Internet Congestion Control via Deep Reinforcement Learning

Nathan Jay*[,1], Noga H. Rotman*[,2], P. Brighten Godfrey[1], Michael Schapira[2], and Aviv Tamar[3]

[1]University of Illinois at Urbana-Champaign, [2]Hebrew University of Jerusalem, [3]UC Berkeley

Abstract

We present and investigate a novel and timely application domain for deep reinforcement learning (RL): Internet congestion control. Congestion control is the core networking task of modulating traffic sources' data-transmission rates so as to efficiently utilize network capacity. Congestion control is fundamental to computer networking research and practice, and has recently been the subject of extensive attention in light of the advent of Internet services such as live video, augmented and virtual reality, Internet-of-Things, and more.

We build on the recently introduced Performance-oriented Congestion Control (PCC) framework to formulate congestion control protocol design as an RL task. Our RL framework opens up opportunities for network practitioners, and even Internet application developers, to train congestion control models that fit their local performance objectives based on small, bootstrapped models, or complex, custom models, as their resources and requirements merit. We present and discuss the challenges that must be overcome so as to realize our long-term vision for Internet congestion control.

1 Introduction

In today's Internet, multiple network users contend over scarce communication resources. Consequently, the data-transmission rates of different traffic sources must be modulated so as to both efficiently utilize network resources and to achieve good user experience. This challenge is termed "congestion control" and is fundamental to computer networking research and practice. Congestion control is amongst the most extensively studied topics in computer networking and, as Internet services and applications become ever more demanding (live video, AR/VR, edge computing, IoT, etc.), and the number of network users steeply rises, is ever increasing in importance. Indeed, recent years have witnessed a surge of interest in the design and analysis of congestion control algorithms and protocols (see, e.g., [26, 27, 6, 4, 7]).

Consider multiple *connections* (also referred to as "*flows*" hereafter) sharing a *single* communication link, as illustrated in Figure 1. Each connection consists of a traffic sender and a traffic receiver. The sender streams packets to the receiver and constantly experiences feedback from the receiver for sent packets in the form of packet-acknowledgements (ACKs). The sender can adjust its transmission rate in response to such feedback. The manner in which the sending rate is adjusted is determined by the *congestion control protocol* employed by the two end-points of the connection. The interaction of different connections gives rise to network dynamics, derived from the connections' congestion control protocols, the link's capacity (bandwidth), and the link's buffer size and "packet-queueing policy", which determines how (and whose) excess traffic should be discarded.

*Equal contribution

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

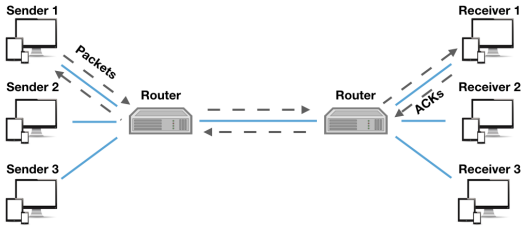


Figure 1: Multiple traffic flows sharing a link

Even the above simple single-link scenario illustrates the complexity of congestion control. Different connections select rates in an uncoordinated, decentralized manner. Also, with few exceptions, a connection's congestion control decisions are oblivious to the number of other connections competing over the link, at what times these connections enter and leave the network, what packet-queuing policy is realized at the link, etc. These challenges are further complicated by the fact that, in practice, different connections might employ different congestion control protocols, and network environments and segments greatly vary in terms of sizes, link capacities, network latency, level of competition between connections, etc. Consequently, even after three decades of research on Internet congestion control, heated debates linger about the "right" approaches to protocol design [16].

Congestion control protocols typically fall into two main categories: (1) protocols designed (either handcrafted [27, 2]) or automatically generated, e.g., Remy [26]) for a specific network environment, or a predetermined range of such environments (say, mobile networks, satellite networks, datacenter networks, etc.), and (2) "all purpose" protocols designed to perform well across a broad range of environments, e.g., PCC [6, 7]. While protocols in the first category might achieve high performance when the network matches their design assumptions, they can suffer from poor performance when this is not so. In contrast, an all-purpose protocol not tailored to a specific network environment might naturally be significantly outperformed by a protocol designed specifically for that context [15].

