DQR: Deep Q-Routing in Software Defined Networks

Syed Qaisar Jalil, Mubashir Husain Rehmani, and Stephan Chalup

***Abstract*—In this paper, we investigate the task of quality of service (QoS) routing in software defined networks (SDN). We consider delay, bandwidth, loss, and cost as QoS parameters. We propose a new deep reinforcement learning solution for greedy online QoS routing in SDN and call it Deep Q-Routing (DQR). DQR utilises a dueling deep Q-network with prioritised experience replay to compute a path for any source-destination pair request in the presence of multiple QoS metrics. In contrast to existing DRL-based routing methods, the proposed DQR method regards the task of routing as a discrete control problem and uses a reward function comprising weighted QoS parameters. Our simulation results show that DQR substantially improves end-to-end throughput compared to other existing learning based methods.**

***Index Terms*—Quality-of-service Routing, Deep-Q Learning, Software defined network.**

I. Introduction

Software-defined network (SDN) is an emerging networking paradigm that provides features, such as on-demand resource allocation, easy reconfiguration, and programmable network management, which significantly improves network performance. In SDN, network functionalities are logically separated into a control plane and data plane. This is one of the key features that differentiates SDN from traditional networks. The control plane includes the SDN controller which is responsible for implementing all control functionalities required for operational decision making. While the data plane includes hardware devices (such as switches) responsible for the execution of the instructions received from SDN controller. The logically centralised SDN controller has a global view of the network that enables network administrators to dynamically optimise network resources and provide flow-level quality of service (QoS) provisioning.

The SDN controller provides application-oriented services by running different modules inside the controller for various tasks such as QoS routing, resource reservation, network monitoring, and queue management. The QoS routing module is responsible for providing flow-level QoS in terms of different parameters such as delay, loss, and bandwidth. It collects network statistics in real-time and determines the routes for different source-destination pairs which satisfy QoS requirements. Broadly speaking, QoS routing can be divided into two types: *greedy online* and *global offline* [1]. Greedy online QoS routing considers individual traffic flows, i.e., it tries to add a new traffic flow in the network while maintaining the QoS requirements of ongoing flows. On the other hand, global offline QoS routing considers a set of ongoing traffic flows and tries to determine the routes which fulfil the QoS requirements for the selected set of flows. For QoS routing, extensive research efforts have been made for different network settings such as presented in the following surveys [1], [2]. Most of this work is model-based where the underlying assumption is that user demand and network environment can be well modelled. Also, it requires high computational resources to deal with multiple QoS parameters. On the other hand, communication networks have evolved into highly dynamic and complicated networks which makes them hard to model and control.

Since the proposal of deep Q-network (DQN) by DeepMind [3], deep reinforcement learning (DRL) methods have become very popular to be used for complex problems where their ability to learn from experience allows to avoid the development of large accurate mathematical models. DRL combines reinforcement learning (RL) and deep learning to overcome the limitations faced by RL methods such as dealing with large-scale systems. Specifically, DRL uses deep neural networks (DNN) in combination with reinforcement learning methods to improve the learning process. DRL has attracted researchers from various disciplines due to its ability to solve large-scale complex problems, for example, in the field of communication and networking [4].

In the context of RL based routing, Boyan and Littman proposed in their 1994 paper [5] a Q-learning based packet routing approach called Q-routing. A more recent study by Lin et al. [6] uses RL in a software-defined network and proposed a QoS-aware adaptive greedy online routing algorithm. However, due to the use of Q-learning, these approaches do not perform well when the routing complexity increases, i.e., when a larger number of QoS metrics has to be optimised in large-scale problems. Some recent studies used DRL for routing in communication networks, e.g., [7]–[9]. These studies considered global offline routing. They take an ongoing traffic flow traffic matrix (TM) and find the solution for TM using shortest paths. Specifically, they optimise TM using the deep deterministic policy gradient (DDPG) algorithm which is widely used for continuous control problems. However, these DRL methods for global offline routing are limited in their performance because they only consider k-shortest paths for each source destination pair, whereas there can be other paths which can provide better performance.

Instead of formulating the routing problem as continuous control problem and using k-shortest paths, we formulate it as discrete control problem. Specifically, we propose a DQN

S.Q. Jalil and S. Chalup are with the School of Electrical Engineering and Computing, The University of Newcastle, Australia. E-mail: syedqaisar.jalil@uon.edu.au and stephan.chalup@newcastle.edu.au

M.H. Rehmani is with the Department of Computer Science, Cork Institute of Technology (CIT), Ireland. E-mail: dr.m.rehmani@ieee.org

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

based greedy online QoS routing method. Our formulation enables the DRL agent to find a path for each source-destination pair request while optimising QoS metrics. We do not restrict the DRL agent to k-shortest paths rather it constructs the routing path considering the current state of the network along with an optimisation of the QoS metrics. Thus, our agent learns the network topology and simultaneously optimises the QoS metrics.

The organisation of the paper is as follows: Section II provides related work. In Section III we present a brief overview of DRL. Section IV describes the system model and the problem formulation. In Section V, we present design details of DQR. Simulation results are presented and discussed in Section VI and finally Section VII concludes the paper.

