On a Deep Q-Network-based Approach for Active Queue Management

Dhulfiqar A. AlWahab*†, Gergő Gombos†, Sándor Laki†
*3in Research Group, ELTE Eötvös Loránd University, Martonvásár, Hungary
†Communication Networks Laboratory, ELTE Eötvös Loránd University, Budapest, Hungary
{aalwahab, ggombos, lakis}@inf.elte.hu

Abstract—**Reinforcement learning has gone through an enormous evolution in the past ten years. It's practical applicability has been demonstrated through several use cases in various fields from robotics to process automation. In this paper, we examine how the tools of deep Q-learning can be used in an AQM algorithm to reduce queuing delay and ensure good link utilization at the same time. The proposed method called RL-AQM has the advantage that it is less prone to the good parameterization and can automatically adapt to new network conditions. The prototype implementation based on OpenAI Gym and NS-3 network simulator has thoroughly been evaluated under various settings focusing on three aspects: the convergence time of learning process, the performance of pre-trained models compared to PIE AQM and generalization ability. We have demonstrated that RL-AQM achieves comparable utilization to PIE AQM but results in much smaller queueing delays. Finally, the pre-trained models have good generalization abilities, enabling to use a pre-trained model in network settings that differ in bandwidth and/or RTT from the one used during the pre-training phase.**

Index Terms—**AQM, reinforcement learning, DQN, queueing delay.**

I. Introduction

End-to-end congestion control of TCP can handle network congestion, but it builds up queues along the network path, leading to large queuing delays that are not tolerated by most recent applications. Active Queue Management (AQM) has been introduced to solve the bufferbloating problem by controlling queueing delays. AQM algorithms proactively start dropping packets or marking them with ECN Congestion Experienced (CE) even before the queue becomes full, enforcing sources to reduce their sending rates. They are usually deployed in routers and switches and deal with traffic aggregates flowing through the bottleneck link.

In the past decade, many AQM proposals like RED, PIE, or CoDel have emerged. The key problem with current AQM algorithms is that their performance largely depends on their parameters and sometimes finding the optimal settings is not trivial. One method with a specific setting can outperform others in a network environment while others can provide better performance in another environment. AQM algorithms are usually tuned by extra parameters like drop rate, queue maximum threshold, target time whose optimal values may be varied form network to network, regarding the actual traffic and network conditions [1], [2]. This recognition led the researchers to parameter-less or auto-tuning AQM proposals [3], [4].

In parallel, artificial intelligence and machine learning have also gone through an enormous evolution. Reinforcement learning proved to be useful in many practical areas from high level control of robots to low-level control of processes. One of their benefits compared to static models is that they can adapt to the changes in the environment, using the well-known trial-and-error approach. A learning agent continuously monitors the state of the environment, using the state information it applies an action that affects the behavior of the environment, evaluate the decision using a reward function, and finally reinforce the internal model according to the goodness of the applied action.

In this paper, we propose an AQM algorithm called RL-AQM that exploits the idea of reinforcement learning. The proposed method consists of two components: 1) a simple queuing discipline (QDisc) that applies a given drop probability at packet arrival, 2) an agent that sets the drop probability applied by the QDisc. More specifically, the agent periodically obtains information about the queue and bottleneck link states from which it computes an action to be executed that modifies the QDisc's drop probability. One can observe that this design fits well to the general view of software defined networking (SDN) and programmable data planes like P4 [5] where the learning agent can be part of the control plane while the data plane is responsible for assembling state information and implementing the simple QDisc mechanism.

We have implemented the proposed RL-AQM in NS-3 network simulator using its OpenAI Gym extension [6]. The prototype has thoroughly been evaluated under various settings to demonstrate that RL-AQM achieves good utilization while results in small queueing delays, and its pre-trained internal model has a good generalization ability.

The paper organized as follow: Section II gives a short overview of the related work. In Section III we present our system the RL-AQM and we evalute in Section IV. Finally, in Section V we make some conclusion.