Can we have the best of both worlds? Can a congestion control protocol both *robustly* provide good performance *and* adapt online so as to optimize performance with respect to the *prevailing* network conditions? We argue that the answer to this question is "Yes", and that the key lies in employing deep reinforcement RL to guide congestion control.

Under RL, a decision maker (agent) continuously adapts her policy, which maps locally-perceptible states to choices of actions, in response to empirically experienced performance ("rewards"). When applied to congestion control, this translates to dynamically adapting the *rules* for mapping feedback from the network and receiver of traffic (e.g., throughput, packet loss, latency, etc.) to choices of sending rates.

RL is *inherently* customizable, in the sense that it adapts the mapping from state to actions so as to maximize the experienced rewards. Recent developments in RL employ deep neural networks to learn complex patterns in the experienced state, and identify choices of actions that lead to high reward (deep RL [17, 18]). In addition, in some contexts, deep RL has also been shown to *generalize* well, i.e., to effectively apply knowledge acquired from past experiences to new environments. We conjecture that (deep) RL can be employed to learn good congestion control strategies, which not only adapt to optimize performance under the actual network conditions, but also react well to changes in the network environment and provide high performance across variable networks.

We formulate a novel framework for RL-based congestion control protocol design, which extends the recently introduced Performance-oriented Congestion Control (PCC) approach [6, 7]. We utilize this framework to design Custard (CUSTomized And Robust Decision). Custard employs deep RL [21, 17] to generate a policy for mapping observed performance-related statistics (e.g., packet-loss rate, averaged latency) to choices of rates. Our preliminary evaluation results suggest training Custard on relatively few, simple environments is sufficient for generating congestion control policies that perform well also in very different network domains.

Our RL framework for congestion control creates new opportunities for network operators, and even application developers, to train congestion control models that fit their performance objectives. These models can be specialized to specific network conditions through training, and specific objectives

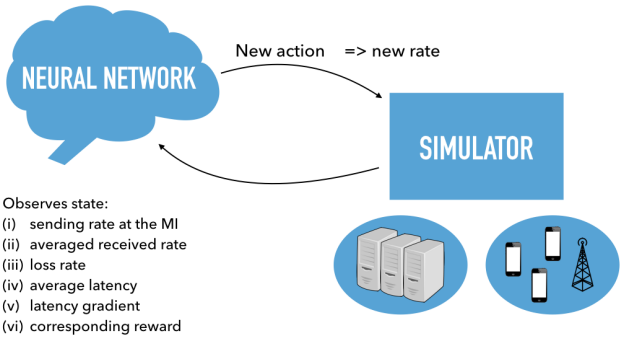


Figure 2: The role of each RL component in Custard

based on a reward function, providing multiple means of customization. We present and discuss the challenges that must be overcome so as to realize our vision.

2 RL Approach to CC

We next provide a high-level overview of RL and then explain how congestion control can be formulated as an RL task. Our RL formulation of congestion control extends the recently introduced Performance-oriented Congestion Control (PCC) [6, 7].

2.1 Reinforcement Learning

In RL, an *agent* interacts with an *environment*. The agent has a set of *actions* she can choose from, and her choice of actions can influence the *state* of the environment.

At each discrete time step $t \in 0, 1, ...$, the agent observes a (locally perceptible) state of the environment s_t, and selects an action a_t. At the following time step $t+1$, the agent observes a *reward* r_t, representing her loss/gain after time t, as well as the next state s_{t+1}. The agent's goal is to choose actions that maximize the *expected cumulative discounted return* $R_t = \mathbb{E}\big[\sum_t \gamma^t \cdot r_t\big]$, for $\gamma \in \big[0, 1\big)$. The parameter γ is termed the *discount factor*.

RL has been successfully applied to solving complex problems such as robotic manipulation [11], computer games [14], and, more recently, to resource scheduling [12], video delivery [13] and routing [23].

2.2 Congestion Control as RL

Formulating congestion control as an RL task requires specifying the actions, states, rewards, etc.

