbps* and *4G 700kbps* also yield good results between each other. This means that training different models based on bandwidth ranges can be a good strategy for generalization. The stable bandiwdth trace, *300kbps* proves good for model training, but hard for other models to perform well on it. Thus, another channel characteristic to take into account when generalizing could be bandwidth variability.

C. Curriculum learning

Despite some transfer ability, we still prefer to have a one-model-fits-all solution. In this subsection we show results when training one model on all traces. When using Deep RL with many different environments, such as traces with various bandwidth profiles, it is very hard to train one model

TABLE I: QoE of best-performing configurations for each trace

Trace	Configuration	QoE Receiving rate		QoE Delay		QoE Loss		Overall QoE	
		ReCoCo	GCC	ReCoCo	GCC	ReCoCo	GCC	ReCoCo	GCC
Wired 200kbps	TD3, not delayed, not tuned	85.60	93.78	98.06	82.47	100.00	100.00	94.46	91.99
Wired 900kbps	SAC, delayed, tuned	75.88	92.94	83.79	38.37	100.00	100.00	86.47	77.03
Wired 35Mbps	TD3, delayed, not tuned	20.52	8.66	66.77	66.51	99.13	99.47	62.08	58.16
4G 500kbps	TD3, delayed, tuned	69.98	78.81	91.52	76.25	99.78	99.83	87.01	84.88
4G 700kbps	TD3, delayed, not tuned	73.67	59.77	86.67	89.76	100.00	99.84	86.69	83.04
4G 3Mbps	SAC, delayed, tuned	81.91	12.32	91.07	96.79	83.51	87.38	85.41	65.43
5G 12Mbps	TD3, delayed, not tuned	55.95	4.19	72.32	85.38	94.09	99.82	74.04	63.07
5G 13Mbps	TD3, not delayed, not tuned	40.54	5.21	61.54	55.83	91.14	99.99	64.34	53.62
300kbps	TD3, delayed, tuned	81.88	99.21	79.83	10.49	100.00	100.00	87.15	69.83

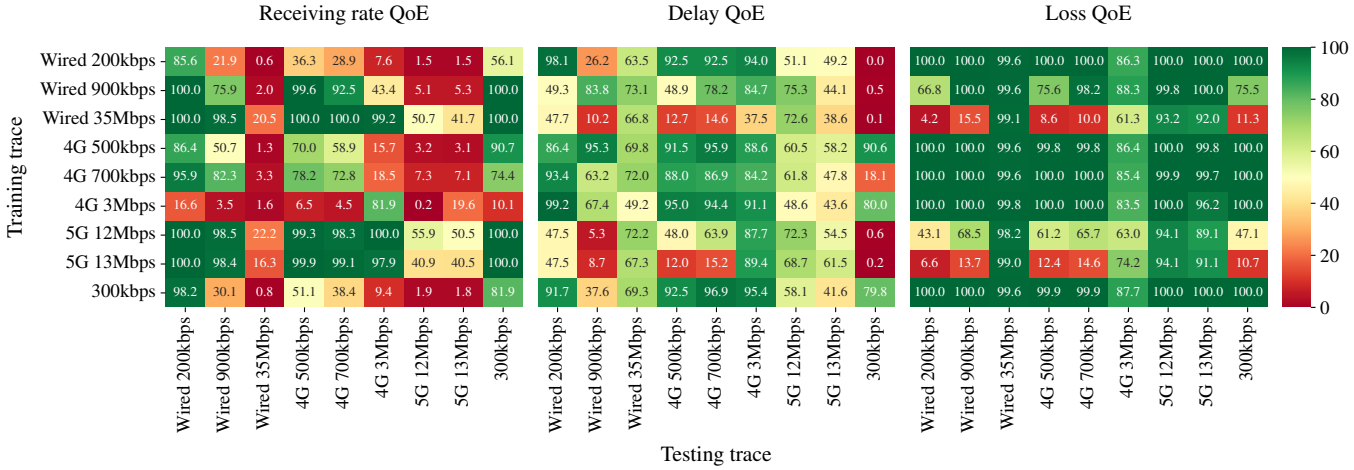


Fig. 7: QoE components when training on one trace and testing on another

that performs well in all of them. How we introduce the environments to the agent becomes an important factor [22]. Curriculum learning is a concept where, during training, we gradually increase the difficulty level of training environments, to resemble how humans learn more complex concepts [23]. Inspired by this, we try three different ways of introducing the environments to train a single model:

1) **Random**: Sample training environments at random. This is the traditionally used approach.
2) **Reward-based**: Start with the trace that obtains the lowest average reward when trained on itself (*4G 3Mbps*) and order in ascending order. We hope to imitate a growing training reward.
3) **Gap-to-baseline**: Easiest to hardest based on how much better their QoE score is against *GCC*, when trained on themselves. A concept introduced by [22].

