Towards Adaptive Packet Scheduler with Deep-Q Reinforcement Learning

Qiwei Wang*, Thinh Nguyen†, Bella Bose‡,
*School of Electrical Engineering and Computer Science, Oregon State University, wangqi@oregonstate.edu
†School of Electrical Engineering and Computer Science, Oregon State University, thinhq@eecs.oregonstate.edu
‡School of Electrical Engineering and Computer Science, Oregon State University, bose@eecs.oregonstate.edu

***Abstract*—The traditional way to design a packet scheduler is usually based on the prior knowledge of networking environments. With advanced networking technologies, we propose a Deep-Q (DQ) learning framework for packet scheduler to take advantage of more available information. The packet scheduler optimizes the application-specific quality of service (QoS) requirements, and adapt to the changing network environment. The DQ framework integrates the online Q-learning algorithm and a deep neural network, making it applicable to problems of large size. Without any prior training or network traffic models, the DQ-based scheduler progressively learns a good policy in real-time, based directly on the available observations.**

I. Introduction

Communication and networking technologies have advanced tremendously over the past several decades. Notably, the original "end-to-end" argument [1] for designing "dumb" and "fast" physical and link layer devices (e.g., routers and switches) are now less applicable with the declining cost of silicon. In fact, the trend in recent years has been to design fast and sophisticated hardware to efficiently support various networking abstractions for many emerging networking technologies such as network virtualization and Software Defined Network (SDN) [2], which provides many benefits such as network isolation, flexible routing, workload orchestration, or application-specific Quality of Service (QoS). Those software, hardware and the information across various OSI layers must be optimized jointly. In this paper, we study a packet scheduler, an important component of the virtualization technologies, for determining which packets should be sent at which time slots to achieve QoS requirements for various applications under resource constraints. Specifically, we propose the Deep-Q (DQ) framework for implementing adaptive packet schedulers. The DQ framework integrates a deep neural network with online Q-learning algorithms. It enables a DQ-based packet scheduler to learn a good packet transmission policy in real time , and can be deployed without any prior knowledge of network traffic models. We posit that with the declining cost of silicon, it is feasible and beneficial to implement a sophisticated DQ-based packet scheduler that learns to improve itself based on the massive amount of data it observes over time, rather than using some schemes with fixed designed parameters that are often sub-optimal under different network conditions. It is shown in the simulation results that the presented DQ algorithm has the ability to adapt quickly when environmental parameters change.

II. Related Work/Background

There is rich literature on resource and scheduling algorithms. In particular, the packet scheduler problem can be formulated as a Markov Decision Process (MDP) problem [3]. An MDP problem is well studied as a stochastic dynamic programming problem with many algorithmic solutions such as Backward Induction , Value Iteration or Policy Iteration. MDP solutions are purely model-based. That is, given a state s and action a, the precise model of the transition probability $T(s, a, s')$ and the instant reward $r(s, a)$ must be provided to the algorithm. This approach is less useful in the real world scenarios, where models cannot be accurately determined. A more popular approach is the online Reinforcement Learning (RL). One particular algorithm is the Q-learning algorithm [4], to be described shortly. Using the Q-learning algorithm, an RL agent learns an optimal state-action pair by interacting with the real environment without any modeling knowledge of the environment. At each time step, the learning agent examines the current state s and takes an action a that maximizes a specified value $Q(s, a)$ related to the total reward. With this mechanism, it is easy to implement an online learning algorithm that gradually improves the agent's performance. As a result, RL has been applied to network/communication control optimization. For example, the routing scheme in [5] employed multi-agent RL to improve delivery time and avoid congestion. And in [6], the scheduling-admission problem of time-varying channels is formulated as a constrained MDP. Then, an improved online learning algorithm was shown to outperform traditional Q-learning.