Actions are changes to sending rate. Intuitively, in our formulation, the agent is the sender of traffic and her actions translate to changes in sending rates. To formalize this, we adopt the notion of *monitor intervals* (MIs) from [6, 7]. Time is divided into consecutive intervals. In the beginning of each MI t, the sender can adjust her sending rate x_t, which then remains fixed throughout the MI. After experimenting with several options, we chose to express actions as *changes* to the current rate (see Section 3.1 for details).

States are bounded histories of sending rates and resulting statistics. After the sender selects rate x_t at MI t, she observes the results of sending at that rate and computes statistics such as goodput, packet loss rate, average latency, etc., from received packet-acknowledgements. We denote the vector of statistics resulting from the sending rate at MI t by v_t. We restrict our attention below to *statistics vectors* that consist of the following elements: (i) sending rate at the MI, (ii) averaged received rate, (iii) loss rate, (iv) average latency (v) latency gradient [7], and (vi) corresponding reward (see discussion below).

The agent's selection of the next sending rate is a function of a *fixed-length history* of previously chosen sending rates and the statistics vectors resulting from sending at these rates. Including a small

history, instead of just the most recent statistics, might allow our agent to detect trends and changing conditions and react more appropriately. Thus, the state at time t, s_t, is defined to be:

$$s_t = (v_{t-(k+d)}, \ldots, v_{t-d})$$

for a predetermined constant $k > 0$ and a small number d representing the delay between choosing a sending rate and gathering results. We discuss how the length of the history, i.e., k, affects performance in Section 3.3.

Setting rewards. The reward resulting from sending at a certain rate at a certain time may depend on the performance requirements of the specific application; some services might prefer a lower-but-constant bandwidth, while others may desire higher bandwidth and are more tolerant to bandwidth variation. In general, reward functions should increase with throughput and decrease with latency or loss rate. We discuss specific reward functions in Section 4.2.

3 Introducing Custard

3.1 Architecture

RL inputs and outputs. As discussed in Section 2.2, our RL agent maps fixed-size histories of statistics vectors s_t to changes in sending rate. Statistics for each MI are computed from selective packet-acknowlegements (SACKs) from the receiver, similar to [6, 7]. The agent's output is a real value a_t that we interpret as the following change to the previous sending rate x_{t-1}:

$$x_t = \begin{cases} x_{t-1} * (1 + \alpha a_t) & a_t \geq 0 \\ x_{t-1} * (1 - \alpha a_t) & a_t < 0 \end{cases}$$

where α is a scaling factor used to dampen oscillations (we use $\alpha = 0.01$).

Neural network. Our RL agent maps real-valued inputs (statistics vectors) to real values (changes in sending rate). We use a neural network for this task [14, 17, 20]. Neural networks come in many shapes and sizes and so choosing the optimal architecture could be a complex process. We show, however, as a starting point, that even a very simple architecture, namely a small, *fully connected* neural network, produces good results. We tested several combinations of the number of hidden layers, as well as the number of neurons per layer. Our experimentation showed that a neural network with three hidden layers and 32 neurons per layer worked consistently well throughout our evaluation process (see Section 4).

3.2 Training

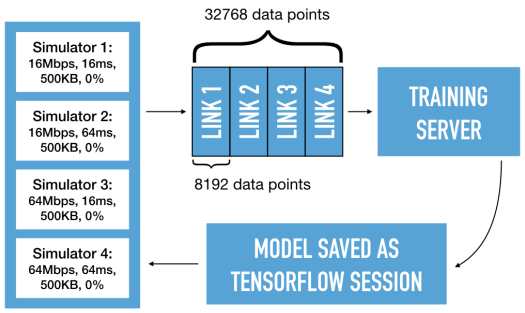


Figure 3: Custard's framework. Four distinct simulators, each running a link with different parameters and a copy of our agent.

We train our agent using multiple simulators in parallel to reduce training time. Each simulator simulates only a single link and sends data to our training server. Once our training server has received a dataset of 8192 monitor intervals from each simulator, it updates our agent's policy using the Trust Region Policy Optimization (*TRPO*) implementation provided by OpenAI Baselines [5] and sends the updated policy to the simulators, as shown in Figure 3. We train on just four links that vary only in capacity and latency. Interestingly, when trained on the combination given, our agent

learned policies that work for a range of link capacities, latencies and buffer sizes. More complex training may be appropriate in some cases, but our evaluation (see Section 4) shows that even this simple training procedure produces a fairly general model.