II. Related Work

Recently various methods related to DRL based routing have been proposed. The study [10] proposed supervised learning and DRL based routing methods to optimise demand matrix. One of the earlier studies that utilised DDPG algorithm for routing optimisation was by Stampa et al. [7]. The goal of their DRL agent was to reduce the mean network delay. They trained their DRL agent on a 14-node topology for 10 different traffic intensity levels and demonstrated its performance improvements. Various later approaches using the DDPG algorithm addressed different applications and different objectives. For instance, Huang et al. [11] proposed a DDPG based quality of experience optimisation method for multimedia traffic. Xu et al. [8] proposed experience driven routing. They proposed traffic engineering (TE) aware exploration and actor-critic based prioritised experience replay in conjunction with DDPG algorithm for optimising delay. Later they proposed another DRL method for multi-path TCP congestion control [12]. Xiao et al. [13] proposed *Deep-Q* in which they used deep generative networks to infer QoS metrics from real traffic data. Chen et al. [14] proposed DeepRSMA which is a DRL routing framework for optical networks. In [15], an intelligent and scalable framework for routing optimisation, named SINET, has been proposed. SINET optimises routing policies using TM to reduce flow completion time. For SDN-IoT, Guo et al. [16] proposed DQSP which is a DDPG based secure and QoS aware routing method. In the domain of knowledge defined networking, a convolutional neural network (CNN) based QoS aware routing was proposed by [9]. They considered loss and delay as QoS metrics and presented a performance comparison of the proposed DDPG with CNN with dense neural networks for different network settings. Suarez-Varela et al. [17] proposed feature engineering for DRL-based routing in the context of optical transport networks and IP networks. However, the drawback of their proposed method is that their action space consists of predefined k-shortest paths and the problem complexity increases exponentially when they use all valid paths between every source-destination pair.

Despite many research efforts, DRL methods are only used for global offline routing. This is due to the fact that greedy online QoS routing poses several challenges such as learning end-to-end paths for different source-destination pair requests, dealing with invalid actions, avoiding network loops, and optimising different QoS metrics together. To deal with these challenges, we propose DQR which is a dueling DQN with PER based greedy online QoS routing method. DQR not only cope with these challenges but also has a flexible design that makes it topology and network state independent.

III. Deep Reinforcement Learning

In this section, we present a brief overview of deep reinforcement learning. Reinforcement learning (RL) is a field of machine learning in which an agent interacts with the system (environment) and tries to learn its behaviour [18]. Particularly, at each iteration t, the agent senses the current state $\mathbf{s}_t$ of the system, takes an action $\mathbf{a}_t$ based on its past experience and receives a reward r_t. The goal of the agent is to learn the policy $\pi(\mathbf{s})$ which maximise the longterm reward, i.e., highest accumulated reward over time $R_0 = \sum_{t=0}^{T} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)$ where $\gamma \in [0, 1]$ and $r(\cdot)$ represent discount factor and reward function respectively.

Q-learning is one of the most popular RL algorithms that aims at finding an optimal policy [19]. It can be implemented using a Q-table which is updated using

$$\begin{aligned} Q_{t+1}(s_t, a_t) := Q_t(s_t, a_t) \\ + \alpha[R_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)] \end{aligned} \quad (1)$$

where α is the learning rate and $\gamma \in [0, 1]$ represents the discount factor which is used to control the effect of immediate and later rewards. The main idea is to find the temporal difference between predicted and current Q-values. In a state s, the true value of an action a under policy π is $Q_\pi(s, a) \equiv \mathbb{E}[R_1 + \gamma R_2 + ... | S_0 = s, A_0 = a, \pi]$. In each state, optimal value is calculated by selecting the highest valued action, i.e., $Q_*(s, a) = \max_\pi Q_\pi(s, a)$, and therefore, these optimal values are used to derive the optimal policy. However, Q-learning is only effective when the state and action space is small. To overcome the scalability issue, DeepMind presented groundbreaking work [3] in which deep neural network (DNN) was used to approximate $Q_*(s, a)$ value instead of using Q-table, called deep Q-network (DQN).

A DQN is a multi-layered neural network which takes state space vector as input and gives a vector of action values $Q(s, \cdot; \boldsymbol{\theta})$ as output, where $\boldsymbol{\theta}$ represents the parameters of the network. DQN learns an optimal policy based on the received reward from the environment. This means that the policy can become affected even with a minor change in Q-values which results in varied correlations and data distributions between target values and Q-values.

To solve this issue, [3] proposed two methods. Firstly, the use of an experience replay buffer that enables the DRL agent to store past experiences and update DNN by using mini-batches randomly sampled from replay memory. The use of the experience replay mechanism allows the agent to learn from both new and old experiences. Also, these experiences are independent and identically distributed which removes correlations between observations. Secondly, DQN is trained through a separate target Q-network, with parameters $\boldsymbol{\theta}^-$, which estimates target values. But this network is trained after

every τ steps from the primary network such that $\boldsymbol{\theta_t^-} = \boldsymbol{\theta_t}$. Therefore, parameters are updated as follows

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta_t} + \alpha(y_t^{DQN} - Q(s_t, a_t; \boldsymbol{\theta_t})) \bigtriangledown_{\boldsymbol{\theta_t}} Q(s_t, a_t; \boldsymbol{\theta_t}), \quad (2)$$

where α represents a scalar step size and the target y_t^{DQN} is defined as:

$$y_t^{DQN} = r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \boldsymbol{\theta_t^-}) \quad (3)$$