All the traces are trained with one configuration - using the algorithm TD3, with delayed states and not tuned. Figure 8 shows the training reward, for all three training types. We run training for 2.7 Million steps, with $300k$ steps per trace. The curves are averages of 3 runs. For both reward-based (orange) and gap-to-baseline (green) training, the plot clearly shows the change of trace, with the flat areas representing the training of each trace. In both approaches, the reward grows with time. For the random trace sampling training, the reward stabilizes very early on and only grows very slightly.

The QoE scores of the three resulting models are summarized on Table II. To evaluate the QoE, we use the models trained by the mid-performer from the 3 runs, so as not to introduce a bias. Table II shows that, out of 9 traces, the model trained with the gap-to-baseline approach has superior performance for 6 traces, while the random-sampling approach for 3 traces (among which the hard case of the 5G traces). If we average out the overall QoE scores for all traces, gap-to-baseline outperforms the reward-based approach by 11 QoE units and the random approach by 1.65 QoE units. The poor performance of the reward-based approach shows the importance of evaluating against the actual QoE metrics and not just the reward.

Next, we compare the general models with the specialized models and *GCC*. Figure 6 shows an example of the sending rate for the trace *4G 700kbps*, where the pink line represents the general model trained with the gap-to-baseline methodology. We notice that, for this trace, the sending rate of the general model is a little more conservative than the specialized one, but still very comparable. For a full comparison of QoE scores, we look at both Table II and Table I. Comparing the specialized models for each trace with the single model solutions, for overall QoE scores, we observe a performance penalty of 10.4 QoE units on average for the random-sampling single model and 8.95 QoE units for the gap-to-baseline model. This is expected, since a specialized model for each trace would always outperform a general one. Comparing with *GCC*, the gap-to-baseline model is more conservative with the sending rate, so it has worse scores for QoE_{rr}, but better for QoE_{delay} and QoE_{loss}. On average, it shows an improvement

TABLE II: QoE of single models trained in different ways on all traces

Trace	QoE Receiving rate			QoE Delay			QoE Loss			Overall QoE		
	Random	Reward-based	Gap	Random	Reward-based	Gap	Random	Reward-based	Gap	Random	Reward-based	Gap
Wired 200kbps	73.38	76.75	79.58	97.31	97.99	95.55	100	100	100	90.14	91.49	91.62
Wired 900kbps	78.73	17.09	70.69	66.03	24.56	63.9	100	100	100	81.51	47.17	78.12
Wired 35mbps	0.55	0.43	1.93	68.51	66.67	72.99	99.52	99.53	99.57	56.14	55.49	58.1
4G 500kbps	65.56	31.06	75.01	90.68	93.83	89.56	100	100	100	85.33	74.89	88.1
4G 700kbps	64.12	23.06	69.02	89.6	93.8	89.4	100	100	100	84.49	72.22	86.05
4G 3mbps	13.69	5.16	9.75	85.03	87.75	95.55	89.99	86.42	84.74	62.84	59.72	63.28
5G 12mbps	1.66	1.14	4.73	80.24	50.06	70.74	100	100	100	60.57	50.35	58.43
5G 13mbps	1.53	1.07	4.36	62.06	47.56	55.92	100	100	99.97	54.48	49.49	53.36
300kbps	72.51	46.72	66.95	2.96	0	48.43	100	100	100	58.43	48.86	71.72

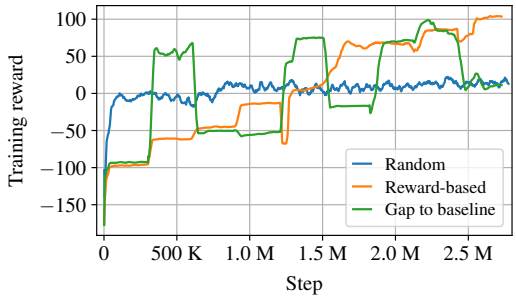


Fig. 8: Cumulative reward during training

in overall QoE over *GCC* of 0.2 units. The random-sampling model shows a performance drop over *GCC* of 1.46 QoE units. The results from the curriculum training show that, with a wide variety of training data, if we train smartly, we could obtain a decent one-model-fits-all solution.