While RL has been widely applied, many challenges need to be further studied to make the RL algorithms useful for real-world problems with a large state space and a fast-changing environment. In particular, the combination of RL and deep learning utilizes the neural network as a non-linear function approximator, enabling the learning agent to deal with complicated Q functions and enormous size of state spaces. This combination is applied to some of the most current research on resource scheduling. [7][8] purposed RL-based packet schedulers for multi-path TCP (MPTCP) to achieve higher overall network throughputs. [9] presented an RL-based flexible Radio Resource Management (RRM) scheme for 5G access network. It selects one of multiple frequency blocks at each time slot to satisfy each users requirement of delay and packet delivery rate. Compared to existing literature, our research is focused on a more generalized framework such that both states and reward functions can be adjusted for multiple objectives. Furthermore, with a specific queueing example, we discussed a few details of the DQ framework to increase its performance, including the choice of optimizer, experience replay and the delayed update of target neural network.

III. Problem Definition

A. Problem Scenario

For clarity, we describe a LAN scenario where the packet scheduler is implemented at the Access Point (AP). We assume several users running a total of N applications with different QoS requirements such as data rate, packet loss rate, and delay.

The links between each user and the AP are characterized by different channel conditions. Due to the limitation of processing power and channel capacity available, the AP can only serve a limited number of users/applications in a fixed amount of time. The goal of the AP is to dynamically allocate its resources to satisfy all users' QoS requirements. Since the network conditions can change quickly, the AP's resource allocation policy should adapt to the new network conditions timely. Fig. 1 shows the components involving the AP's operations.

1) *External environment:* The environment is modeled as a black box to the AP. The parameters of the environment can be the channel quality, data rates, movement speed and direction of each user, and the like. The AP can only infer the external environment by interacting with it and observing the outputs, or in RL terminology, the immediate rewards.
2) *Observable states:* The observable state is the internal states of the AP that can be observed and used to infer about the environment and make a good decision. States can be pending packets to be transmitted, current packet loss count, and the like.
3) *Resources:* Examples of resources are computational power or total bandwidth.
4) *QoS requirements:* Examples of QoS requirements are minimum bandwidth or maximum delay requested by an application. The AP needs to find the policy that satisfies those requirements.
5) *Policy:* A policy is a mapping between the observable states and an action, such as sending a packet from a particular application given the current backlogs of all other applications.

By observing the states and actions, together with the corresponding rewards over time, the goal of the smart AP is to learn a policy that optimally allocates its resources while satisfying the users' QoS requirements.

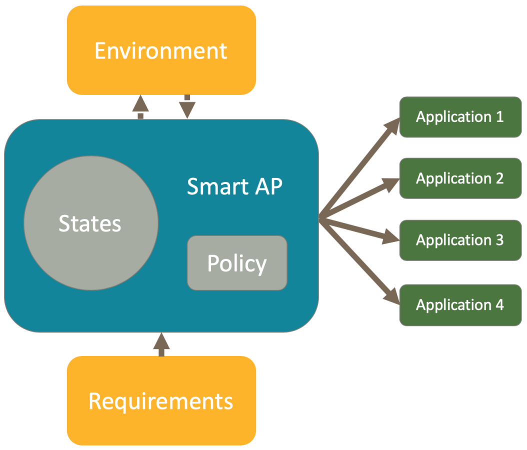

Figure 1: Problem framework

We now focus on the particular problem of designing a packet scheduler to provide QoS for different applications. A packet scheduler uses multiple queues for different applications as shown in Fig. 2. The en-queue and de-queue rates determine the data rates, delay, and packet loss rates of the applications. Assuming that each user/application is associated with a buffer of length $L_i, i = 0, 1, ..., N-1$. Time is discretized into small time slots with the length of T_0. To simplify the analysis, in a single time slot, we assume that the AP will take one action to transmit first, then a new packet arrives.

Departure of a packet: At the beginning of each time slot, the AP observes the current network state and decides which packet from the queues (applications) to serve, i.e., selects a packet from a queue and transmits. Depend on the QoS requirements, the AP can also do nothing if necessary (e.g., to save energy). The channel associated with each application is modeled with the packet delivery rate (PDR), denoted as $q_i, i = 0, 1, ..., N-1$. If a transmission attempt fails, the transmitted packet has to be put back in the queue for re-transmission.