DQN was able to achieve super human-level performance on Atari games. In the sequel several extensions of DQN have been proposed that led to further enhanced performance. For instance, to converge faster and with better stability, authors in [20] proposed a dueling architecture for DQN. In the dueling architecture, a state value function $V(s)$ and an associated advantage function $A(s, a)$ are estimated separately and then combined to estimate an action value function $Q(s, a)$. In DQN, $Q(s, a)$ is obtained by:

$$\begin{aligned} Q(s, a; \boldsymbol{\theta}, \eta, \zeta) &:= V(s; \boldsymbol{\theta}, \zeta) \\ &+ \big(A(s, a; \boldsymbol{\theta}, \eta) - \max_{a'} A(s, a'; \boldsymbol{\theta}, \eta)\big) \end{aligned} \quad (4)$$

where η and ζ are parameters of two streams of fully connected layers. In the dueling architecture, $Q(s, a)$ is obtained by:

$$\begin{aligned} Q(s, a; \boldsymbol{\theta}, \eta, \zeta) &:= V(s; \boldsymbol{\theta}, \zeta) \\ &+ \big(A(s, a; \boldsymbol{\theta}, \eta) - \frac{a}{|\mathcal{A}|} A(s, a'; \boldsymbol{\theta}, \eta)\big) \end{aligned} \quad (5)$$

Another improvement to the performance of DQN was made by using prioritised experience replay (PER) [21] instead of simple ER. The use of ER plays a vital role in the learning of DQN. In ER, experience transitions are uniformly sampled without considering their importance. The idea behind the PER is to sample experience transitions based on their significance. Temporal difference errors are used to measure the importance of transitions. The use of PER enabled DQN to learn efficiently by frequently replaying important experience transitions. We used dueling DQN along with PER as a core algorithm for DQR.

IV. System Model and Problem Formulation

We consider a SDN which consists of three layers as shown in Fig. 1. The data layer also referred to as the infrastructure layer, consists of hardware equipment such as switches. The main responsibility of the data layer is to perform data forwarding among network clusters. The control layer provides a logically centralised controller which enables the communication between the application and the data layer. The control layer provides functionalities like dynamically updating forwarding rules and programming network resources. The communication between the control and the data layer is achieved through southbound interfaces (SBIs) whereas, communication between the control and the application layer is achieved through northbound interfaces (NBIs). The application layer is the highest level layer of SDN which includes storage, servers, data centres, and applications. The concept of application-oriented services is achieved in this layer. Network state information is collected through the data layer and used by various network applications such as network monitoring, network security, and routing.

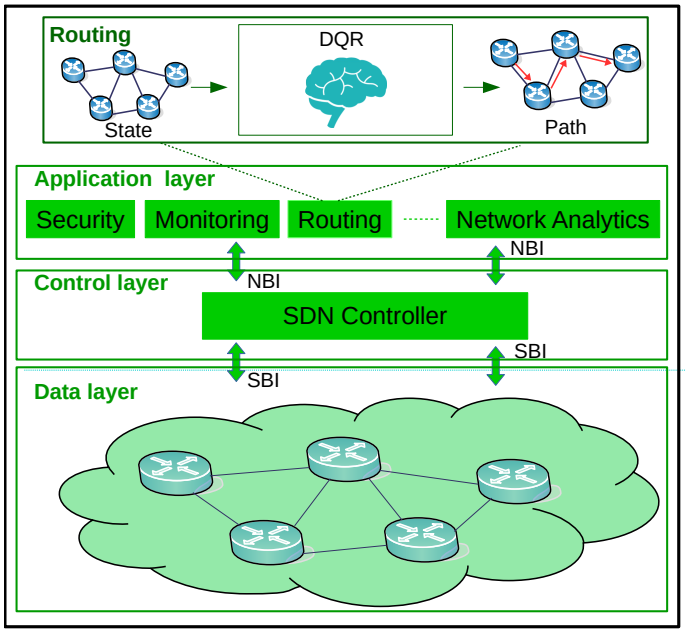


Fig. 1: A software defined network framework consists of data, control and application layer. Various applications, such as routing, are running inside the application layer. The proposed DQR runs inside the routing module where it takes network state as input and gives path as output which is then installed into the switches by SDN controller.

Routing application is responsible for determining the paths for source-destination pair requests. It collects network statistics periodically such that the path found for each source-destination pair request is based on the current state of the network. The proposed DQR method runs inside the routing application where it takes the current state of the network as input and provides a path as the output (more details in Sec. V).

The network is represented as a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of all switches and $\mathcal{E}$ denotes the set of links between them such that $\mathcal{E} = \{(i, j) | (i, j) \in \mathcal{V} \times \mathcal{V}, i \neq j\}$. For any link $(i, j) \in \mathcal{E}$, the delay, loss, bandwidth, and cost (also referred as metrics) values are represented by $\{d_{ij}, l_{ij}, b_{ij}, c_{i,j}\} \in \mathbb{R}_+$, respectively. Here, cost is a general metric which can be used for any QoS parameter such as jitter [2]. Given a source node x with flow f for destination node y, we want to find a path that minimises delay, loss, and cost while maximising the bandwidth.

V. Deep-Q Routing (DQR)

In this section, we present deep-Q routing (DQR) for finding a path from x to y while optimising QoS parameters. DQR is a deep reinforcement learning mechanism that utilises dueling deep Q learning with PER as its core algorithm to find QoS optimised paths. In the following, we present design and implementation details of DQR.

