Electronic Communications of the EASST
Volume 080 (2021)

Conference on Networked Systems 2021 (NetSys 2021)

Deep Reinforcement Learning for Smart Queue Management

Hassan Fawaz, Djamal Zeghlache, Pham Tran Anh Quang, Jérémie Leguay and Paolo Medagliani

14 pages

Guest Editors: Andreas Blenk, Mathias Fischer, Stefan Fischer, Horst Hellbrueck, Oliver Hohlfeld, Andreas Kassler, Koojana Kuladinithi, Winfried Lamersdorf, Olaf Landsiedel, Andreas Timm-Giel, Alexey Vinel
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Deep Reinforcement Learning for Smart Queue Management

Hassan Fawaz[1], **Djamal Zeghlache**[1], **Pham Tran Anh Quang**[2], **Jérémie Leguay**[2] **and Paolo Medagliani**[2]

[1] hassan.fawaz, djamal.zeghlache@telecom-sudparis.eu
Télécom SudParis, Institut Polytechnique de Paris, Evry, France

[2] phamt.quang, jeremie.leguay, paolo.medagliani@huawei.com
Huawei Technologies Ltd., Paris Research Center, France

Abstract: With the goal of meeting the stringent throughput and delay requirements of classified network flows, we propose a Deep Q-learning Network (DQN) for optimal weight selection in an active queue management system based on Weighted Fair Queuing (WFQ). Our system schedules flows belonging to different priority classes (Gold, Silver, and Bronze) into separate queues, and learns how and when to dequeue from each queue. The neural network implements deep reinforcement learning tools such as target networks and replay buffers to help learn the best weights depending on the network state. We show, via simulations, that our algorithm converges to an efficient model capable of adapting to the flow demands, producing thus lower delays with respect to traditional WFQ.

Keywords: Queue Management, Smart Queuing, Reinforcement Learning, DQN

1 Introduction

Traditional traffic management and network engineering techniques are struggling to keep up with the ever increasing demand for higher throughput values and lower delays. This necessitates novel approaches to queuing, load balancing, and any other aspect of network management that can sustain faster response times and maintain throughput demands. In the domain of queuing control, Active Queue Management (AQM) has emerged as a smart networking tool to selectively transmit and receive packets. Initial approaches to queuing and dequeuing packets, such as First-in-First-out (FIFO), are passive and have been proven to be unstable [B+94]. AQM techniques use mechanisms such as priority queuing, packet marking, and packet dropping in order to fine tune the network to avoid congestion and minimize end-to-end delays. To this end, a plethora of state-of-the-art approaches exist, including Controlled Delay (CoDel) [NJ12], Random Early Detection (RED) [FJ93], and Proportional Integral controller Enhanced (PIE) [PNP+13] among many others.

In the case of round-robin schedulers (e.g., WFQ, DRR) [BSS12], the latency experienced by a flow depends on its arrival curve (burst size, rate, ...), the arrival curves of the flows it interferes with, and the service rates of queues it traverses. Thanks to network calculus models, worst case estimations can be derived for the end-to-end latency of each flow. However, these models can be complex when deriving tight latency bounds and are facing a trade-off between complexity and tractability for the optimization of queuing parameters.

In our work, we aim to use Reinforcement Learning (RL), a subclass of machine learning, to further improve the efficiency of AQM algorithms. Our goal is to improve decision-making when optimizing Quality of Service (QoS). We design an algorithm, DQN-WFQ, that leverages deep Q-learning to improve decision-making in a smart queuing approach. We classify the majority of network flows into three classes: Gold, Silver, and Bronze. Each of these classes represents a set of throughput and delay requirements, which we aim to meet.

The designed algorithm targets a Weighted Fair Queuing (WFQ)-based scheduler, to which we apply a Deep Q-Learning Network (DQN) to output the optimal weights in order to meet the SLA requirements of each flow. At every set time interval, the DQN agent, placed on bottleneck links, receives an observation of the environment. The latter is made up of the throughput and delay values experienced by each flow class. Following this observation, the agent takes a decision on either increasing, decreasing, or not adjusting the class weights. The objective it has is to maximize the reward that will be issued following its decision depending on the number of flow class requirements it met or violated. At the start of the training process, the agent's choices are random. As the training proceeds, the agent learns how to best react to the environment and output the optimal weights at each step. Finally, in order to improve the efficiency of the agent, we implement deep learning tools such as replay buffers and target networks.

The rest of this paper is structured as follows. Section 2 reviews the related works in the state-of-the-art. Section 3 presents our deep learning approach to smart queuing. It introduces the WFQ approach, the formulation of the problem at hand, and a detailed description of the implemented deep learning mechanism. Section 4 shows our simulation results, and finally Section 5 concludes this paper.

2 Related Works

In what follows, we highlight three main areas where smart queuing and network management take advantage of machine learning to improve network performance: active queue management, traffic engineering and Multi-Path TCP (MPTCP), and service management.