3.3 Choice of Parameters

More than a dozen parameters and some degree of randomness affect the final model we produce at the end of training. We next discuss two significant parameter choices: history length (how many past monitor intervals are used as the agent's input), and discount factor (the degree to which expected future rewards affect the current rate decision).

Before we can address the effect of parameters on the resulting model, we need to specify a reward function. Here, we use a simple linear reward function: $reward(t) = throughput_t - rtt_t - 10^8 loss_t$

where t is the time of the monitor interval, throughput is measured in megabits per second, rtt is measured in milliseconds and loss rate is the proportion of packets lost, between 0 and 1. This function says that a gain of 1Mbps is worth 1ms of increased latency or a 1% increase in packet loss. We use this reward function here and in parts of the evaluation because it is simple. We examine others in Section 4.2, and expect reward functions to vary based on application needs.

History length. A history length of k means that the agent makes a decision based on the k latest monitor intervals worth of data. Intuitively, increasing history length should increase performance because extra information is given. Figure 4 shows this effect. The agent with 100 monitor intervals of history has the greatest reward, but only by a small margin, therefore we decided to use the agent with just 10 monitor intervals of history during our evaluation.

Discount factor. For γ near 0, decisions are made almost exclusively based on the immediate expected reward. For γ near 1, the overwhelming factor in decision making is the long term expected award. Figure 5 shows the effect of γ on the training reward. The effect here was surprisingly small. We attribute this tiny effect to our reward function and the stability of our training links. With just one sender per link, and links that have perfectly stable capacity, the ideal policy will always change its rate towards link capacity. In scenarios with varied capacity, when a sender might achieve higher throughput by keeping a small amount of data in the network buffers, this may not be the case. We use $\gamma = 0$ for most of our evaluation, but re-visit the choice of γ when discussing other reward functions in Section 4.2.

4 Evaluation

Custard's framework allows anyone to train a custom congestion control agent based on a network and reward function of interest. In this section, we use two groups of tests to investigate a small set of agents that were trained in a simulated setting. First, we test the robustness of a simple model with a simple objective. We identify the operating range of link capacities and latencies (16-64Mbps and 16-64ms) of our model (as described in Section 3) and examine the model's behavior in and around this range. Second, we test Custard's ability to specialize to different reward functions by training two customized models (using the same training from Section 3) with reward functions that have

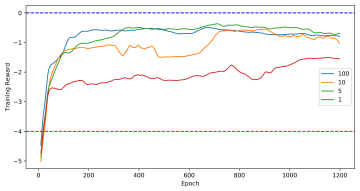


Figure 4: Training reward for agents with different history lengths. The red dashed line indicates the reward given for sending at 0Mbps. Below this line, agents are performing worse than sending nothing at all. The blue dashed line is the optimal reward, computed by assuming the agent sends at exactly link capacity with no increase in latency or loss. In practice, this is unattainable. Each line represents the average of three models.

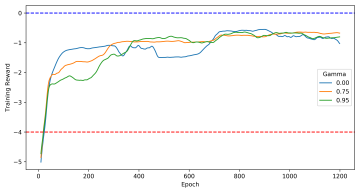
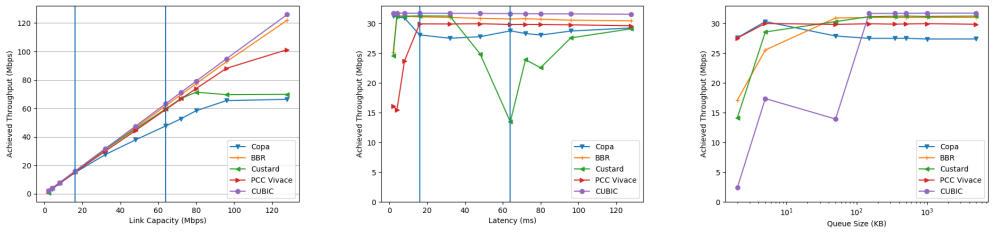


Figure 5: Training reward for agents with different values of gamma. Dashed lines shown as in Figure 4. Each line is the average of three models.

been presented in recent congestion control work. We then compare the performance of our Custard agents with congestion control algorithms that are intended to optimize those reward functions.