A. State Space

We design the state space in such a way that it captures the current state of the network without including any unnecessary information. For each QoS metric, we define a two-dimensional matrix of size $|\mathcal{V}|\times|\mathcal{V}|$. Since we have real values for each metric and their range varies depending upon the current state of the network, we use a rescaling function that rescales the values of each metric to $[0, 1]$. This not only helps the agent to converge faster but it also enables the agent to deal with a variable range of values for each metric in real-time. For any metric, we define the rescaling function as

$$u_{ij}^{rescaled} = \frac{u_{ij} - \min(\vec{u})}{\max(\vec{u}) - \min(\vec{u})}, \tag{6}$$

where $\vec{u}$ is a vector that consists of selected metric link values. The rescaled values of delay, loss, bandwidth, and cost are denoted as $d_{ij}^{rescaled}, l_{ij}^{rescaled}, b_{ij}^{rescaled}, c_{i,j}^{rescaled}$ respectively. The delay matrix is constructed as

$$\mathcal{D} = [\mathcal{D}(\mathcal{G})]_{ij} = \begin{cases} -d_{ij}^{rescaled} & (i,j) \in \mathcal{E} \\ 2 & i = j = y \\ -2 & i = j = x \\ 0 & i = j, x \neq i = j \neq y \\ -1 & \text{otherwise} \end{cases} \tag{7}$$

The loss matrix $\mathcal{L}$ and the cost matrix $\mathcal{C}$ are constructed in the same way. Since bandwidth needs to be maximised, the bandwidth matrix $\mathcal{B}$ is constructed as follows:

$$\mathcal{B} = [\mathcal{B}(\mathcal{G})]_{ij} = \begin{cases} b_{ij}^{rescaled} - 1 & (i,j) \in \mathcal{E} \\ 2 & i = j = y \\ -2 & i = j = x \\ 0 & i = j, x \neq i = j \neq y \\ -1 & \text{otherwise} \end{cases} \tag{8}$$

B. Action Space

The action space consists of all edges of the network. Specifically, the action space vector is defined as $A = [a_1, a_2, \ldots, a_{|\mathcal{E}|}]$ where each action corresponds to a link in the network $(i, j) \in \mathcal{E}$.

C. Reward Function

The reward function directly affects the learning of the DRL algorithm and should be designed carefully. In our case, reward function incorporates signals to cope with invalid actions, network loops, and the optimisation of multiple metrics. Invalid actions or network loops can occur because the agent is free to choose any action (edge) at any timestep.

In an episode of T timesteps, the DRL agent has to find a path from the source node x to the destination node y. At any node z, the agent selects action a_t at timestep t which corresponds to link (i, j) and receives the reward r_t by the reward function $f\big((i,j)\big)$ as follows:

$$r_t = f\big((i,j)\big), \tag{9}$$

where,

$$f\big((i,j)\big) = \begin{cases} g\big((i,j)\big) & (i,j) \in \mathcal{E}_{valid}^{z} \\ -\frac{|\mathcal{V}|}{2} & \text{otherwise} \end{cases}$$

where $\mathcal{E}_{valid}^{z}$ represents the set of valid actions from node z, i.e., only outgoing links from node z are valid. If the selected action is invalid then the agent receives a reward of $-\frac{|\mathcal{V}|}{2}$, i.e., a penalty. Otherwise, reward is obtained by the function $g((i, j))$ which is defined as follows:

$$g\big((i,j)\big) = \begin{cases} -\frac{|\mathcal{V}|}{3} & (i,j) \in \mathcal{E}_{visited} \\ |\mathcal{V}| & j = y \\ -|\mathcal{V}| & t = |\mathcal{E}| \\ (-d_{ij}^{rescaled} \cdot \phi_1) & \\ +((b_{ij}^{rescaled} - 1) \cdot \phi_2) & \\ +(-l_{ij}^{rescaled} \cdot \phi_3) & \\ +(-c_{ij}^{rescaled} \cdot \phi_4) & \text{otherwise} \end{cases}$$

where $\mathcal{E}_{visited}$ represents a set that is defined as empty at the start of each episode and then becomes populated with the visited links. If the selected link was already visited then the agent receives a reward of $-\frac{|\mathcal{V}|}{3}$. This ensures that agent does not get stuck in network loops by repeatedly selecting the same links in an episode. If the selected link includes the destination node y then this is a terminal state and the agent receives $|\mathcal{V}|$ as reward, i.e., the agent has found the path for this source-destination pair request. We limit each episode to T timesteps to avoid that the agent gets stuck in infinite loops while exploring the action space. If the current timestep is greater than the total number of edges then the agent is lost and the episode is ended with a high penalty of $-|\mathcal{V}|$. Otherwise, the agent receives a reward based on the weighted current rescaled values of the network. Here, $\phi_1, \phi_2, \phi_3, \phi_4 \in (0, 1]$ are tuneable weights which can be used to prioritise any metric during link selection.

The goal of the agent is to accumulate maximum positive reward in each episode. This is supported by the design of the reward function. It penalises the agent most strongly if the agent selects an invalid action because this can lead to divergence. Since agent can select actions from a large action space where only a handful actions are valid at each timestep, the agent should first learn to choose valid actions. Using this knowledge, valid actions are selected and network loops are avoided due to high penalty compared to first time selected actions. The agent further optimises selected actions by minimising negative reward (finding a path that optimises the QoS parameters). Lastly, if the agent is stuck between invalid actions and network loops then the episode is ended by a large penalty of $-|\mathcal{V}|$. The reward function has been designed to encourage the agent to reach the destination node quickly while optimising the QoS parameters.

D. DRL-agent and Environment Implementation

We used the above-defined state space, action space, and reward function for DQR. RLlib[1] was used for the implementation of DQR. The input layer was of dimension $|\mathcal{V}|\times|\mathcal{V}|\times 4$.