Arrival of a packet: At each time slot, a packet might arrive, and it will be added to the end of the appropriate queue. The probability of a packet arrival for each application is denoted as $p_i, i = 0, 1, ..., N-1$. If the time slot length T_0 is small enough, the arrival model is approximately Poisson, and p_i can be found by the average throughput of an application. After a packet arrives, if its destination queue/buffer is full, the packet is dropped. The packet scheduler is trained to minimize a certain objective, such as the probability of a packet drop due to full queue as a result of channel conditions. In this paper, the objective of a smart packet scheduler is to find a policy that minimizes the packet loss rates. We note that the AP can only observe the current backlog length of each queue (applications) at the beginning of a time slot. The environmental parameters p_i and q_i are not available.

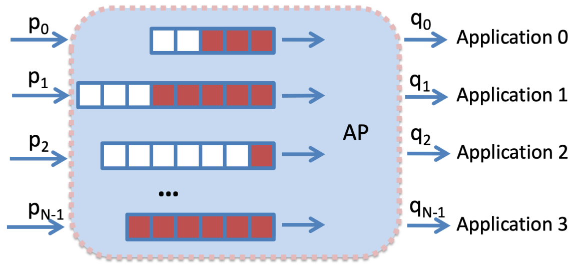

Figure 2: Application flows are modeled as queues

B. Markov Decision Problem

Now we model our problem as an MDP problem:
States: At a time slot t, the observable state is an $N-1$ tuple, denoting the backlog length of each application.

$$s^t = (l_0^t, l_1^t, ..., l_{N-1}^t), \tag{1}$$

where l_i^t is the backlog length of application i at time slot t.

Actions: For simplicity, we assume only deterministic policy. That is, at each time slot, the agent either send one packet for one of the applications or not to send at all. The total number of possible action is $N+1$. a^t, the action at time t, is determined by the policy. After an action is taken, the agent interacts with the environment and observes the state of the next time slot, s^{t+1}.

Instant Rewards: Given s^t, a^t, s^{t+1}, the AP will obtain the an instant reward R^t, which is the sum of instant rewards from all applications:

$$R^t = \sum_{i=0}^{N-1} r_i(s^t, a^t, s^{t+1}). \tag{2}$$

For each application, its QoS requirements can be modeled by choosing an appropriate $r_i(\cdot)$. A typical $r_i(\cdot)$ can be a function of current backlog length, accumulated lost packets, and so on. For this paper, as an example, we consider the packet loss due to a full backlog buffer, so a negative constant C_i is assigned to each application as a penalty for packet loss, that is:

$$r_i(s,a,s') = \begin{cases} C_i, & \text{if a packet is lost.} \\ 0, & \text{elsewise.} \end{cases}$$

Remark 1: By changing C_i for each application, the controller is able to distribute the network resources unevenly to some of the applications. Thus, the priority for each application can be designed by the assignment of C_i. A larger C_i means smaller penalty, indicating the corresponding application has a higher priority when the network is congested. The effect of different values of C_i is shown in Section V.

Optimal policy: The optimal policy is a policy that maximizes the estimated discounted reward, that is, a mapping $\pi^*(s) \rightarrow a$ such that the total discount reward:

$$R_{total} = E[\sum_t \beta^t R_t], \tag{3}$$

is maximized. β is a discount parameter between 0 and 1. Since the instant reward is the penalty of packet loss, maximizing R_{total} will minimize the expectation of packet loss.

Given the environment model, this problem can be solved exactly by many well-known algorithms such as the value iteration (VI). However, since the environment changes dynamically, and the AP does not know its parameters, we will use model-free online learning algorithms.