The above experiments use Pantheon tunnels [28] over Emulab links [25], with five trials for each data point. We also test a variety of modern congestion control schemes for reference.

4.1 Robustness



(a) Bandwidth sensitivity **(b) Latency sensitivity** **(c) Buffer size sensitivity**

Figure 6: Tests over a single link, showcasing the model's sensitivity to changes in bandwidth, latency and buffer size. Plotted here is the throughput of each algorithm tested against changes in the relevant parameter.

In Figure 6, we show how our model behaves as bandwidth, latency and buffer size vary from several binary orders of magnitude below to an order of magnitude or more above our agent's training conditions. Each test was run for two minutes with a single sender over a single link. For each test, we compare the results of our model against TCP CUBIC [8], BBR [4], Copa [3], and PCC Vivace [7].

Bandwidth sensitivity. Our model was trained on only 16Mbps and 64Mbps bandwidth links, but any reasonable congestion control should operate in a wider range. Here, we show that range by examining the model's sensitivity to bandwidths outside of its training conditions. To test bandwidth sensitivity, we configured Emulab links with 32ms latency, a 500KB queue and a 0% random loss rate, in line with our training values for those parameters. Then, we configured the bandwidth differently for each test, ranging from 2 to 128Mbps. Our model achieves 80% throughput up to 80Mbps, somewhat above its training range. Even as low as 2Mbps, three orders of magnitude below the training range, our model achieves near-capacity throughput with low self-inflicted latency, suggesting that training gives an upper bound on model capability for bandwidth, but no real lower bound. This implies that we could have trained our model for much higher bandwidths to increase its operating range.

Latency sensitivity. Again, our model was trained on only 16ms and 64ms latency links, but we would like to know its latency sensitivity across a wider range. To test latency sensitivity, we configure links within our training range for all other parameters (32Mbps, 500KB queue and 0% random loss). Each test is performed with a different base latency, ranging from 2 to 128ms. We find that our model performs poorly at both low and high latencies, but has an extreme dip in throughput when run on a link with a base latency between 40 and 80ms. Inspecting the individual flows graphs from these runs showed that our Pantheon over Emulab environment had 2 to 3ms of noise in the base latency, which caused our agent to reduce its sending rate. Running the 64ms latency test in our simulator showed near-capacity throughput. Our model may be resilient to changes in link parameters, but it can suffer significantly from even minor changes in the environment, as this demonstrates.

Our model was trained on links with only 500KB buffers. Still, it shows high performance even with a buffer just 1% of that size. We ran buffer sensitivity tests on 32Mbps links with 32ms latency and 0% random loss. The results of these tests appear in Figure 6c. As expected, CUBIC performs poorly at a very low buffer size (2KB, just one packet). Our model still manages about 50% of the maximum throughput with a one packet buffer, and more than 80% for all other buffer sizes tested.

Multiflow competition. In all its training, our agent neither competed nor cooperated with another sender, so we had little expectation for its multiflow performance. Figure Figure 7 shows Jain's Fairness Index computed for a set of competing senders with various numbers of competitors for several congestion control schemes. With just two senders, our agent is extremely unfair, with one sender taking a greater and greater share of the bandwidth after the two senders initially diverge by only a small amount. With more senders though, our agent is actually more fair. We suspect this is due in large part to randomness in the timing of the senders. With just two senders, the one that loses bandwidth always gives it to the same opponent. With four senders, the extra bandwidth might be given away randomly or distributed among competitors, preventing one flow from dominating as significantly. Improving our agent's robustness to multiflow competition is an interesting avenue for future work, a point of discussion in Section 6.

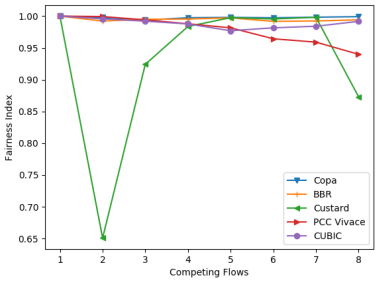


Figure 7: Fairness of various congestion control schemes on a 32Mbps, 32ms latency link with 0% random loss and a 500KB buffer. Each test was run for two minutes.