[1] https://ray.readthedocs.io/en/latest/rllib.html

Algorithm 1: DQR training algorithm

```
Initialise Environment
Initialise replay memory
Initialise main deep-Q network with weights θ
Initialise target deep-Q network with weights θ⁻ = θ
for episode = 1 to N do
    Reset edge's metrics value
    Normalise each metric value according to eqn. (6)
    Select a random source-destination pair (x, y)
    Create state space s_t according to Sec. V-A
    Create an empty set E_visited
    for t = 1 to T do
        With probability ϵ select random action a_t
        Otherwise select a_t = {arg max Q(s, a; θ)}
        Get valid actions set E^x_valid
        if a_t ∈ E^x_valid then
            Execute action a_t in the environment
            Obtain reward r_t according to eqn. (9)
            Update E_visited
            x = z (where z is the new selected node)
            Update state space s_{t+1}
        else
            Obtain reward r_t according to eqn. (9)
            s_{t+1} = s_t
        end
        Store transition (s_t, a_t, r_t, s_{t+1}) in PER with max
         priority
        Sample random mini-batch of transitions
         (s_j, a_j, r_j, s_{j+1}) from PER according to their
         priority
        Set y_j^{DQN} using eqn. (3)
        Perform gradient descent step
         (y_j^{DQN} − Q(s_j, a_j; θ_t, η_t, ζ_t))^2 w.r.t. θ
        Update target network weights every τ time-steps
        if x == y then
            break
    end
end
```

Two hidden layers with 512 neurons and rectified linear units as activation functions were used. The output layer was of size $|A|$, where at each timestep maximum Q-value was selected as action. In addition, we used the following hyper-parameters in the training process of our DRL-agent: batch size 64, learning rate = 0.001, epsilon = 0.9, epsilon decay = 0.99, and buffer size = 50000.

We developed a custom environment using the NetworkX library [22] to train the DRL agents. Using a custom environment not only gave us the flexibility to use it for any size of the network graph but it also sped up the training process. We defined a range of values for each QoS metric (delay, bandwidth, loss and cost) from which edge values were selected uniformly. Note that once the agent is trained with a custom environment, the saved neural network model can be used with any network simulator.

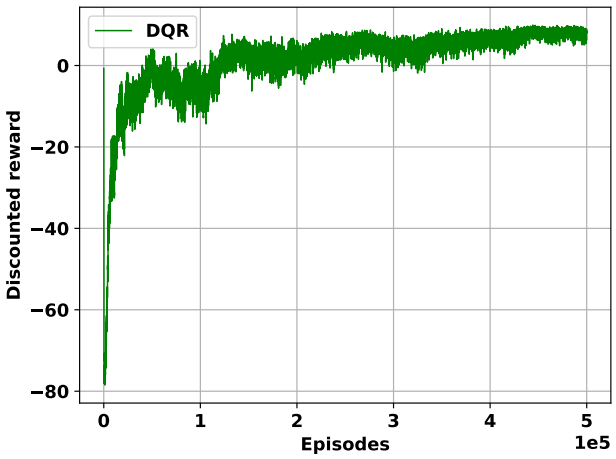


Fig. 2: DQR training performance

E. Training Algorithm

The training process of DQR (utilising DQN as DRL algorithm) is given in the above listing Algorithm 1. At the start of the algorithm, an instance of the environment is created by specifying the number of nodes and edges of the graph. Then, the replay buffer, main Q-network, and target Q-network are initialised. The algorithm runs for a total of N episodes. At the start of each episode, the edges' metric values (delay, bandwidth, loss, and cost) are reset and normalised according to equation (6). Then, a random source-destination pair is selected and a state space vector is created as described in Section V-A. An empty set $\mathcal{E}_{visited}$ is created that later is used to inform the reward function about already visited edges. Each episode has a duration of T time-steps (where $T = |\mathcal{E}|$). At each time-step t, an action a_t is selected using an epsilon-greedy approach. If the selected action is a valid action, then it is executed in the environment and reward r_t is obtained using equation (9), a_t is included in $\mathcal{E}_{visited}$, and the state space vector is updated. Otherwise, if the selected action is not a valid action, r_t is obtained using equation (9) and the same state space is used unchanged for the next iteration. After obtaining s_t, a_t, r_t, s_{t+1}, the transition is stored in the experience replay buffer. Then, a random mini-batch of transitions is sampled from the replay buffer and the weights of the deep neural network are optimised using gradient descent with respect to θ to minimise the loss. The target Q-network is updated after every τ steps. The iteration of episodes ends when the destination node has been found or if $t > |\mathcal{E}|$.

Figure 2 shows the training performance of DQR in our experiments on a widely used 14-node 21-bidirectional NSFNET communication topology [17]. The y-axis presents the discounted reward while the x-axis shows the number of episodes. It can be observed that, at the start of training, the agent spends most of its time on exploring the environment while it receives penalties for invalid or already selected actions. Once the agent starts exploiting its knowledge, it tries to maximise the reward by avoiding invalid actions and network loops. As the discounted reward gets closer to 0, the

agent has already learnt the topology. Now it tries to optimise QoS metrics for different network states. It can be seen in Figure 2 that the discounted reward becomes positive after this stage after about 120 episodes. At this stage the agent has successfully learnt the communication topology and is also optimising the QoS parameters while selecting paths for different source-destination pair requests.