IV. Approaches

A. Q-learning with function approximation

Given a policy π, to estimate the goodness of a state-action pair, define the value $Q^\pi(s,a)$ as the discounted reward starting at state s and taking action a Mathematically, :

$$Q^\pi(s,a) = R_{total}|(\pi,s,a), \tag{4}$$

The optimal Q value satisfies:

$$Q^*(s,a) = E_{s'}[r(s,a) + max_{a'}Q(s',a')], \tag{5}$$

in which s' is the possible next state if the agent takes action a in state s. The optimal policy will be:

$$\pi(s) = argmax_a Q^*(s,a). \tag{6}$$

Given an initial value of $Q(s,a)$ for all s,a pairs, when a transition $(s,a,s',r(s,a))$ is observed, a model-free online learning agent uses the temporal differential to update the Q value as:

$$Q(s,a) = Q(s,a) + \alpha(r(s,a) + max_{a'}Q(s',a') - Q(s,a)). \tag{7}$$

To find the accurate values $Q(s,a)$ for all (s,a) pairs, each state and action should be sufficiently explored by the agent. To accomplish this, a following exploration/exploit policy is often used:

$$a = \begin{cases} argmax_{a\in A}Q(s,a), \text{with probability } \epsilon, \\ \text{Random action with probability } 1-\epsilon. \end{cases} \tag{8}$$

In this specific problem setting, the state and action space can get very large. For example, 10 users with a maximum queue length of 20 will result in 20^{10} states. With limited computation resources at the AP, it is not realistic to store a huge table for all (s,a) pairs and to visit all (s,a) pairs. Instead, a practical approach is to use function approximation which extracts a feature vector $\phi(s,a)$ from s, and the $Q(s,a)$ table can be approximated by a function $F(\cdot)$:

$$\hat{Q}(s,a) = F(\phi(s,a)). \tag{9}$$

If $F(\cdot)$ is convex and can be parameterized by a vector θ, the optimal $\hat{Q}(s,a)$ can be found by minimizing the square error loss function:

$$||\hat{Q}(s,a) - Q(s,a)||_2, \tag{10}$$

using a gradient method over θ. Since the true value of $Q(s,a)$ in Eq.10 is not immediately available during the online training, we use a sampled version to replace $Q(s,a)$:

$$Q_{sample}(s,a) = r(s,a) + \beta max_{a'}\hat{Q}(s',a'). \tag{11}$$

Thus, the stochastic gradient update rule for θ will be:

$$\theta \leftarrow \theta + \alpha(r(s,a) + max_{a'}\hat{Q}(s',a'|\theta) - \hat{Q}(s,a|\theta))\nabla_\theta \hat{Q}(s,a|\theta). \tag{12}$$

B. Non-linear function approximation with neural network (NN)

In this section, we describe a non-linear function approximator based on the DQN framework introduced in [10]. Due to the large number of states, we use a neural network with multiple hidden layers to approximate the $Q(s,a)$.

1) Features and NN model: We model this problem as a regression problem. Instead of extracting the feature from (s,a) pair, only the state is used to generate the input feature. The output layer of the NN contains $N+1$ neurons and each of them is associated with an action, as shown in Fig. 3. By doing this we take advantage of the smaller action space size (number of applications, usually way less than the number of states), so we can obtain the $\hat{Q}(s,a)$ value for all actions and find the best action in just one pass of forward propagation. The input feature vector is the vector presentation of the

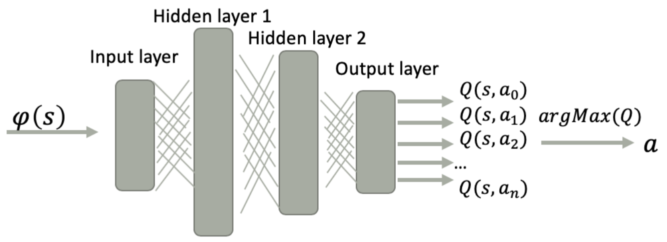

Figure 3: An example of the NN model

backlog lengths of all applications at the beginning of each epoch. The backlog lengths are normalized over the total queue size such that $0 \leq \phi_i(s) \leq 1$:

$$\phi(s) = [l_0/L_0, l_1/L_1, ..., l_{N-1}/L_{N-1}]. \tag{13}$$

2) Experienced replay: For most of the time, the action is taken by choosing a corresponding to the maximum output of the NN. As a result, a slight change of NN parameters may cause a totally different policy and alters the learning trajectory. As a result, updating the NN parameters using only one sampled $Q(s,a)$ is very risky and may result in very unstable performance.