VII. Conclusion and Future Work

In this paper, we proposed *ReCoCo*, a Reinforcement learning-based Rate controller for real-time applications. *ReCoCo* gains information about the network conditions at the receiver-side, and predicts the available bandwidth in the next time bin. We showed its performance, both in a specialized and generalized version and found it outperforms current widely adopted CC heuristics, namely GCC. We believe that the story of training *ReCoCo* provides valuable lessons for the RL for networking community, especially on transferability and generalization of RL models on different network types.

As future work, we would like to try the solutions outside a simulated environment, cover a larger variety of bandwidths and find an optimal trade-off between specialized and general models. We would also like to evaluate *ReCoCo*'s ability to continue training in the wild.

References

[1] N. Bloom, "How working from home works out," *Stanford Institute for economic policy research*, vol. 8, 2020.

[2] A. Nistico, D. Markudova, M. Trevisan, M. Meo, and G. Carofiglio, "A comparative study of rtc applications," in *2020 IEEE International Symposium on Multimedia (ISM)*, pp. 1–8, IEEE, 2020.

[3] R. Frederick, S. L. Casner, V. Jacobson, and H. Schulzrinne, "RTP: A Transport Protocol for Real-Time Applications." RFC 1889, Jan. 1996.

[4] L. De Cicco, G. Carlucci, and S. Mascolo, "Congestion control for webrtc: Standardization status and open issues," *IEEE Communications Standards Magazine*, vol. 1, no. 2, pp. 22–27, 2017.

[5] D. Vucic and L. Skorin-Kapov, "The impact of packet loss and google congestion control on qoe for webrtc-based mobile multiparty audiovisual telemeetings," in *International Conference on Multimedia Modeling*, pp. 459–470, Springer, 2019.

[6] S. Holmer, H. Lundin, G. Carlucci, L. D. Cicco, and S. Mascolo, "A Google Congestion Control Algorithm for Real-Time Communication," Internet-Draft draft-ietf-rmcat-gcc-02, Internet Engineering Task Force, July 2016. Work in Progress.

[7] J. Fang, M. Ellis, B. Li, S. Liu, Y. Hosseinkashi, M. Revow, A. Sadovnikov, Z. Liu, P. Cheng, S. Ashok, *et al.*, "Reinforcement learning for bandwidth estimation and congestion control in real-time communications," *arXiv preprint arXiv:1912.02222*, 2019.

[8] B. Wang, Y. Zhang, S. Qian, Z. Pan, and Y. Xie, "A hybrid receiver-side congestion control scheme for web real-time communication," in *Proceedings of the 12th ACM Multimedia Systems Conference*, pp. 332–338, 2021.

[9] H. Li, B. Lu, J. Xu, L. Song, W. Zhang, L. Li, and Y. Yin, "Reinforcement learning based cross-layer congestion control for real-time communication," in *2022 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pp. 01–06, IEEE, 2022.

[10] J. Eo, Z. Niu, W. Cheng, F. Y. Yan, R. Gao, J. Kardhashi, S. Inglis, M. Revow, B.-G. Chun, P. Cheng, *et al.*, "Opennetlab: Open platform for rl-based congestion control for real-time communications," *Proc. of APNet*, 2022.

[11] A. C. Begen, "Grand challenge on bandwidth estimation for real-time communications."

[12] X. Zhu, R. P. *, M. A. Ramalho, and S. M. de la Cruz, "Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media." RFC 8698, Feb. 2020.

[13] I. Johansson and Z. Sarker, "Self-Clocked Rate Adaptation for Multimedia." RFC 8298, Dec. 2017.

[14] K. MacMillan, T. Mangla, J. Saxon, and N. Feamster, "Measuring the performance and network utilization of popular video conferencing applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, pp. 229–244, 2021.

[15] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[16] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

[17] I. ITU-T, "Recommendation g. 114," *One-Way Transmission Time, Standard G*, vol. 114, p. 84, 2003.

[18] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Analysis and design of the google congestion control for web real-time communication (webrtc)," in *Proceedings of the 7th International Conference on Multimedia Systems*, pp. 1–12, 2016.

[19] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*, pp. 1861–1870, PMLR, 2018.

[20] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*, pp. 1587–1596, PMLR, 2018.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[22] Z. Xia, Y. Zhou, F. Y. Yan, and J. Jiang, "Genet: automatic curriculum generation for learning adaptation in networking," in *Proceedings of the ACM SIGCOMM 2022 Conference*, pp. 397–413, 2022.

[23] G. Hacohen and D. Weinshall, "On the power of curriculum learning in training deep networks," in *International Conference on Machine Learning*, pp. 2535–2544, PMLR, 2019.