In the context of AQM, authors in [KE19] propose an AQM mechanism to reduce queuing delay in fog and edge networks. It relies on deep RL to handle latency and trades-off queuing delay with throughput values. The proposed algorithm is compared to other AQM approaches, namely P-FIFO, RED, and Codel. Their proposal outperforms the other schedulers in terms of delay and jitter, while maintaining above-average throughput. The state in this work is related to the current queue length in packets, the dequeue rate, and the queuing delay. The action considered is to either drop or serve the packets. Another interesting approach in the state-of-the-art is that of [GGC$^+$18] which proposes, as scheduling schemes, a dynamic-weight-earliest-deadline first (DWEDF) algorithm and a reinforcement learning approach, called DWEDF-RL, to address the scheduling of heavy tailed and low tail flows while accounting for bursty traffic. They provide delay-bound-based fairness to flows having similar tail distributions in an intra-queue buffering process with DWEDF. RL is shown to provide benefits dealing with flows whose behavior is hard to predict or characterize beforehand. The authors in [BFZ20] use a deep neural network to dynamically learn the optimal buffer size per network flow. Their proposal assigns the queue size depending on congestion control, delay and bandwidth. They show that they can reduce queue sizes without any loss in throughput performance. Finally, within the same context, an RL approach for bursty load management is proposed in [BAT$^+$20] to improve average wait times and manage over-saturation in the queues.

For MPTCP traffic control, in [Ros19] a deep Q-network is used for traffic management, wherein a DQN agent selects the optimal paths. This work illustrates the benefits of using a DQN approach to traffic scheduling and path selection problems. The work in [LZDX20] addresses scheduling of flows in MPTCP, with a focus on short and long MPTCP flows, and uses RL to improve performance compared with traditional traffic scheduling methods. The proposed DDPG-based deep reinforcement learning framework determines how to distribute the packets over multiple paths while decreasing the out-of-order queue size under such paths. The RL model is solved via an actor-critic framework and transformer encoders are used to process the states of dynamic sub flows for the neural networks.

Finally, in the domain of service management, [KMRW19] describes a deep reinforcement learning approach addressing the resource allocation problem in network slicing, accounting for both highly dynamic traffic arrival characteristics as well as job resource requirements (e.g., compute, memory and bandwidth resources). The authors demonstrate that their solution improves overall resource utilization and reduces latency in comparison to an equal slicing strategy.

In our work, we seek to assist existing QoS schedulers with a DRL agent and deep reinforcement learning. The proposal in [KE19] is purely an AQM. The proposed DRL algorithm is only concerned with making decisions on either dropping or serving packets. The algorithms in [Ros19] and [LZDX20] use deep learning for optimal path selection in networks, and finally the approach in [BFZ20] is concerned with buffer size optimization. Our objective is to meet stringent demands for network flows belonging to a set of three classes: Gold, Silver, and Bronze, by descending priority. This issue has not been addressed in the state-of-the-art using deep learning tools. We aim to use new tools to address a classic quality of service issue. In what follows, we present a deep Q-learning weighted fair queuing-based approach to managing network flows. We want to determine the optimal scheduling weights per flow class depending on the network and traffic scenarios at hand. Our model adapts to the variation in the traffic flows, always outputting the optimal weights in order to maintain the respective priority of the flows.

3 A Deep Reinforcement Learning Approach to Smart Queuing

In this section, we discuss our deep Q-learning approach for optimal weight selection in a WFQ-based environment. We start by introducing the concept of weighted fair queuing and our implementation of it, and afterwards discuss our deep RL approach.

3.1 Weighted Fair Queuing Based AQM

Weighted Fair Queuing [BLN99] is a state-of-the-art approach to network scheduling which constitutes a packet-based implementation of the generalized processor sharing algorithm [PG93]. In a classic WFQ approach, each flow $i \in N$ would achieve a general data rate equal to:

$$R_i = \frac{w_i}{(w_1 + w_2 + ... + w_N)} R, \tag{1}$$

where R_i is the rate of flow i, R is the total link rate and w_i is the weight associated with flow i. As such, the greater the weight of the flow, the better it is served.

In our implementation, the classic WFQ approach is installed above classic AQM algorithms. We chose WFQ, instead of a strict priority scheduler, in order to ensure fairness. Each flow, based on its priority, is enqueued in a different child queue, separated from the others and following either the same or different AQM approaches: RED, FIFO, CoDeL, PIE, etc. The algorithm would then dequeue packets from each queue in order depending on the associated weight. This means that on top of the packet queuing and dropping mechanisms, as well as the inter-class bandwidth sharing maintained by AQMs, we have QoS guaranteed by the WFQ algorithm.

In our approach, we assume that all flows in the network can, based on packet priority, be classified intro three classes, listed by order of importance: Gold, Silver, and Bronze. Each of these classes has throughput and end-to-end delay targets to meet. Each flow is mapped into a corresponding queue based on its priority (flow type), and is thus associated with a certain QoS.

3.2 Deep Q-Learning Network For Optimal Weights

In this section, we introduce our DQN-WFQ algorithm. We aim to utilize a deep reinforcement learning approach, specifically DQNs, to create a model that is able to input the network state in terms of throughput and delay values and output the weights that would lead to satisfying the largest amount of flow classes' throughput and delay requirements. In what follows, we highlight the main features of this DQN network.