To counter this effect, we employ experienced replay [10][11]. Experience replay uses a memory pool to memorize the newest M transitions. At the beginning of each time slot, the agent takes actions based on the output of the NN, and the transition (s, a, s', r) is recorded in the memory pool. The oldest transition is deleted if the pool is full. Then, rather than using just one sampled $Q(s, a)$, a mini-batch is randomly chosen from the pool to perform gradient descent update of the NN parameters θ.

3) Loss function: As in Eq. 11, $Q_{sample}(s, a)$ is used to estimate the true value of $Q(s, a)$. Then, stochastic gradient descent method is used to carry out one step of update to minimize the mean square error:

$$f_c = \frac{||Q_{sample}(s,a) - \hat{Q}(s,a)||_2}{\text{size of mini-batch}}. \tag{14}$$

4) Delayed update of target NN: Note that in Eq. 11, we still need one pass of forward propagation of the NN to find the $\hat{Q}(s', a')$. When the NN parameters are updated, the sample itself changes too. In the meantime, in one update step, only *one* biased sample of transition (or one mini-batch if using Experienced replay) is used to update the whole neural network. This update can be very inaccurate and may negatively impact the future learning trajectory. To further improve the stability of the algorithm, a dual NN structure is introduced. The decision NN is used to find the current best action and is updated at each time slot by the mini-batch. The target NN is used to find the $Q_{sample}(s, a)$. The target NN is only updated at every T_{target} time slots by copying the current decision NN to it. Thus, the target NN is updated with all transactions (or multiple mini-batches) that are sampled during T_{target} time. For most of the time, the decision NN is optimized towards a ”fixed target” instead of a target that keeps changing. If the decision NN and target NN are denoted as θ and θ', respectively, the new update rule is:

$$\theta \leftarrow \theta + \alpha(r(s,a) + max_{a'}\hat{Q}(s',a'|\theta') - \hat{Q}(s,a|\theta))\nabla_\theta \hat{Q}(s,a|\theta) \tag{15}$$

5) Learning procedure: The complete learning algorithm is shown in Fig. 4 and Algorithm 1.

- Decision Phase: At the beginning of each time slot, the agent observes the state and feeds the input feature vector into the decision NN. Based on the output, the agent either does random exploration or takes the action a that is associated with the maximum NN output. At the end of the time slot, the agent observes the state s' and instant reward r', then records the transition (s, a, s', r) in the memory.
- Learning Phase: A mini-batch is randomly chosen from the memory and is used to update the decision NN's parameters by the stochastic gradient descent algorithm. If the number of time slots is a multiple of T_{target}, then copy the decision NN to the target NN.

Algorithm 1. *Non-linear Q function approximation*

```
-Random initialize decision NN parameters $\theta$;
-Set the target NN parameters $\theta' = \theta$;
for $i$ in $[0, max\ number\ of\ epochs]$ do
    -Random initialize backlog lengths for all buffers: $(l_0, l_1, ..., l_{N-1})$;
    for $j$ in $[0, max\ number\ of\ transitions]$ do
        -Find feature vector $\phi(s)$;
        -$i = random([0, 1])$;
        if $i < \epsilon$ then
            $a = random([a_0, a_1, ...a_{N-1}])$ ;
        else
            $a = argmax_a \hat{Q}(s, a|\theta)$;
        end if
        -Take action $a$, observe $r, s'$ and add $(s, a, r, s')$ to history pool;
        -Randomly sample a mini-batch from history pool;
        -Find $Q_{sample}(s, a|\theta') = r + max'_a \hat{Q}(s', a'|\theta')$ with target NN;
        -Find $Q(s, a|\theta)$ with decision NN;
        -update $\theta$ by Eq.15;
    end for
    -Update target NN: $\theta' = \theta$.
end for
```