II. Related Work

In [7], Reinforcement Learning Gradient-Descent (RLGD) AQM scheme has been proposed to control the queue length and to maximize the throughput. Simulation results show that the RLGD can achieve the stability of the queue length under various network conditions in a shorter jitter time and more robust than the traditional PI and REM controllers. In [8] QRED introduced, a Q-learning algorithm used to optimize the original maximum dropping probability calculation method in RED. Results demonstrate QRED improves the overall network performance and reduce the parameter sensitivity issue of RED. The work in [9] also had presented a new AQM algorithm based on reinforcement learning to increase the network performances, the proposed algorithm in this work (RL-QDL) performances better than RED in two simulation-scenarios. This work also mentioned that RL can be used to support QoS in all IP-networks. In [10], the reinforcement signal (reward) is used to learn the congestion control from experience with no prior knowledge about the network dynamics. The usage of the RL with the Kanerva coding algorithm in the agent is called QTCP, in this work, which achieved higher throughput and lower delay than TCP-Reno and QTCP-Baseline. In [11] the authors gave an overview about the use of RL in a different area like wireless networks. In [6] two well-known systems namely the NS3 and OpenAI Gym were combined to produce a benchmarking system called NS3-Gym. It simplifies feeding the RL with the data from the network in the NS3 simulation and we also use this environment in this paper. In contrast to prior work, this paper does not rely on existing AQM schemes like RED and does not only tune the parameters of such schemes with RL. Instead, we propose a very simple AQM scheme that can be implemented in nowadays P4-programmable switches, and a learning agent that sends an action modifying the drop probability to the AQM component periodically (in every 1-20 ms). The agent relies on the celebrated model of RL called Deep Q-Learning [12].

III. AQM with Reinforcement Learning

The methods of active queue management aim at reducing queeuing delay while keeping the bottleneck link fully utilized. To solve these objectives they continuously monitor both the queue and link states and make decisions on dropping or forwarding packets. Note that instead of dropping, AQMs can also mark packets with ECN Congestion Experienced (CE), reducing the number of re-transmissions in this way. In this paper, we propose RL-AQM, an AQM algorithm that uses the tools of reinforcement learning to control the applied drop probability and thus to find a good trade-off between high link utilization and low queueing delay. The implementation of our method is based on the OpenAI Gym extension of NS-3 network simulator [6].

TABLE I
Possible actions

a_0	do not modify the drop probability
$a_\uparrow$	increase drop probability by 0.01
$a_\downarrow$	decrease drop probability by 0.01
$a_{\uparrow\uparrow}$	increase drop probability by 0.1
$a_{\downarrow\downarrow}$	decrease drop probability by 0.1

According to the traditional RL approach [12], [13], the environment that represents the AQM method running inside a router can be modeled as a Markov Decision Process. The process is represented by a five-tuple (S, A, T, R, γ) where S is the state space, A is the action set, $T(\underline{s}'|\underline{s}, a)$ is the state transition probability where $\underline{s}', \underline{s} \in S$ and $a \in A$, R is the reward function while $\gamma \in [0, 1)$ is the discount factor. In our method illustrated by Figure 1, $\underline{s} \in S$ represents a queue state at a given point of time. $\underline{s}$ is a tuple composed of three factors: queuing delay, bottleneck link utilization and applied drop probability. We have introduced 5 actions described in Table I through which the applied drop probability of the AQM method can be modified with different intensities. In this system model, the AQM policy is a function mapping every state to a distribution over actions: $\pi : S \longrightarrow A$. When we follow a policy π from state $\underline{s}_t$ at time t, the value function can be calculated as the sum of rewards of succeeding states: $V^\pi(\underline{s}_t) = E[\sum_{i=0}^{T} \gamma^i R(\underline{s}_{t+i}, \pi(\underline{s}_t))]$. The value function ranks the policies according to the cumulative reward. The value of a single step can be calculated similarly by the Q-value function. $Q^\pi(\underline{s}_t, a)$ represents the expected value starting from $\underline{s}_t$, taking action a and then following π. Q-value function can recursively estimated by the Bellman equation: $Q^\pi(\underline{s}_t, a) = E[R(\underline{s}_t, a) + \gamma \max_{a'} Q^\pi(\underline{s}_{t+1}, a')]$. The optimal policy have the highest Q-value function over all policies. In Deep Q-Networks (DQN), the Q-value function is approximated by a neural network and the reinforcement learning process uses the previously introduced Bellman equation.