3.2.1 Formulation of the Problem

In order to properly construct our reinforcement learning network, we first need to model the problem as a Markov Decision Process (MDP) [Bel57]. The latter provides a mathematical framework to model decision-making in the learning environment. An MDP is represented by the tuple (S,A,P,r), where S represents the set of states, A is the set of possible actions, $P(s_1|s_0,a_0)$ is the probability of transitioning from a state s_0 to a state s_1 after taking an action $a_0 \in A$, and $r(a_0,s_0)$ is the reward issued for moving from state to another. Our objective is to continuously find the optimal weights for the WFQ algorithm. We use deep Q-learning which is a value-based implementation of the Markov process. It consists of taking actions based on eventual values known as Q-values. In order to use a DQN, we must first define the set of possible states, actions and rewards in our scenario.

1. **Problem States**: We define a state in our problem, at a given time t, as a sextuple of the current throughput and end-to-end delay values achieved by each class of flows: gold, silver, and bronze: $s_t = \{T_g, d_g, T_s, d_s, T_b, d_b\}$, where T_g is the throughput of the gold flows, d_g is the delay of the gold flows *i.e.*, the average delay of all flows classified as gold, and so on. Since in theory this state space is infinite, we normalize and discretize the set of possible states. An increased state space would mean the algorithm has more room to explore for better solutions, but would incur a time penalty for convergence.

2. **Possible Actions**: The DQN agent is tasked with manipulating the weights of the WFQ algorithm. As such the set of actions it can take on the weight of each type of queue

(Gold, Silver, or Bronze) is to either increase it by a preset value ($+x$), decrease it by that same value ($-x$), or leave the weight unchanged (+0). The agent will take an action on the weights of all the flow classes simultaneously. The weights are then normalized to ensure their sum is always equal to one. The action on the gold class, for example, can be expressed as $a_g \in \{-x, 0, +x\}$, and as such the action of the agent at each step can be formulated as: $a_t = (a_g, a_s, a_b)$. Since the agent acts on all three class weights simultaneously, and since there are three possible options to adjust each weight, there exists a total of 27 possible actions.

3. **Action Reward**: The reward is issued for an agent depending on how its action has impacted the throughput and delay thresholds of each flow class. Let η_i be the reward for meeting the throughput demand of flow class i, and ϕ_i the reward for meeting the delay requirement of class i. The total reward is computed as follows:

$$r_t = \omega_g^t \cdot \eta_g + \omega_g^d \cdot \phi_g + \omega_s^t \cdot \eta_s + \omega_s^d \cdot \phi_s + \omega_b^t \cdot \eta_b + \omega_b^d \cdot \phi_b. \tag{2}$$

ω_i^t is a binary value that equal to -1 if the demand throughput for flow class i is not met and +1 if it is. Similarly, ω_i^d is equal to -1 if the delay requirement is not met for flow class i and +1 if it is. This means that the reward can be in the negative *i.e.*, a penalty. Finally, we define the delay reward/penalty in terms of the throughput reward/penalty: ϕ_g = $\kappa \cdot \eta_g$. This enables tuning the reward to put more emphasis on the throughput or vice-versa. In principle, $\eta_g > \eta_s > \eta_b$ indicating that the agent is rewarded better and penalized harder for meeting, or failing to meet, the demands of the gold class than it is for the silver and bronze classes, respectively. These values can be fine-tuned to further improve the significance of one class over the other, but if the value of η_g, for example is too high, the agent will be incentivized to meet the demands of the gold flows while disregarding the others as it attempts to maximize its reward.

3.2.2 Deep Learning Process

Deep Q-learning builds on traditional Q-networks which work as follows. In a Q-network algorithm, the objective of the agent is to output a policy $\Phi(a|s)$ which would dictate how the agent moves from one state to another and which actions to take in order to maximize the long-term reward [WD92]. Once the Q-values are defined, and when in any given state at a certain time t $s_t \in S$, a chosen action to improve the policy can be expressed as:

$$a_t = \underset{\forall a \in A}{\arg\max}\ Q(s_t, a_t). \tag{3}$$

In order to find the optimal policy Φ, the Bellman equation can be used to compute the optimal value function starting from a state s as:

$$V(s_t) = \underset{\forall a_t \in A}{max}\{r(s_t, a_t) + \gamma \cdot \sum_{s_{t+1} \in S} Pr\{s_{t+1}|s_t, a_t\} \cdot V(s_{t+1})\}, \tag{4}$$

where s_{t+1} is the state following s after taking the action a_t and γ is the discount factor $\in [0,1]$. The discount factor quantifies how much weight we put on future rewards. The value in the