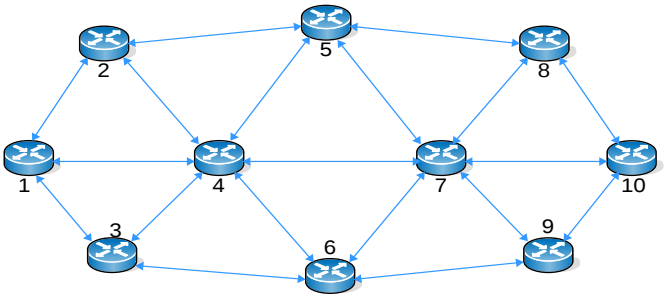


Fig. 3: 10-node communication topology adopted from [1]

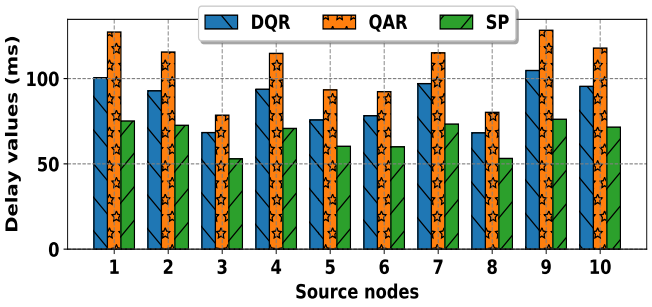
VI. Performance Evaluation

In this section, we present our simulation results that show how well the proposed DQR method performs. The experiments benchmark DQR against two other greedy online routing methods. The first method is a shortest path (SP) approach where the path length is measured by delay. The second method is QAR which is an on-policy reinforcement learning based QoS-aware adaptive algorithm [6]. It uses softmax for action selection and its Q-values are updated using SARSA [18].

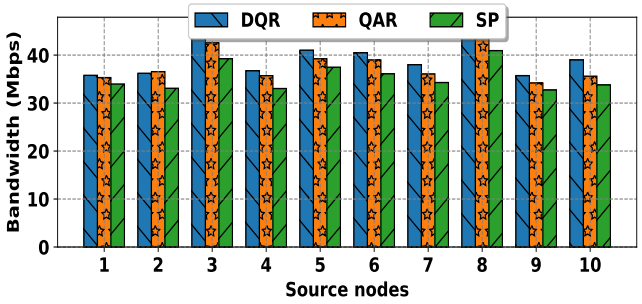
Our simulation experiments utilised two different communication topologies. The first topology is a 10-node topology which is adopted from [1] and is shown in Figure 3. The second topology is a widely used 14-node NSFNET topology [17]. Link delay, bandwidth, loss rate, and cost values were uniformly selected in the following ranges (1, 100) ms, (50, 100) Mbps, (0.01, 1), and (1, 100), respectively. For the cost metric, lower values are better. The reason for selecting link values from the above mentioned ranges is that it allowed us to verify the effectiveness of the proposed method for different network conditions (varying network states). We ran simulations for the three different settings presented in the following paragraphs.

In the first case, we run extensive numerical simulations using the 10-node communication topology. Figure 4 provides an overview of the analysis of the QoS metrics at each communication node in our simulation experiments. For each source node we selected a destination node 1000-times randomly, where every source-destination pair selection was associated with a different network state (link metric values were uniformly selected from the ranges listed above).

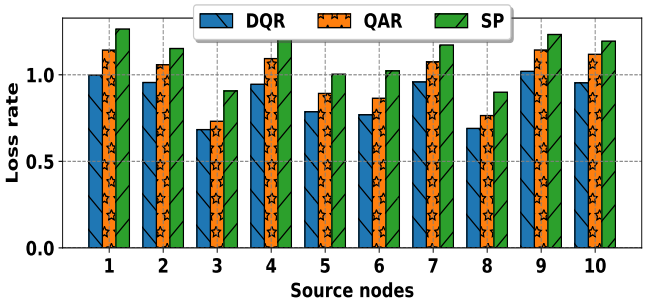
Figure 4a depicts the average end-to-end delay for each approach. QAR has the worst performance in terms of delay. This is due to the use of Q-tables for the optimisation of multiple QoS metrics simultaneously. The performance limitation of Q-tables did not allow QAR to optimise delay. However, QAR showed better performance for the remaining QoS metrics.



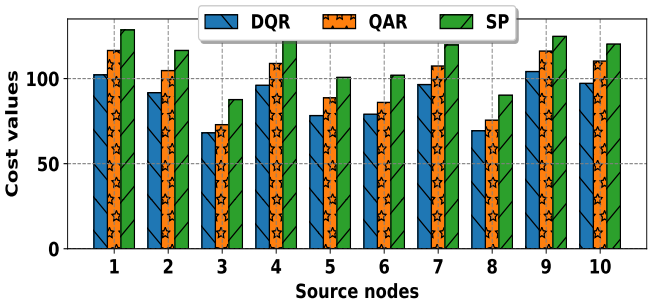
(a) Avg end-to-end delay of selected paths



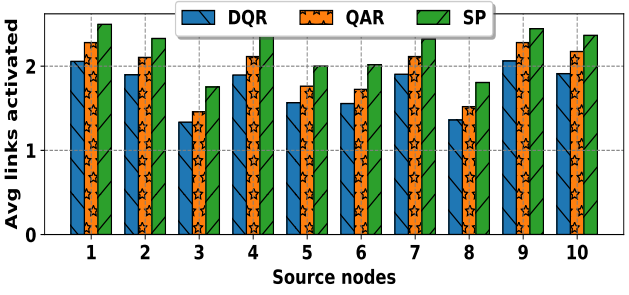
(b) Avg minimum bandwidth of a link in selected paths