The architecture of the proposed method is depicted in Figure 1. It consists of two components: 1) the OpenAI-based learning agent, and 2) the NS-3 simulation environment. The NS-3 environment implements the simulation scenarios using a dumb-bell topology with various traffic intensities and link properties (bandwidth and RTT), and the frame of the proposed RL-AQM method as a queuing discipline (QDisc in NS-3 terminology) that simple drops packets at arrival with a specific probability, maintains state information and evaluates the reward function. The business logic is fully implemented by the Agent. The simulator is connected to the Agent and reports the current state of the QDisc (queueing delay, utilization and actual drop probability) periodically, in every 20 ms in our setting. It also computes the reward $(R(\underline{s}_t, a_t))$ for the simulation period elapsed since the last update of drop probability (using action a_t at time t). The Agent takes the computed reward and the new state $\underline{s}_{t+1}$, and applies the Bellman equation to update the DQN model. Then the new state $\underline{s}_{t+1}$ is fed into the DQN to calculate the Q-values $Q(\underline{s}_{t+1}, a)$ for each action a. The candidate action to be performed is the one with the highest Q-value: $a_{t+1} = \arg_a \max Q(\underline{s}_{t+1}, a)$. To determine the action to be executed we also apply further heuristics and random selection.

A. Action selection

Packets arrive in bursts at the bottleneck and thus they result in temporal loads that are handled by the AQM methods. However, between two such bursts the link is not overloaded and the queue is almost always empty. When we first applied RL-AQM, it showed a very slow convergence since the DQN also learned Q-values for the trivial cases. To fasten learn-

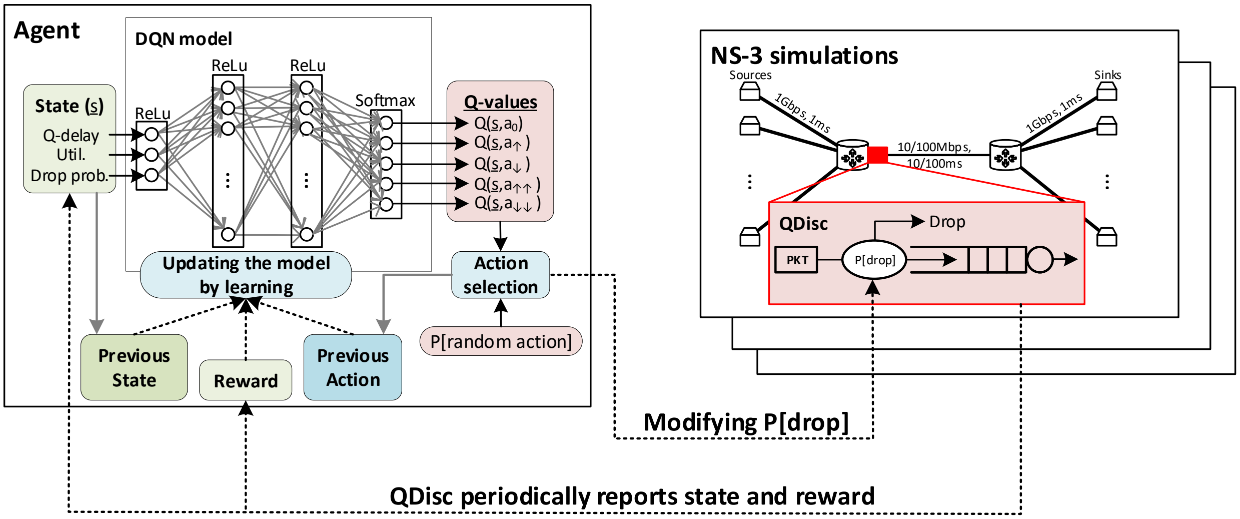

Fig. 1. RL-AQM architecture with a learning agent and NS-3 components.

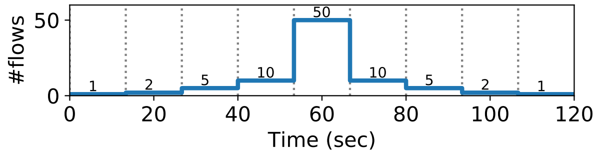

Fig. 2. The simulation is split into 9 sections with various traffic intensities.