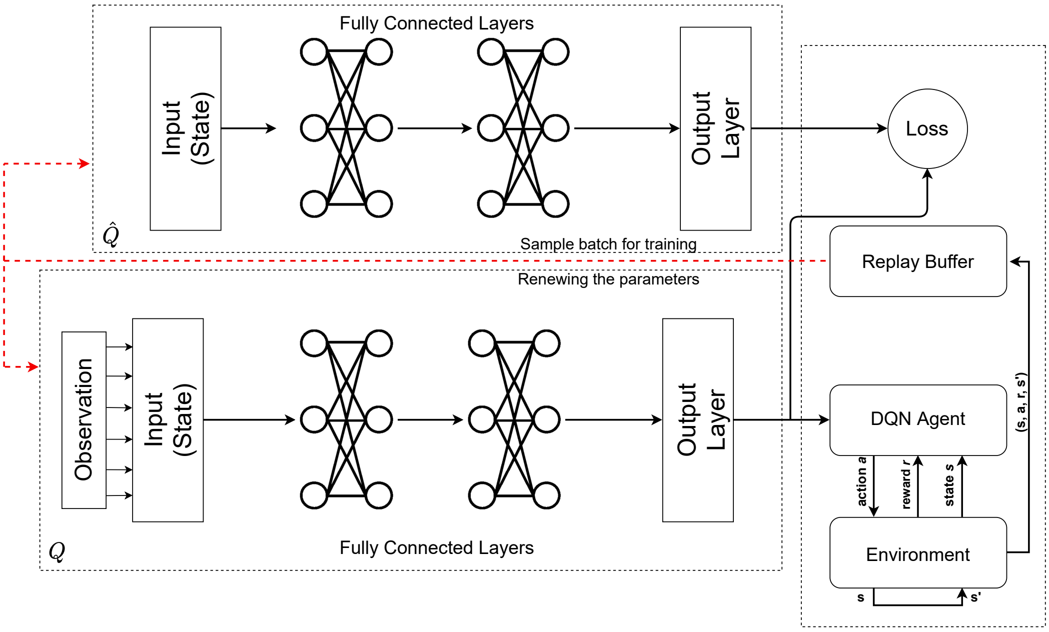


Figure 1: DQN algorithm structure

Q-table is defined as:

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma \cdot \sum_{s_{t+1} \in S} Pr\{s_{t+1}|s_t,a_t\} \cdot \max_{a_{t+1}} Q(s_{t+1},a_{t+1}). \tag{5}$$

As such, the Q-table can be updated following:

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha(r(s_t,a_t) + \gamma \cdot \max_{a_{t+1}} Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)). \tag{6}$$

α is the learning rate $\in$ [0,1], and a_{t+1} is the action taken when in state s_{t+1} based on the policy Φ. The learning rate controls the speed with which the algorithm learns: if too low, the algorithm might never reach its objective, while if too high, the algorithm might oscillate around the objective.

The problem with classical Q-learning is that it requires a table entry for each possible state-action combination. Any simple realistic scenario would explode the amount of required memory to store the Q-table. As such, when using Q-learning with a deep neural network, we change the Q-values completely, at each training step, instead of updating them with the increments. The learning rate α is also dropped as it is now accounted for within the back propagating optimizer as we detail in the following section.

3.2.3 Target Network and Experience Replay

In Q-learning, we are "*updating a guess with a guess*". This becomes even harder when the target value we are chasing *i.e.*, $r(s_t,a_t) + \gamma \cdot \max_{a_{t+1}} Q_t(s_{t+1},a_{t+1})$ is moving. The two consecutive states

Algorithm 1: Training the DQN Agent

```
Initialize the main network Q
Initialize the target network Q̂
Initialize the experience replay memory $D$
Initialize the Agent to interact with the Environment
while not converged do
    /* Sample phase for the replay buffer */
    $\varepsilon \leftarrow$ set new epsilon via $\varepsilon$-decay
    Choose an action $a$ from state $s$ using policy $\varepsilon$-greedy($Q$)
    Agent takes action $a$, gets reward $r$ and next state $s'$
    Store transition ($s$, $a$, $r$, $s'$, $done$) in the replay memory $D$
    /* done indicates if the DQN agent reached its set target
       or not */
    if enough experiences in $D$ then
        /* Learning and training phase */
        Sample a random minibatch of $N$ transitions from $D$
        for Every transition in ($s_i$, $a_i$, $r_i$, $s'_i$, $done_i$) in minibatch do
            if $done_i$ then
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \cdot \max_{a' \in A} \hat{Q}(s'_i, a')$
            Calculate the Loss $\mathscr{L} = \frac{1}{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
            Update Q using the SGD algorithm by minimizing the loss $\mathscr{L}$
            Every $C$ steps, update Q̂ using Q's weights
```

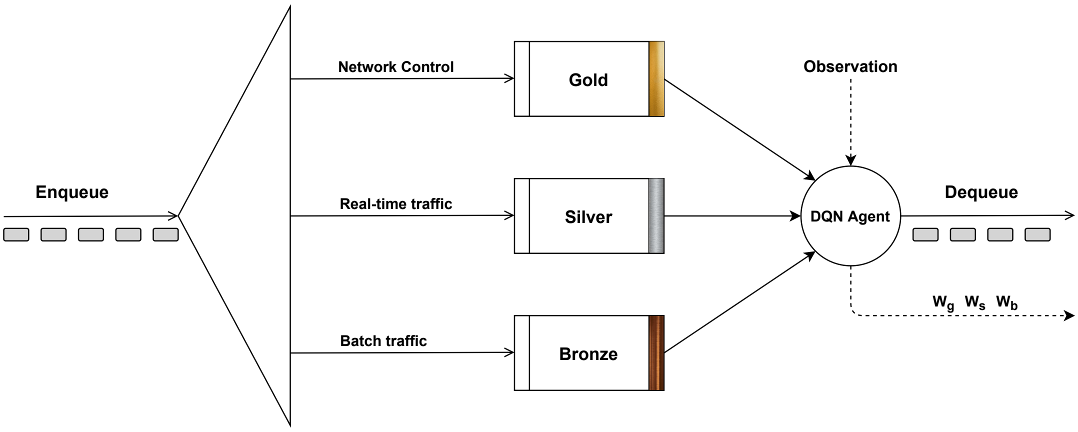


Figure 2: Agent placement and decision making

s_t and s_{t+1} have one step between them, they could be very similar and the neural network would not be able to distinguish between them. When the neural network's parameters are updated to make $Q(s,a)$ closer to a desired value, the value for $Q(s_{t+1},a_{t+1})$, a different entry in the table, could be altered. This could make the training unstable.