4.2 Previously Proposed Reward Functions

Recent algorithms have proposed utility functions that their congestion control algorithm is intended to optimize for. We train our agent to optimize two such functions (the "power" function, a common objective in congestion control, and a version of which was used in Remy and Copa [26, 3], R_{power}, and a loss-based utility function presented with PCC-Vivace, R_{V-loss} [7]). We compare our RL approach to the original congestion control algorithms.

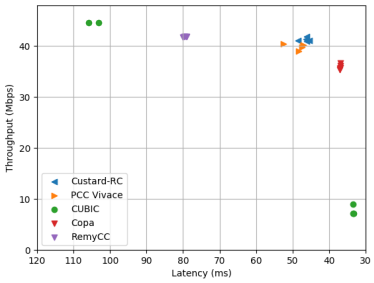
$$R_{power} = \frac{throughput}{latency}, \qquad R_{V-loss} = throughput^{0.9} - 11.35 * throughput * loss$$

Comparing our RL approach to the original algorithms on a single static link yields uninteresting results because such links have essentially no tradeoff between throughput and loss or latency, so we create a dynamic link whose bandwidth varies uniformly in the range from 16 to 64Mbps, changing once every five seconds. We use the same random seed to create link variations for all tests.

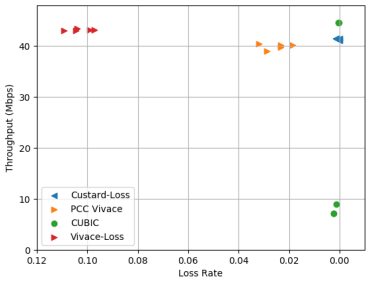
Figure 8a shows the throughput to latency tradeoff made by a RemyCC-generated algorithm provided by Pantheon [28], Copa, Vivace-Loss, and Custard, with CUBIC shown as a baseline. RemyCC and Custard achieve similar throughput, but Custard has an average latency of 47ms, while RemyCC's average latency is about 79ms. While Custard clearly has a preferable operating point to RemyCC, Copa's operating point may be more desirable. With just 12% lower throughput, Copa operates at just 36ms average latency. The description of the utility function presented with Copa suggests that it optimizes queuing delay [3], which is just 4ms for Copa compared to 15ms for Custard.

CUBIC performance is split between high throughput and very low throughput. Examining these traces shows that the high throughput CUBIC flows have loss rates of about 0.02%, while the low throughput flows have loss rates of about 0.15%. We suspect this may be an artifact of frequent link configuration changes, as the CUBIC tests in Figure 6 show much more typical performance.

Figure 8b shows the throughput to loss rate tradeoff made by PCC-Vivace (with loss-based reward) and Custard. Custard obtains about 2Mbps lower throughput than Vivace-Loss, but suffers near zero loss,



(a) Custard, Copa and RemyCC are all optimized for a similar utility function.



(b) Custard optimizes a loss-based reward function better than Vivace-Loss, and manages to have both higher throughput and lower loss than Vivace-Latency.

Figure 8: Performance on a dynamic link within its training range (16-64Mbps, 32ms latency, 500KB buffer and 0% loss rate).

compared to Vivace-Loss's 10% loss (in part induced when the link capacity suddenly drops while the buffer is full). This tradeoff gives Custard an average reward score of 28, while Vivace-Loss's reward is -20. PCC-Vivace's default reward function includes a latency gradient term that prevents it from inducing high loss, but the result of that reward function depends on exactly the interval over which it is computed, making it difficult to compare with. Still, we plot the results of Vivace-Latency, which shows both lower throughput and higher loss than Custard in this test.

Reaction to Early Signals of Congestion The comparison of our loss only reward function makes one benefit of our RL approach clear: instead of waiting until undesirable congestion occurs (as Vivace-Loss does with loss), our agents can react to earlier signs of congestion (like increasing latency), and completely avoid the problem. Figure 8b shows just how significant this improvement can be. The PCC family of protocols *react* to decreasing reward. Given the proper inputs, our agent can *predict* a decreasing reward and act to avoid the problem. We discuss this further in Section 6.