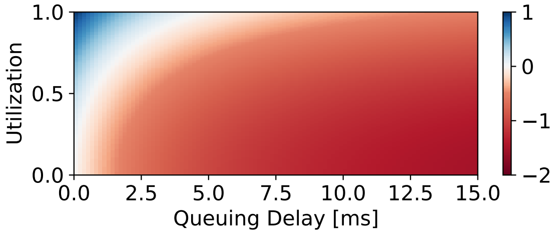

Fig. 3. Reward function.

ing process, we introduced two simple heuristics: 1) if the queueing delay is less than 1 ms and the drop probability is 0, action a_0 is chosen to keep drop probability at zero, 2) if the queueing delay is less than 1 ms and the drop probability is not zero, we select $a_{\downarrow\downarrow}$ that reduces the drop probability by 0.1. In addition to these trivial cases, we also allow the model to discover new decisions by giving some probability to choose a random action to be executed. This enables the model to continuously learn and adapt to the changes in the environment. After the pre-training phase, this probability was set to 0.01 in our setting.

B. Pre-training

At the beginning the Agent's internal model is empty, it basically knows nothing about the environment and the optimal policy π. To learn the appropriate policy and its Q-value function, we show example episodes to the Agent during the pre-training phase. The method is the same as in Figure 1 with a single exception: the probability of random action selection is not constant. It starts with 0.9 in the first episode and decays exponentially with a minimum of 0.01. Accordingly, in the ith episode it is set to $\max(0.9^i, 0.01)$. Each pre-training episode covers a 120 sec long simulation with 9 sections with various traffic intensities (see Figure 2). The bottleneck capacity and the link delays are fixed and a dumb bell topology is used.

C. Reward function

The core essence of reinforcement learning methods is the good choice of reward function. The reward is the optimization objective used for finding the optimal policy π. In contrast to traditional methods like PIE AQM or CoDel where a target queueing delay parameter is needed, we have decided to follow the idea of [3]. This paper shows that there is no unified target delay that is optimal in all cases. The good target value also depends on the number of flows in the system. For example, large number of flows can fully utilize the link with smaller target delays. This idea led us to define a reward function depicted in Figure 3 that takes both the utilization and queuing delay into account: $R(U, D) = (U^2 - 0.5) + (\frac{2}{1+D/5} - 1.5)$ where $U \in [0, 1]$ is the link utilization and $D \geq 0$ is the queueing delay in ms. The reward function returns values in the range $(-2, 1]$ and is composed of a utilization- and a delay component. Though it is more sensitive to the decrease in utilization than in delay, it results in a good trade-off between low delay and good link utilization. If the load is low, the utilization component cannot grow above a certain level and thus reward can only be increased with reduction in queuing delay. However, if the the load is high, full utilization may require larger queuing delays and thus delay cannot be reduced below a limit without penalty on the utilization component. One can also observe that for wrong cases (e.g., low utilization and high queueing delay) the reward function assigns a negative value (marked by red in the figure), preventing the increase of the cumulative reward and fastening the learning process.

IV. Evaluation

The network topology used in this work to evaluate the proposed RL-AQM method is shown in Figure 1. It consists of 50 source and 50 sink nodes. They share the same bottleneck between n0 and n1. Link capacities between the sources and n0, and the sinks and n1 are set to be 1 Gbps. The bottleneck link between n0 and n1 is set to 10 Mbps or 100 Mbps, depending on the scenario. The end-to-end RTT between sources and sinks are varied between 20 and 100 ms. In every scenario, the simulation time is 120 seconds that is split into 9 sections with various number of flows (see Figure 2). Each flow represents a TCP connection with NewReno congestion control working in non-ECT mode. If it is not mentioned otherwise, we present the results of pre-trained Agents (with 200 pre-training episodes).