In order to avoid this situation, we utilize a target network in addition to the main neural network. This target network is a copy of the main one and we use it to keep the values of (s_{t+1},a_{t+1}) [HS19]. The predicted Q-values of this copy Q-network are then utilized to backpropagate through and train the main Q-network. The parameters of the target network are not trained, but regularly updated with the values of the main network. Note that when moving from classical Q-networks to deep Q-networks, the learning rate is no longer used in the same capacity. It is accounted for within the optimizer used to minimize the loss. We use the default value of 0.001.

In a deep neural network, we can learn through backpropagating and by using stochastic gradient descent (SGD) optimization. A fundamental requirement to do so is that the data we are training the model on is independent and identically distributed. If the agent is being trained on the data as it comes, the sequence of experience could be highly correlated. This can cause action values to oscillate and diverge. As a solution to this problem, we use a large buffer to store past experiences. These past experiences are tuples of the form (S,A,R,S') *i.e.*, they have the current state, the action taken, the reward issued, and the subsequent state. This experience replay buffer is then randomly sampled to train the model. This process of sampling batches from the replay buffer to learn is known as experience replay [ZS17]. The agent learns by minimizing the loss. The structure of the DQN algorithm can be seen in Figure 1. A pseudo-code of the process is shown in Algorithm 1. In the referenced code, simplified notations are used with s referencing the current state, s' referring the next state following the taken action a, and r being the issued reward. Our model is trained on a certain network topology as the traffic varies. As long as there is not a major change in the topology, there would be no need to retrain the model. Finally, in Figure 2 we illustrate how the DQN agent is pushed on top of the bottleneck queues where it monitors the environment and calibrates the weights of each class accordingly.

4 Simulation Results

In this section, we benchmark the performance of the proposed DQN-WFQ based smart queuing algorithm by using the packet-level simulator NS3 [RH10]. The considered topology, illustrated in Figure 3a, is composed by only one bottleneck link, in order to study the impact of the queue management proposal. This topology well represents some realistic scenarios such as a satellite link communication or the link between an access router and a leased line. We consider six origin-destination (OD) pairs that can send data in only one out of three classes of services, i.e., Gold, Silver, and Bronze. Each two OD pairs belong to one traffic class. To mimic diurnal time-varying traffic patterns, the origins generate traffic with a sinusoidal rate between 0.25 and 1.5 Mbps. The bottleneck link bandwidth is 4 Mbps and its propagation delay is 20 ms. In these results we consider only small transmission rates to speed up simulation duration. However we verified that the results remain the same even with larger link bandwidth and transmission data rates. Our trained agent is place on the bottleneck link. The DQN agent is composed of an input layer with six inputs representing the observation, and an output layer with a softmax activation function with 27 outputs, representing the possible action space (3x3x3). For each class (Gold, Silver, Bronze), the agent is taking one out of the three possible actions, *i.e.*, increase the weight, decrease it, or keep it the same. In between the input and output layers, there are two fully connected dense layers with 128 units each. The rest of the parameters are detailed in Tables 1 and 2. We consider two different types of traffic, UDP and TCP. In order to avoid having UDP traffic starving out its TCP counterpart, we run the simulations assuming that all the nodes send the same type of traffic.

Table 1: Scenario Simulation Parameters

Parameter	**Value**
Number of OD Pairs	6
SQ implementation	Bottleneck on top of RED
AQM on other links	FIFO
Simulation duration / snapshot	10 sec
Transport protocol	UDP-TCP
Bottleneck link delay	10 ms
Bottleneck link BW	4 Mbps
Traffic generation per node	About 0.25 to 1.5 Mbps
Weights WFQ UDP	0.5/0.33/0.16 & 0.6/0.25/0.15
Weights WFQ TCP	0.58/0.26/0.16 & 0.5/0.33/0.16

Table 2: DQN Simulation Parameters

Parameter	**Value**
Activation function connected layers	RELU
Activation function output layer	Softmax
N^o of fully connected layers	2 each with 128 units
Training batch size	32
ε - decay	x 0.99955 per episode
Discount factor γ	0.99
N^o of terminal steps to update target	5
Reward relative to flows G/S/B	$3x/2x/x$
Delay to throughput relevance κ	0.8
N^o of training episodes	15,000