(c) Avg end-to-end loss rate of selected paths

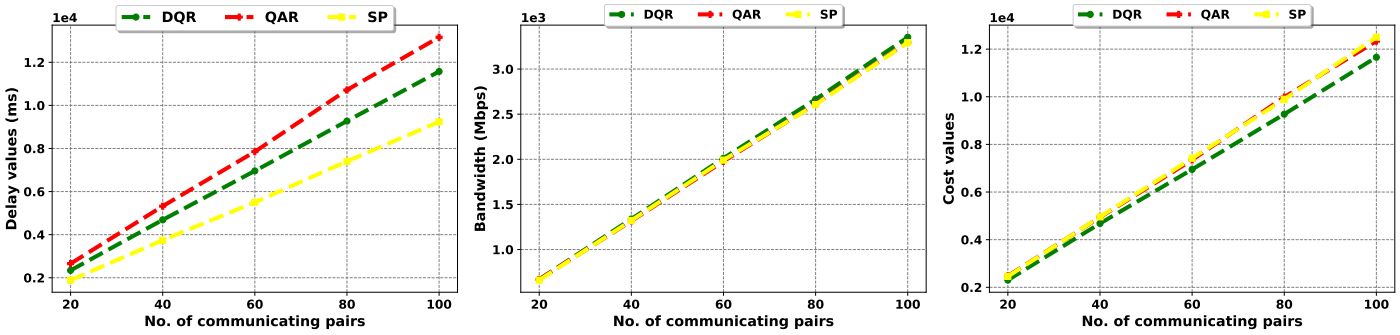


(d) Avg end-to-end cost of selected paths



(e) Average number of links activated for selected paths

Fig. 4: Simulation results for 10-node topology



(a) Avg end-to-end delay (b) Avg bandwidth (c) Avg end-to-end cost

Fig. 5: Simulation results for NSFNET topology

DQR performed better compared to QAR. SP was the best performer in terms of delay which is plausible because SP only considers delay while selecting a path. Figure 4b shows the average minimum bandwidth of a link in selected paths. It can be seen that DQR has a higher bandwidth than QAR and SP. Figures 4c and 4d present the average end-to-end loss and the cost of the selected paths. For both metrics, SP shows the worst performance, followed by QAR, while DQR significantly outperforms SP and QAR. Finally, Figure 4e shows the average number of links activated in selected paths for each source node. DQR has the least number of link activations when compared to QAR and SP. This shows that DQR can achieve better performance in comparison to QAR and SP even if it selects a lower number of links.

As the results of Figure 4 show, SP only performs well in the case of end-to-end delay. The reason is that SP only considers the delay metric while selecting a path and ignores other QoS metrics. Thus, SP should be used for those network applications which only have delay requirements. This also applies to global offline DRL based routing methods where only SP paths are considered. The performance of QAR in Figure 4 shows that it is able to optimise QoS parameters while selecting paths. QAR does not select paths based on a single parameter like SP but it tries to optimise multiple QoS metrics. Compared to SP, QAR showed far better performance with respect to all QoS metrics except delay. However, when compared to DQR, QAR only achieved limited performance. The reason is that QAR uses Q-tables for learning the complex task of QoS routing. As the problem complexity increases (in this case it depends on the number of QoS metrics), Q-table based Q-learning suffers performance issues.

In our experiments DQR performed better than SP and QAR. One of the reasons is that DQR uses a dueling DQN network at its core. However, using a dueling DQN alone is not sufficient to solve the complex task of greedy online QoS routing. There are multiple factors that can lead to divergence such as learning the communication topology, dealing with invalid actions, avoiding network loops, and optimising QoS parameters. The proposed design of the state space, action space, and, most importantly, the design of the reward function made sure that the DRL agent was able to overcome these challenges. It should be emphasised that the design of DQR is not topology dependent and the number of QoS metrics can be varied.

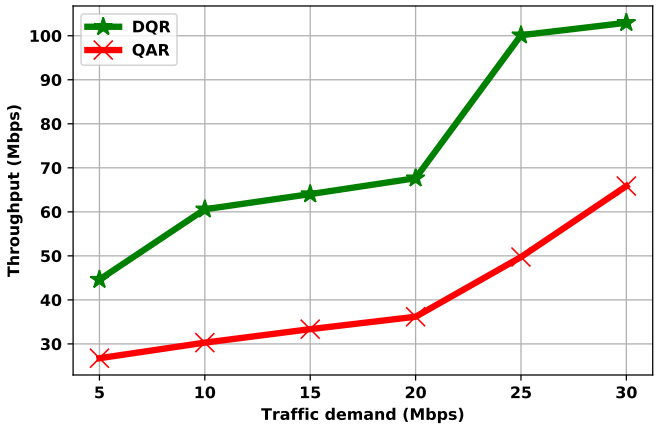


Fig. 6: End-to-end throughput of DQR vs DQR

In the second case of the 14-node NSFNET [17] communication topology, we ran numerical simulations for different numbers of communicating source-destination pairs. Figure 5 presents our results using the NSFNET topology network simulation for delay, bandwidth, and cost. The x-axis represents the number of communicating source-destination pairs while the y-axis represents the QoS metrics. It can be seen that with an increasing number of source-destination pair requests, the overall network usage also increases in each case. In this second case that uses a different communication topology (14-node) and different network settings, we obtained similar results as in the first case (10-node) that was discussed above. Figure 5a shows the average delay achieved by DQR, QAR and SP for different numbers of communicating source-destination pairs. As explained earlier, again QAR has maximum average delay for all communicating source-destination pairs. While DQR performs better than QAR, it is outperformed by SP in terms of delay. Again SP can only perform better in the case of delay and it suffers for other QoS metrics. Figures 5b and 5c show the results for the bandwidth and cost metrics and that DQR performs better than QAR and SP.