A. Pre-training and convergence

Figure 4 shows the convergence of the average reward (blue curve) and queueing delay (red dashed curve) over the pre-training episodes. An episode is a 120 sec long simulation with various traffic intensities (see Figure 2). Note that the maximum value of the applied reward function is 1.0. Episode 1 starts with an empty internal model. One can see that after the first 10 episodes we reach 0.82 and then the reward stabilizes around this point. The queuing delay is near 50 ms in Episode 1 and it reaches 20 ms by the end of pre-training. Some outlier episodes can also be seen but in these cases the probability of random action selection was still high, resulting in small drop ratio in the QDisc that led to larger queueing delays, implicating the lower reward value.

As mentioned previously, the Agent can also apply an action selected randomly. During the pre-training phase, the probability of random selection is high which enables the agent to trial various actions and learn their effects. The probability of this decision is decreasing in every pre-training episode. Figure 5 shows the same reward (blue) as the previous figure and the probability of random action selection (red). When this probability is high, the less stable agent behaviour may lead to episodes with low reward (e.g., Episode 12 and 20), but it is natural in the pre-training period.

B. Comparison to PIE AQM

In order to consolidate the proposed RL-AQM, we will compare its performance to PIE AQM [14] under the same network topology. We have selected PIE among the others AQMs because PIE is also based on control theory to oversee queue size and modify drop probability accordingly, as in the proposed RL-AQM. It also contains auto-tuning features, allowing its usage under various network conditions without significant re-configuration. PIE uses PI controller to update the drop probability whereas the proposed AQM relies on the DQN-based Agent. Figure 6 shows the total throughput on the bottleneck link (top), the per-flow throughput curves (middle), and the queue delay (bottom) under the action of RL-AQM when the bottleneck link is set to 10 Mbps and RTT is 20 ms. Figure 7 depicts these three graphs for PIE AQM. One can observe that in comparison to PIE, RL-AQM significantly reduces the queue delay and results in the same link utilization. RL-AQM managed to keep the average queue delay almost all the time under the PIE's delay target value (15 ms as default) especially when the number of flows is high (50-60 secs). Even when the number of flows decreases after the 80th second, PIE can not quickly push the delay under the target value because it gradually decreases the drop probability value according to its algorithm. Both AQMs obtain the same throughput value and the fairness of flows are also relatively the same.

Figure 8-9 show the same three metrics after we increased the bottleneck link capacity to 100 Mbps. PIE shows better performance in the average queue delay comparing to its performance under the 10 Mbps but still higher than the PIE's target value. RL-AQM has managed to keep the average queueing delay low in most of the simulation time. It generates congestion signals earlier than PIE, leading to that TCP sources starts reducing their sending rates without waiting the queue to get full. On the other side, the utilization seen for RL-AQM is slightly worst than for PIE, it is the price we have to pay for the low delay. However, one can also observe that large deviations can only be seen when the number of flows are 1 or 2. In other cases, the total throughput is almost 100 Mbps all the time. In contrast to PIE, RL-AQM reward function was designed to find a good trade-off between the utilization and delay, and do not contain any target value. The reward can be increased by decreasing the delay at the expanse of utilization for some level, and vice-versa. Note that this behavior is similar to what a virtual queue does with a slightly smaller serving rate than the one of associated physical queue.

C. Generalization

During the pre-training phase, we fix the parameters of the simulations. Though the number of flows are varied in a wide