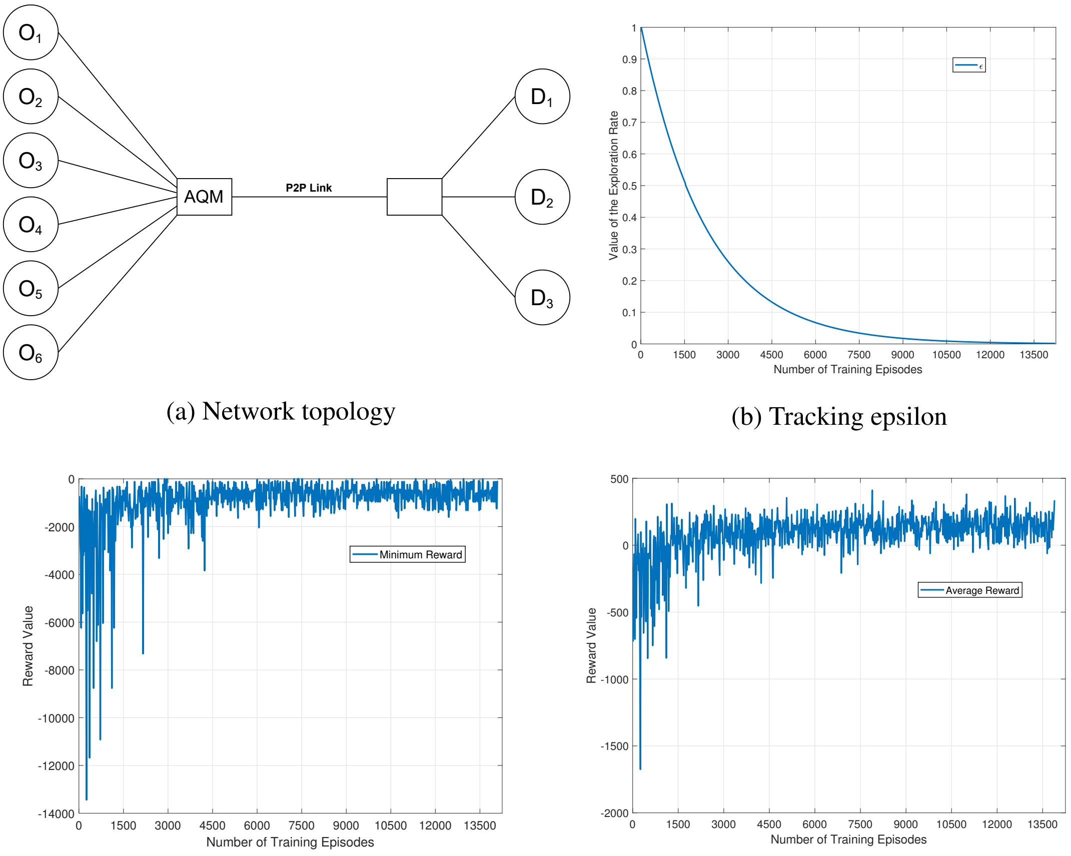
4.1 Algorithm Convergence

We first illustrate how the exploration rate varies with each training iteration. ε starts at 1 (i.e., always explore a new solution) and is decayed by a factor of 0.99955 each time the model is trained, as shown in Figure 3b. After 1500 episodes, it is a coin flip whether the algorithm explores or exploits. The minimum value for ε is set at 0.001 or a 1% chance that the algorithm decides to explore. This method of decaying ε ensures that the algorithm has enough iterations to explore the entire state space. If the total number of training iterations is to be reduced, the decaying could be accelerated by decreasing the value of the decay rate. As illustrated subsequently, the algorithm does not need as many iterations to converge (1500 iterations are enough).

We verify that the algorithm is converging by looking at the cumulative rewards for sets of 15 episodes. We do this to smoothen the curve and root out any undesirable fluctuation. In Figure 3 we track both the minimum and average rewards for these aggregated episodes. Note that the episode's reward is its total accumulated rewards until it reaches its goal (*i.e.*, $done = 1$). We set the goal to be equal to zero for this simulation. This means the DQN agent is trained whenever the reward is positive. In Figure 3c, we notice that the minimum reward fluctuates severely over the first 500 episodes. However it quickly starts to stabilize at around the value of 0. Similarly, the average reward shown in Figure 3d, starts to stabilize at around the mark of 1500 episodes afterwards reaching a positive reward almost every iteration.

4.2 Case of UDP Traffic

In order to highlight the advantages of using a DQN agent to choose the weights, we simulate our DQN-WFQ algorithm and compare it against the traditional WFQ. The requested QoS for