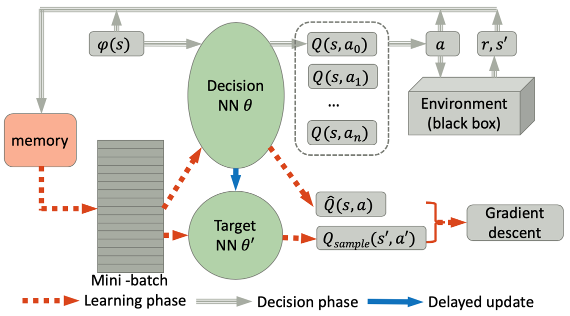

Figure 4: Graph of FC-NN approximation algorithm

6) Improvement of Stochastic Gradient Method (SGM): We use ADAM optimizer [12] as a replacement of SGM for better convergence speed. An ADAM optimizer is a combination of gradient with momentum and RMSprop. It shows better convergence performance in many other deep learning algorithms [13][14]. ADAM's performance in this problem is evaluated in Section V.

V. Performance Evaluation

In this section, we show the performance evaluation of the presented learning algorithm. We assume the AP has a total capacity of 12 Gbps for all the users. While most commonly used wireless routers have a smaller capacity, AP with larger capacity is expected in the future. A scenario with 10 applications is simulated using various channel conditions and traffic patterns. Hyperparameters of the NN are shown in Table I.

Hidden layers	64×32
Mini-batch size	64
History pool size	100000
Parameter initializer	Xavier initializer
Delayed update frequency T_{target}	$10000T_0$
Learning rate α	0.0001
Discount rate β	0.9999

Table I: Hyperparameters of the neural network

First, the applications are associated with a randomly generated p_i and q_i for $i = 0, 1, ..., 9$ to model network conditions and data rates. We also make $\sum_i p_i$ close to the mean of q_i so the total incoming data rate is close to the AP's total channel capacity. Thus, the algorithm is running on a slightly congested environment.

Fig. 5 shows the convergence curve of the total discounted reward in each epoch, up to 5000 epochs (equivalent to 5 seconds in real time). The plain SGD is very noisy especially at the beginning since most of the states are not visited. Due to a fixed learning rate, the total discounted rate converges very slowly when compared to others. The ADAM optimizer with a constant step-size converges faster, but it diverges from the optimal soon after reaching the optimal due to its instability. A carefully chosen shrinking step-size can deal with the instability, but it requires fine tuning of the parameters and a longer converging time. The delayed update of target NN handles the instability well and it maintains a better converge time (< 1000 epochs) as shown in Fig. 5. Fig. 6 shows the average packet loss rate of last 500 epochs of the 5000-epoch training period. Again, ADAM with delayed target NN update outperforms others with a much lower packet loss rate of 0.55%, while the packet loss rate of the other three are 22%, 19% and 3%.

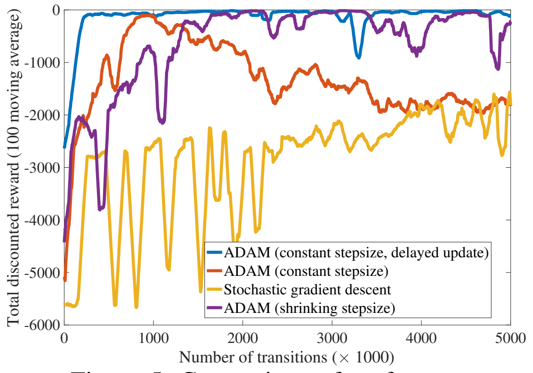

Figure 5: Comparison of performance

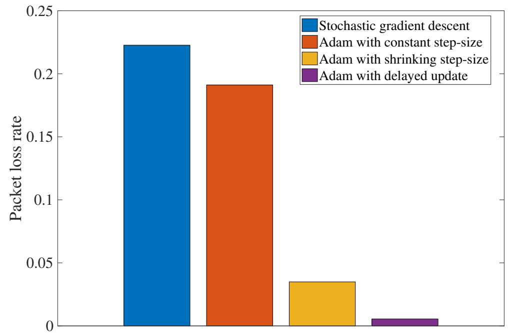

Figure 6: Comparison of average packet loss rate

Now we evaluate the robustness of the algorithm by introducing a sudden change of the arrival data rates. In Fig. 7, we randomly choose an application and assign a large $p_i = 0.35$ to it, while the other 9 applications have $p_i = 0.05$. After training for 2000 epochs, the large incoming data rate is re-assigned to another application. In a real scenario, this can happen when some applications are newly started/reconfigured. It can be seen that the algorithm adapts to the new environment very well within a short period of fluctuation (less than 500 epochs, in our set up that is about 0.5 seconds). The packet loss rate changes with the total discounted reward accordingly.