5 Related Work

There have been several attempts (mostly over a decade ago) to apply RL to congestion control in specialized domains. These cannot be easily generalized to model the problem of CC outside of the specific settings considered. An early endeavor focused on ATM networks [22]. [10] employs RL to create a cooperative congestion control controller for multimedia networks. [19] and [9] focus on multimedia applications. [1] explores designing a TCP-style congestion control algorithm using Q-learning [24]. [29] employs RL to solve congestion problems in wireless sensor networks.

Remy [26] generates offline congestion control algorithms for an input model of the network and traffic conditions. Unlike RL-based approaches, the Remy solution is static; if the actual network conditions change, performance could potentially degrade substantially. Custard, on the other hand, can continue to be trained online. Our evaluation, although small scale, shows that Custard substantially outperformed Remy in latency, suggesting that deep RL is more effective than Remy's learning approach in some scenarios.

PCC [6, 7] employs online learning to guide congestion control. While online learning congestion control provides valuable *worst-case* guarantees (namely, no regret [7]), unlike Custard, it does not learn the prevailing network regularities and adapts accordingly. Thus, while PCC does provide robustness, it does not automatically customize to the experienced network.

6 Conclusion and Future Research

We presented Custard, a congestion control protocol powered by a deep RL mechanism. In this work we tested both the robustness and specialization of Custard. Our evaluation showed that Custard was fairly robust with respect to link capacity, latency and buffer size, but we identify room for improvement in multiflow competition. Our specialization test showed that Custard was capable of

optimizing previously proposed reward functions, even better than PCC-Vivace and RemyCC, and comparably to Copa.

Our work has provided a foundation and showed promising results for RL-based congestion control, but our evaluation is preliminary and leaves open a variety of interesting questions.

Acknowledgments

The fourth author is funded by the Israel Science Foundation (ISF). We thank Huawei for ongoing support of the PCC project.

References

[1] Akshay Agrawal. Xavier: A reinforcement-learning approach to tcp congestion control, 2016.

[2] Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Dctcp: Efficient packet transport for the commoditized data center. 2010.

[3] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. USENIX{ Association}, 2018.

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

[5] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

[6] Mo Dong, Qingxi Li, Doron Zarchy, Philip Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *NSDI*, volume 1, page 2, 2015.

[7] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 343–356. USENIX{ Association}, 2018.

[8] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[9] Ali Hamieh, Yoohwan Kim, and Ju-Yeon Jo. On using reinforcement learning techniques for congestion control in multimedia applications. Technical report, University of Nevada Las Vegas, 08 2013.

[10] Kao-Shing Hwang, Cheng-Shong Wu, and Hui-Kai Su. Reinforcement learning cooperative congestion control for multimedia networks. In *Information Acquisition, 2005 IEEE International Conference on*, pages 6–pp. IEEE, 2005.

[11] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *JMLR*, 17, 2016.

[12] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016.

[13] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM, 2017.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[15] Michael Schapira. Network-model-based vs. network-model-free approaches to internet congestion control. In *IEEE International Conference on High Performance Switching and Routing*. IEEE, 2018.

[16] Michael Schapira and Keith Winstein. Congestion-control throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 122–128. ACM, 2017.

[17] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

[20] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[21] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[22] Ahmed A Tarraf, Ibrahim W Habib, and Tarek N Saadawi. Reinforcement learning-based neural network congestion controller for atm networks. In *Military Communications Conference, 1995. MILCOM'95, Conference Record, IEEE*, volume 2, pages 668–672. IEEE, 1995.

[23] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 185–191. ACM, 2017.

[24] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[25] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[26] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *SIGCOMM Comput. Commun. Rev.*, 43(4):123–134, August 2013.

[27] Keith Winstein, Anirudh Sivaraman, Hari Balakrishnan, et al. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, volume 1, pages 2–3, 2013.

[28] Francis Y Yan, Jestin Ma, Greg Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. *Measurement at http://pantheon. stanford. edu/result/1622*, 2018.

[29] SAOSENG THU WING ONE JEAN YVES. Machine learning based congestion control in wireless sensor networks. Master's thesis, National University of Singapore, 2008.