(a) Network topology

(b) Tracking epsilon

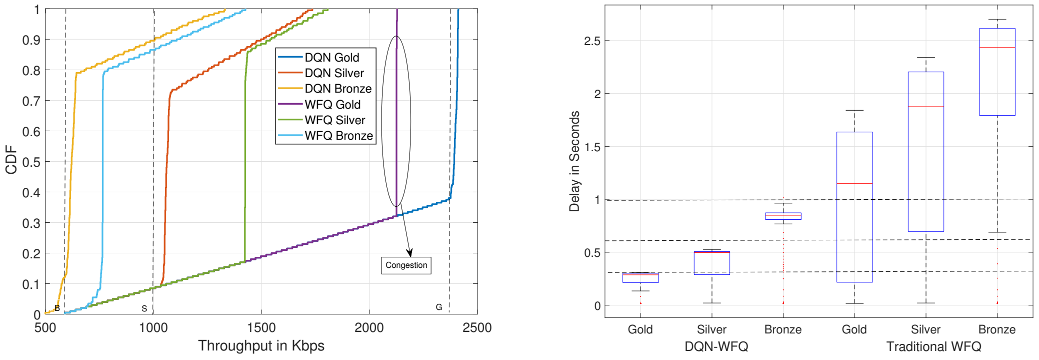
(c) Tracking the minimum reward

(d) Tracking the average reward

Figure 3: Minimum and average reward as a function of the number of episodes

the classes of flows are: 2.4 Mbps for gold flows, 1 Mbps for silver, and 0.6 Mbps for bronze. There are two ways to input the weights for the traditional WFQ approach. The first consists of choosing the weights based on the rewards which we used to train the DQN (section 3.2.1), and the second following the demanded levels of services we input to the DQN as requested. Note that in our simulation the traffic generated per flow does not depend on the class it belongs to. As such, there is no intuitive way to vary the weights on-the-fly for the traditional WFQ approach. We assume that in the case of congestion on the bottleneck link, a minimum level of service must be guaranteed to each flow. For the delay, the QoS requirement is to guarantee an average delay bounds of 0.3, 0.6 and 1 second for gold, silver, and bronze flows, respectively.

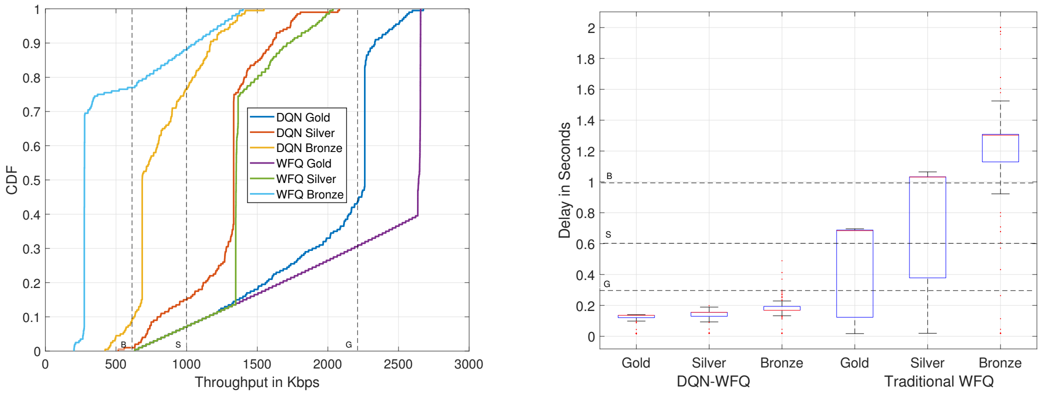
We first look at the throughput result when the traffic sources generate UDP traffic. Figure 4a shows the throughput results for DQN-WFQ compared to traditional WFQ. The weights here for the traditional WFQ algorithm are chosen based on the rewards used to feed the DQN *i.e.,* gold weight is 2x the silver and 3x the bronze (Table 1). The figure has cumulative distribution plots (CDF) for all types of flows for both algorithms. The dotted lines show the demanded values to maintain during congestion periods. The areas where the plot is a straight vertical



(a) Impact on throughput

(b) Impact on delay

Figure 4: Performance in case of UDP Traffic. The dashed lines correspond to the required QoS.



(a) Impact on throughput

(b) Impact on delay

Figure 5: Performance in case of TCP Traffic. The dashed lines correspond to the required QoS.

line indicate that congestion has occurred on the bottleneck. At small data rates *i.e.*, around 0.25 Mbps requested by each node, the bottleneck link is not congested, so it can transmit all the requested traffic. By looking at the congestion areas on the plot, we notice that WFQ meets the throughput levels requested for both silver and bronze flows while it fails to meet the requirement for gold flows (2.12 Mbps instead of 2.4). DQN-WFQ, instead, meets the QoS requirements for all the flows, providing a throughput of 2.405 Mbps for the gold class flows, 1.06 Mbps for the silver, and 0.615 Mbps for the bronze. The DQN agent continuously learns how to update weights in order to maximize the reward, which is contingent on meeting all the flow throughput requirements.

In Figure 4b, we present the results in terms of the average delay experienced by the flow packets. We note that our algorithm can meet all the set delay values while traditional WFQ, notorious for causing delay penalties, cannot guarantee any. Furthermore, setting the weights for traditional WFQ following the delay requirements does not yield any difference in performance as the weights effectively remain unchanged from the current case.

Finally, if we run the traditional approach with the weights set proportional to demanded throughput levels, the traditional WFQ algorithm would be able to meet the gold and bronze flow requirements while failing to meet the silver class demands (around 0.96 Mbps). No significant changes can be noted in terms of delay targets.