Finally, we ran simulations for varying traffic demands using the 10-node communication topology (Figure 3). We used Mininet [23] and Ryu [24] as SDN emulator and controller, respectively. We generated network traffic using iPerf3 [25]. The link metric values were selected uniformly within given ranges, specifically: delay (50, 100) ms, bandwidth (100, 150) Mbps, loss rate (0.001, 0.1), and cost (50, 100). We selected 9 random source-destination pairs that started communicating with 5Mbps traffic demand. During simulation, the traffic demand for each pair was increased in steps of 5Mbps until it reached 30Mbps. We used end-to-end throughput as the performance metric for comparison. The corresponding simulation results for DQR and QAR are shown in Figure 6 where the x-axis shows the traffic demand of each communication session and the y-axis presents the total end-to-end throughput for each traffic demand. It can be seen that DQR is able to achieve higher end-to-end throughput than QAR for each traffic demand level.

The above presented results show that DQR can be used for greedy online QoS routing and that it achieves better performance than QAR and SP with respect to different QoS metrics by efficiently utilising network resources. The design of DQR is flexible enough that it can be used for any network topology and it can be used for different number of QoS metrics. The normalisation of the QoS metrics while training enables DQR to be applicable in any real world scenario. In summary, our results indicate that DQR can be used for a variety of real world network applications, while efficiently utilising network resources.

VII. Conclusion

In this paper, we proposed DQR for greedy online QoS routing in SDN. DQR uses a dueling deep Q-network with prioritised experience replay to learn the network topology in the presence of multiple QoS metrics (delay, bandwidth, loss, and cost). Different from existing DRL-based routing methods which use shortest paths, DQR learns the network topology. The flexible design of the reward function allows DQR to optimise QoS metrics while routing and to avoid invalid actions and network loops. Our simulation results demonstrated that DQR can significantly reduce delay, cost, and loss, while maximising bandwidth when compared to other existing learning methods for greedy online routing.

Acknowledgements

SQJ was supported by a UNRSC50:50 PhD scholarship at the University of Newcastle, Australia.

References

[1] J. W. Guck, A. V. Bemten, M. Reisslein, and W. Kellerer, "Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 388–415, 2018.

[2] M. Karakus and A. Durresi, "Quality of service (qos) in software defined networking (sdn): A survey," *Journal of Network and Computer Applications*, vol. 80, pp. 200–218, 2017.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[4] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, 2019.

[5] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," in *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds. Morgan-Kaufmann, 1994, pp. 671–678.

[6] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo, "QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach," in *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 25–33.

[7] G. Stampa, M. Arias, D. Sanchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[8] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 1871–1879.

[9] Q. T. A. Pham, Y. Hadjadj-Aoul, and A. Outtagarts, "Deep reinforcement learning based QoS-aware routing in knowledge-defined networking," in *Qshine 2018-14th EAI International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, 2018, pp. 1–13.

[10] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route with deep rl," in *NIPS Deep Reinforcement Learning Symposium*, 2017.

[11] X. Huang, T. Yuan, G. Qiao, and Y. Ren, "Deep reinforcement learning for multimedia traffic control in software defined networking," *IEEE Network*, vol. 32, no. 6, pp. 35–41, 2018.

[12] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-driven congestion control: When multi-path tcp meets deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1325–1336, 2019.

[13] S. Xiao, D. He, and Z. Gong, "Deep-q: Traffic-driven qos inference using deep generative network," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018, pp. 67–73.

[14] X. Chen, B. Li, R. Proietti, H. Lu, Z. Zhu, and S. B. Yoo, "Deeprmsa: a deep reinforcement learning framework for routing, modulation and spectrum assignment in elastic optical networks," *Journal of Lightwave Technology*, vol. 37, no. 16, pp. 4155–4163, 2019.

[15] P. Sun, J. Li, Z. Guo, Y. Xu, J. Lan, and Y. Hu, "Sinet: Enabling scalable network routing with deep reinforcement learning on partial nodes," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019, pp. 88–89.

[16] X. Guo, H. Lin, Z. Li, and M. Peng, "Deep reinforcement learning based qos-aware secure routing for sdn-iot," *IEEE Internet of Things Journal*, 2019.

[17] J. Suarez-Varela, A. Mestres, J. Yu, L. Kuang, H. Feng, P. Barlet-Ros, and A. Cabellos-Aparicio, "Feature engineering for deep reinforcement learning based routing," in *IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[19] C. J. C. H. Watkins, "Learning from delayed rewards," King's College, Cambridge University, UK, 1989.

[20] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International Conference on Machine Learning*, vol. 48, 2016, pp. 1995–2003.

[21] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[22] D. A. S. Aric A. Hagberg and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *7th Python in Science Conference (SciPy)*, 2008, pp. 11–15.

[23] Mininet, Available at, http://mininet.org/, Accessed September 2018.

[24] Ryu, Available at, http://osrg.github.io/ryu/, Accessed September 2018.

[25] iPerf3, Available at, https://iperf.fr/, Accessed September 2018.