Now we evaluate the algorithm's performance under a sudden change in channel conditions by changing the PDR. As shown in Fig. 8, in the beginning, the channel PDR associated with a particular application is $q_i = 0.9$. After training for 1500 epochs (1.5 seconds), the PDR is changed to $q_i = 0.65$. This could be a result of this user moving away from the transmitter or behind some obstacles. The channel condition is so bad that the expectation of the number of transitions to finish all packets in 1000 time slots is about 1030. In other words, on average it takes about 1030 transmissions to send all arriving packets in an epoch. We can see that the algorithm converges quickly to a point where the packet loss rate is 0.3%, close to the best it can do. After 3500 epochs (3.5 seconds), the channel PDR is changed to a good value, $q_i = 0.99$, and the packet loss rate is back to nearly 0.

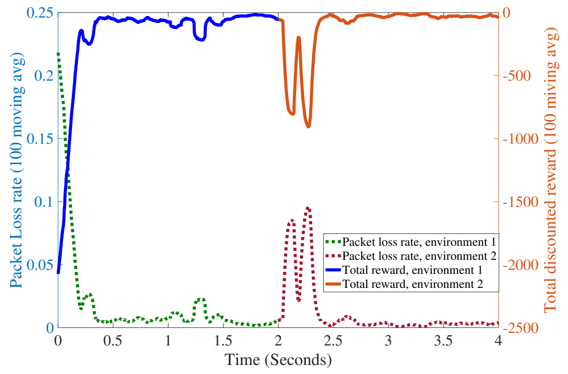

Figure 7: Dynamic policy adjustment with a sudden change in traffic rates

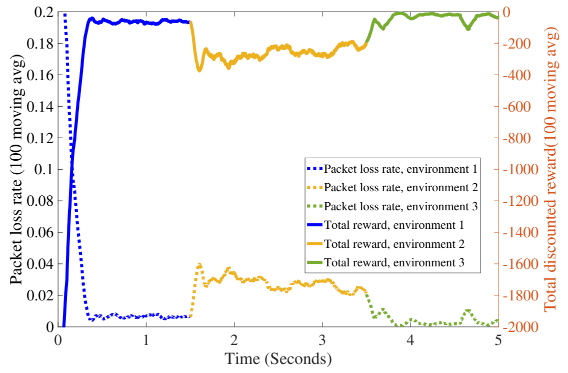

Figure 8: Dynamic policy adjustment with a sudden change of channel conditions

Next, we show the impact of the instant reward C_i of a specific user. In this scenario, we set $p_i = 0.091$ and $q_i = 0.9$ for all applications to eliminate the bias from the environment. Also, we make $\sum_i p_i$ slightly *larger* than average q_i to simulate a more congested network conditions. Fig. 9 shows the average packet loss when the instant rewards are uniformly assigned, that is, $C_i = -10$ for all applications. The packet loss rate for each application is around 2.5 per 1000 time slots. In Fig. 10, application 3's instant reward is increased to -1 to decrease its priority. Because the AP does not have enough resources to fulfill all applications' requirements, it serves fewer packets for application 3. As a result, the packet loss rate of application 3 is increased by a large amount to give the other 9 applications a better performance. This can be verified by the decrease of the average packet loss rate from 2.62 to 1.57 per 1000.