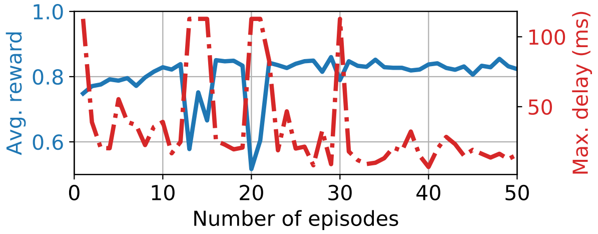

Fig. 4. Rewards and observed maximum delays over the episodes

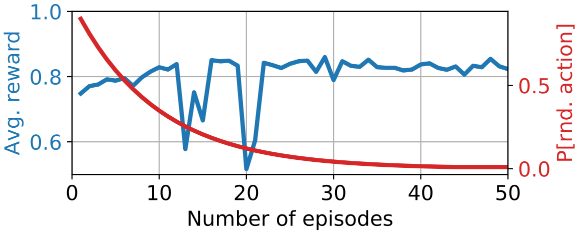

Fig. 5. Rewards and the probability of random action selections over episodes

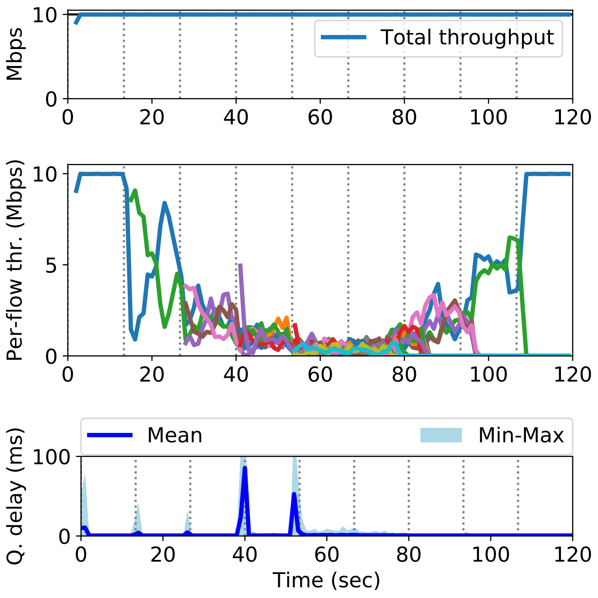

Fig. 6. RL-AQM, BN capacity=10 Mbps RTT=20 ms

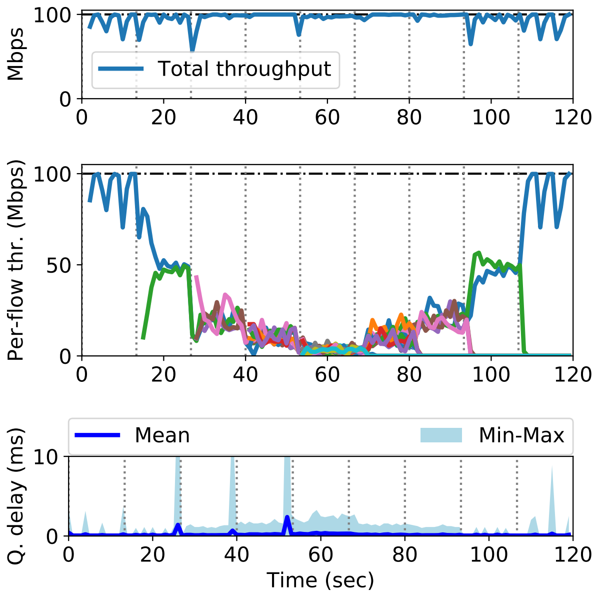

Fig. 8. RL-AQM, BN capacity=100 Mbps, RTT=20 ms

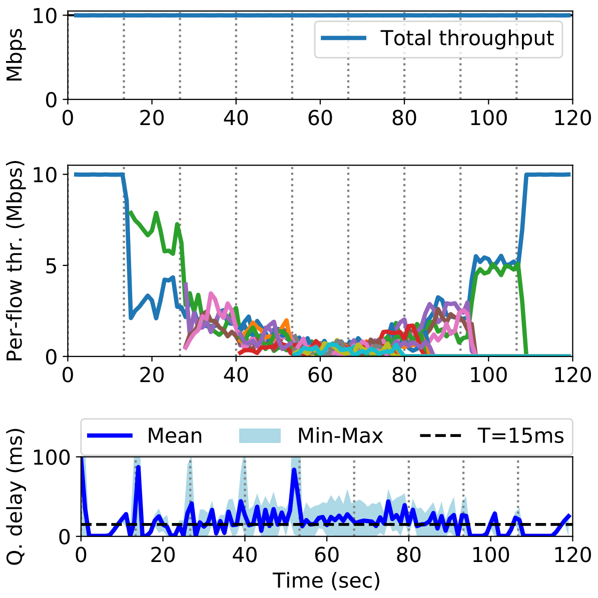

Fig. 7. PIE AQM, BN capacity=10 Mbps, RTT=20 ms

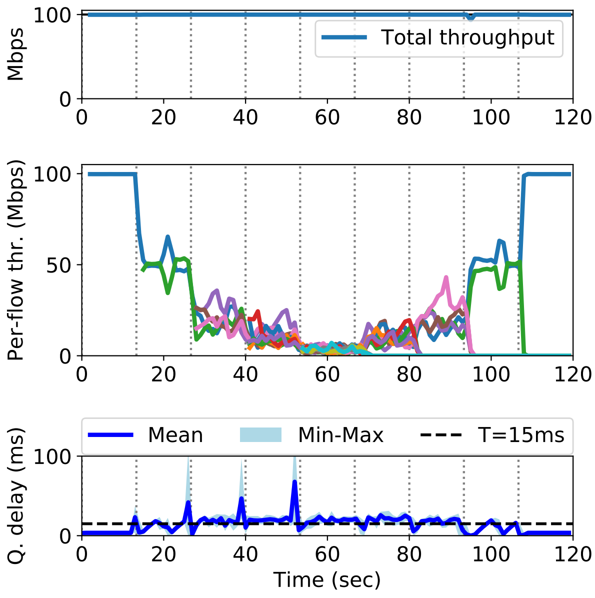

Fig. 9. PIE AQM, BN capacity=100 Mbps, RTT=20 ms

range, the bottleneck capacity and the RTT remain the same over the pre-training episodes. To check the generalization ability of a pre-trained Agent, we have executed simulations with different bottleneck capacity and RTT. We created an Agent trained with the following parameters: bottleneck capacity was 10 Mbps and the RTT was 20 ms. We used this Agent in evaluations with different settings. Figure 10 shows the first case when the bottleneck capacity is 100 Mbps instead of 10 Mbps while the RTT is still 20 ms. One can see that the total throughput of flows reaches the maximum capacity and flows share the bottleneck in a fair way. However, the time needed for reacting to temporal bursts is longer than it is in Figure 8, esp. when the number of flows is 1 or 2. Note that it is also reflected by the higher queueing delays in these simulation sections. The reason is that the model has been trained with lower bandwidth where the decision is based on fewer packets, so the congestion signal (drop probability) generated by the AQM is higher than what is actually required for reducing the delay.

Figure 11 depicts the second case where the bottleneck capacity remains 10 Mbps, but the RTT is 100 ms instead of 20 ms. Despite the feedback loop of TCP's congestion control is much longer, the pre-trained Agent results in almost perfect utilization and good fairness among flows. The large RTT results in longer ramp-up in the sending rates. The longer queueing delays are natural results of the longer RTT, since TCP is RTT-clocked. RL-AQM changes the drop probability, but flows do not react until 100 ms. When the number of flows