4.3 Case of TCP Traffic

We now study the performance of our model in case of TCP traffic, a protocol that differs from UDP because of its innate congestion control mechanism. We lower the throughput requirement for the gold class flows to 2.2 Mbps to count for the bandwidth taken by TCP ACKs. For WFQ, we calculate the weights based on the requirements: for instance, as for the gold flows, we require to guarantee 2.2 Mbps of throughput during congestion, its weight is calculated as $2.2/3.8 \simeq 0.58$. The same approach is taken for the weights of the silver and bronze classes. Similar to before, there is no benefit in varying this weight as the different classes are generating traffic with the same data rates. The CDF distribution of the achieved throughput is shown in Figure 5a. As with the case of UDP traffic, DQN-WFQ is capable of meeting all the throughput requirements. For traditional WFQ, the gold and silver QoS requirements are met, but the bronze flow is starved and served at about 0.2 Mbps during congestion, well below the requirement. This gold class outperforms and has throughput values higher than required. This effect is due to the congestion control mechanisms of TCP. Because of the relatively high weights of the gold and silver flows, not a few packets are being sent or acknowledgments received for the bronze flow. This triggers the congestion control for the bronze flow, further lowering its capability of meeting the set requirements. The bandwidth is then dominated by the gold flows which get about 65% of the bandwidth instead of the 58% they were supposedly allotted by WFQ.

In Figure 5b we highlight the performance of DQN-WFQ in terms of delay. As in the case with UDP traffic, traditional WFQ fails to meet the delay requirements, while DQN-WFQ achieves delay values well below the delay thresholds for all three flow classes, as the continuous weight adaptation carried out by DQN-WFQ helps optimize bandwidth utilization. Finally, we note that calculating the weights for traditional WFQ using the rewards fed to the DQN, or relative to the delay requirements, results in no difference in which flow requirements are being met.

5 Conclusion

In this article, we proposed a deep reinforcement learning approach to help meet stringent demands of classified network flows. We implemented a DQN-WFQ agent that learns the optimal weights for dequeuing different classes of network flows: Gold, Silver, and Bronze. The agent implements DRL tools such as replay buffers and target networks to help in convergence. We showed via simulation results that our proposal is efficient and helps fulfill flow requirements in terms of throughput and delay. In our concurrent and future works, we will implement a multi-agent reinforcement learning approach in a network with multiple bottlenecks.

Bibliography

[B+94] M. Bramson et al. Instability of FIFO queueing networks with quick service times. *The Annals of Applied Probability* 4(3):693–718, 1994.

[BAT+20] V. Balasubramanian, M. Aloqaily, O. Tunde-Onadele, Z. Yang, M. Reisslein. Reinforcing Cloud Environments via Index Policy for Bursty Workloads. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. Pp. 1–7. 2020.

[Bel57] R. Bellman. A Markovian decision process. *Journal of mathematics and mechanics* 6(5):679–684, 1957.

[BFZ20] M. Bachl, J. Fabini, T. Zseby. LFQ: Online Learning of Per-flow Queuing Policies using Deep Reinforcement Learning. In *2020 IEEE 45th Conference on Local Computer Networks (LCN)*. Pp. 417–420. 2020.
doi:10.1109/LCN48667.2020.9314771

[BLN99] V. Bharghavan, S. Lu, T. Nandagopal. Fair queuing in wireless networks: issues and approaches. *IEEE Personal Communications* 6(1):44–53, 1999.

[BSS12] M. Boyer, G. Stea, W. M. Sofack. Deficit Round Robin with network calculus. In *6th International ICST Conference on Performance Evaluation Methodologies and Tools*. Pp. 138–147. 2012.

[FJ93] S. Floyd, V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking* 1(4):397–413, 1993.

[GGC+18] M. Guo, Q. Guan, W. Chen, F. Ji, Z. Peng. Delay-Optimal Scheduling for Heavy-Tailed and Light-Tailed Flows via Reinforcement Learning. In *2018 IEEE International Conference on Communication Systems (ICCS)*. Pp. 292–296. 2018.

[HS19] J. F. Hernandez-Garcia, R. S. Sutton. Understanding multi-step deep reinforcement learning: a systematic study of the DQN target. *arXiv preprint arXiv:1901.07510*, 2019.

[KE19] M. Kim, B. Eng. Deep reinforcement learning based active queue management for iot networks. *PhD Thesis*, 2019.

[KMRW19] J. Koo, V. B. Mendiratta, M. R. Rahman, A. Walid. Deep reinforcement learning for network slicing with heterogeneous resource requirements and time varying traffic dynamics. In *2019 15th International Conference on Network and Service Management (CNSM)*. Pp. 1–5. 2019.

[LZDX20] B. Liao, G. Zhang, Z. Diao, G. Xie. Precise and Adaptable: Leveraging Deep Reinforcement Learning for GAP-based Multipath Scheduler. In *2020 IFIP Networking Conference (Networking)*. Pp. 154–162. 2020.

[NJ12] K. Nichols, V. Jacobson. Controlling queue delay. *Communications of the ACM* 55(7):42–50, 2012.

[PG93] A. K. Parekh, R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking* 1(3):344–357, 1993.

[PNP$^+$13] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th international conference on high performance switching and routing (HPSR)*. Pp. 148–155. 2013.

[RH10] G. F. Riley, T. R. Henderson. The ns-3 network simulator. In *Modeling and tools for network simulation*. Pp. 15–34. Springer, 2010.

[Ros19] M. M. Roselló. Multi-path Scheduling with Deep Reinforcement Learning. In *2019 European Conference on Networks and Communications (EuCNC)*. Pp. 400–405. 2019.

[WD92] C. J. Watkins, P. Dayan. Q-learning. *Machine learning* 8(3-4):279–292, 1992.

[ZS17] S. Zhang, R. S. Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.