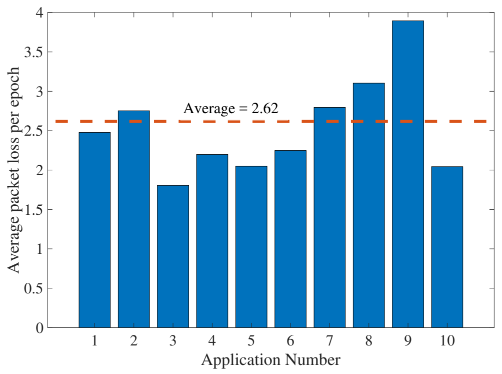

Figure 9: Packet loss comparison with uniformed instant reward

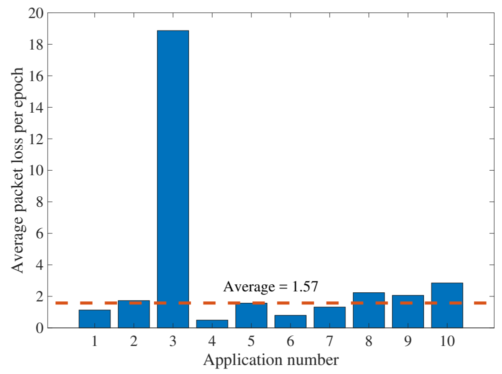

Figure 10: Packet loss comparison with biased instant reward

Remark 2: Importantly, due to the proposed DQN architecture, one pass of forward/backward propagation (computing the output for a given input to the DNN) can be very fast. Also, since the algorithm does not require any hand-labeled data, the agent can be continuously trained and make real-time decisions simultaneously in a practical settings where traffic characteristics and channel conditions change frequently.

VI. Conclusion

In this paper, we describe a DQ framework for implementing adaptive packet schedulers that optimize for application-specific quality of service (QoS) requirements. The DQ framework integrates a deep neural network with online Q-learning algorithms that enables a DQ-based packet scheduler to learn a good packet transmission policy. Importantly, a DQ based packet scheduler can be deployed without any prior training or network traffic models. Rather, the DQ- based packet scheduler progressively learns a good policy in real-time, based directly on the available observations. Our simulation results indicate that the proposed DQ-based scheduler can adapt to the changes in network conditions and/or application requirements in real-time to achieve various QoS.

References

[1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design", *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

[2] H. Shimonishi and S. Ishii, "Virtualized network infrastructure using openflow", in *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*, 2010, pp. 74–79.

[3] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[4] F. S. Melo and M. I. Ribeiro, "Q-learning with linear function approximation", in *Proceedings of the 20th Annual Conference on Learning Theory*, ser. COLT'07, San Diego, CA, USA, 2007, pp. 308–322.

[5] S. Khodayari and M. J. Yazdanpanah, "Network routing based on reinforcement learning in dynamically changing networks", in *ICTAI'05*, Nov. 2005, 5 pp.-366.

[6] K. T. Phan, T. Le-Ngoc, M. van der Schaar, and F. Fu, "Optimal scheduling over time-varying channels with traffic admission control: Structural results and online learning algorithms", *IEEE Transactions on Wireless Communications*, vol. 12, no. 9, pp. 4434–4444, Sep. 2013.

[7] H. Zhang, W. Li, S. Gao, X. Wang, and B. Ye, "Reles: A neural adaptive multipath scheduler based on deep reinforcement learning", in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, Apr. 2019, pp. 1648–1656. DOI: 10.1109/INFOCOM.2019.8737649.

[8] J. Luo, X. Su, and B. Liu, "A reinforcement learning approach for multipath tcp data scheduling", in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2019, pp. 0276–0280. DOI: 10.1109/CCWC.2019.8666496.

[9] I. Comfffdfffda, S. Zhang, M. E. Aydin, P. Kuonen, Y. Lu, R. Trestian, and G. Ghinea, "Towards 5g: A reinforcement learning-based scheduling solution for data traffic management", *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1661–1675, Dec. 2018. DOI: 10.1109/TNSM.2018.2863563.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning", *CoRR*, vol. abs/1312.5602, 2013.

[11] V. Mnih and et.al, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 00280836.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *CoRR*, vol. abs/1412.6980, 2015.

[13] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning", *CoRR*, vol. abs/1611.02167, 2017.

[14] Z. Zhang, "Improved adam optimizer for deep neural networks", in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, Jun. 2018, pp. 1–2. DOI: 10.1109/IWQoS.2018.8624183.