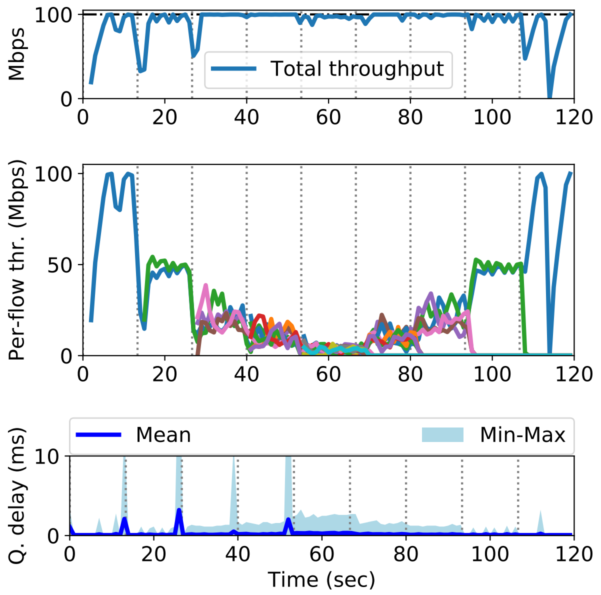

Fig. 10. Generalization ability of RL-AQM, BN cap. increased to 100Mbps

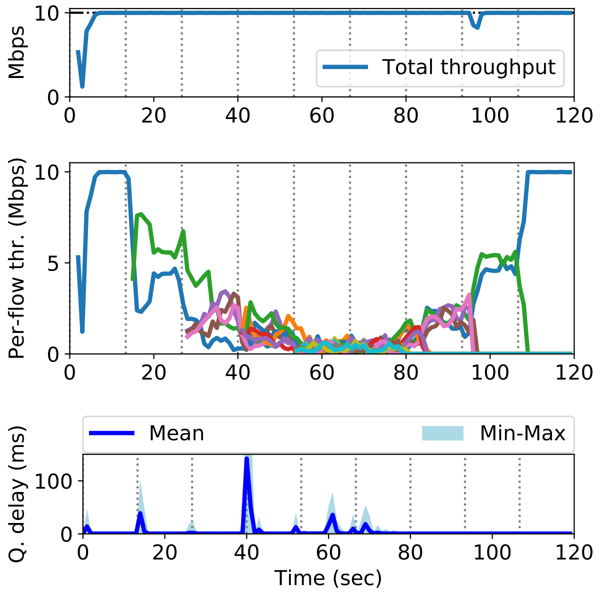

Fig. 11. Generalization ability of RL-AQM, RTT increased to 100ms

is high, high average queueing delays can be observed. It is the same situation: the AQM makes changes, but flows need 100 ms for reaction.

V. Conclusion

This paper proposes RL-AQM, a new AQM method based on reinforcement learning to manage the network resources and keep the queue delay low at the same time. The proposed method has the advantage that it adapts to the changing network environment automatically, without the need of extra parameters and parameter tuning. The queue delay control is done by an Agent implementing a Deep Q-learning model. We implemented the RL-AQM in NS-3 network simulator with the use of OpenAI Gym tools. Our evaluation results demonstrate that RL-AQM provides good performance in different types of networks (different bottleneck link capacities and RTTs). For the purpose of validation, we have compared its performance to the state-of-the-art PIE AQM. We have shown that RL-AQM can keep the queue delay under PIE's target delay during most of the simulation time. The price of ultra-low delay is that some bandwidth is sacrificed, esp. if the number of flows is small, leading to a virtual queue-like behavior.

Acknowledgment

”Application Domain Specific Highly Reliable IT Solutions” project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme. Supported by the ÚNKP-20-4 and the ÚNKP-20-5 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. The research has partially been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

References

[1] D. A. Alwahab *et al.*, “A simulation-based survey of active queue management algorithms,” in *Proc. of the 6th International Conference on Communications and Broadband Networking*, 2018, pp. 71–77.

[2] C. Kulatunga *et al.*, “Tackling bufferbloat in capacity-limited networks,” in *2015 European Conference on Networks and Communications (EuCNC)*. IEEE, 2015, pp. 381–385.

[3] R. Bless *et al.*, “Policy-oriented aqm steering,” in *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 2018, pp. 1–9.

[4] P. Chuprikov *et al.*, “Towards declarative self-adapting buffer management,” *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 3, pp. 30–37, 2020.

[5] P. Bosshart *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[6] P. Gawłowicz *et al.*, “ns-3 meets openai gym: The playground for machine learning in networking research,” in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019, pp. 113–120.

[7] Y.-B. ZHANG *et al.*, “A robust active queue management algorithm based on reinforcement learning [j],” *Journal of Software*, vol. 7, 2004.

[8] Y. Su *et al.*, “Qred: A q-learning-based active queue management scheme,” *Journal of Internet Technology*, vol. 19, no. 4, pp. 1169–1178, 2018.

[9] N. Vucevic *et al.*, “Reinforcement learning for active queue management in mobile all-ip networks,” in *2007 IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*. IEEE, 2007, pp. 1–5.

[10] W. Li *et al.*, “Qtcp: Adaptive congestion control with reinforcement learning,” *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 445–458, 2018.

[11] W. Samek *et al.*, “The convergence of machine learning and communications,” *arXiv preprint arXiv:1708.08299*, 2017.

[12] R. M. Annasamy *et al.*, “Towards better interpretability in deep q-networks,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4561–4569.

[13] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[14] R. Pan *et al.*, “Proportional integral controller enhanced (pie): A lightweight control scheme to address the bufferbloat problem,” in *RFC 8033*. IETF, 2017.