A Packet Routing using Lightweight Reinforcement Learning Based on Online Sequential Learning

Kenji Nemoto and Hiroki Matsutani
Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan 223-8522
Email: {kenji,matutani}@arc.ics.keio.ac.jp

***Abstract*—Existing simple routing protocols (e.g., OSPF, RIP) have some disadvantages of being inflexible and prone to congestion due to the concentration of packets on particular routers. To address these issues, packet routing methods using machine learning have been proposed recently. Compared to these algorithms, machine learning based methods can choose a routing path intelligently by learning efficient routes. However, machine learning based methods have a disadvantage of requiring training time. Therefore, we use a lightweight machine learning algorithm, OS-ELM (Online Sequential Extreme Learning Machine), to reduce the training time in this paper. There is a previous work about reinforcement learning method using OS-ELM, though it has a problem of low learning accuracy. Hence, we propose OS-ELM QN (Q-Network) with a prioritized experience replay buffer and multi-agent learning function to improve the learning performance. It is compared to a deep reinforcement learning based packet routing method using a network simulator. Experimental results show that the introduction of the experience replay buffer improves the learning performance. In terms of learning speed, OS-ELM QN achieves approximately 2 times speedup than a DQN (Deep Q-Network). The multi-agent learning further improves the learning speed of OS-ELM QN.**

1. Introduction

In the past few years, the amount of traffic flowing through the Internet has increased rapidly [1]. Existing routing protocols such as OSPF [2] and RIP [3] may not be able to deal with the increase of network traffic. For example, OSPF protocol uses Dijkstra algorithm to find the shortest path without considering the congestion. Therefore, when a data flow increases, it can overload certain routers and reduce a throughput in the network. On the other hand, packet routing methods using machine learning have been proposed recently. These methods can intelligently select a routing path by utilizing a high representational ability to take into account complex information. Some previous works report that machine learning methods achieve a higher throughput than OSPF [4] [5].

However, many of the previous methods only aim to improve a packet transfer efficiency; there has been little research on reducing the training costs. A lower training cost has some advantages. Since network conditions change from time to time, it is better to shorten the training time to deal with these changes. In addition, all the network nodes may not have computing resources enough to train deep neural networks.

In this paper, we propose a packet routing method using OS-ELM (Online Sequential Extreme Learning Machine), which enables a sequential learning of neural networks. It is known as a lightweight machine learning method compared to deep neural networks using a backpropagation algorithm.

A reinforcement learning method using OS-ELM for Q-learning has already been proposed [6]. This previous work used a random update technique, which has a disadvantage of slow convergence speed during the training. In this paper, we newly introduce an experience replay buffer to OS-ELM-based reinforcement learning in order to stabilize the training. In addition, multi-agent learning is implemented to further speed up the training.

In this paper, we design the state and reward to achieve a lower latency in the packet transfer. Specifically, we aimed to avoid congestion by using negative numbers of delays in the reward. We evaluate OS-ELM QN as a reinforcement learning based packet routing method.

The rest of this paper is organized as follows. Section 2 describes preliminary knowledge. Section 3 overviews related works. Section 4 proposes OS-ELM QN for packet routing. Evaluation results in terms of learning performance, execution time, packet routing performance, and multi-agent learning are presented in Section 5. Section 6 discusses the usefulness of OS-ELM QN. Section 7 concludes this paper.

2. Preliminaries

This section introduces OS-ELM (Online Sequential ELM), DQN (Deep Q-Network), prioritized replay buffer, and multi-agent learning.

2.1. OS-ELM

OS-ELM [7] is an online sequential learning algorithm for 3-layer neural networks that consist of an input layer, a hidden layer, and an output layer. Here, we assume that the numbers of their nodes are n, $\tilde{X}$, and m nodes, respectively. Figure 1 shows an example network model of OS-ELM.

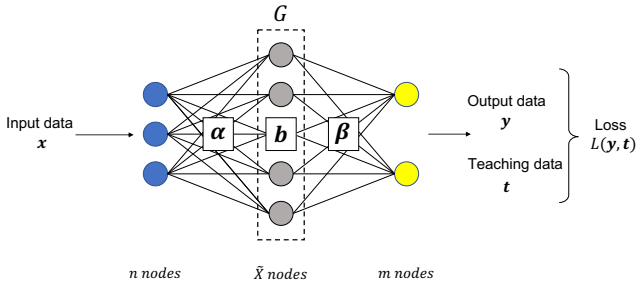

Figure 1. OS-ELM (Online Sequential Extreme Learning Machine)

$\boldsymbol{\alpha} \in \boldsymbol{R}^{n \times \tilde{X}}$ is an input weight matrix between the input and hidden layers, $\boldsymbol{\beta} \in \boldsymbol{R}^{\tilde{X} \times m}$ is an output weight matrix between the hidden and output layers, and $\boldsymbol{b} \in \boldsymbol{R}^{\tilde{X}}$ is a bias vector of the hidden layer.

Assuming that the i-th training chunk $\{\boldsymbol{x}_i \in \boldsymbol{R}^{k_i \times n}, \boldsymbol{t}_i \in \boldsymbol{R}^{k_i \times m}\}$ with batch size k_i is given, the i-th optimal solution $\boldsymbol{\beta}_i$ can be computed as the following equation.

$$\begin{aligned} \boldsymbol{P}_i &= \boldsymbol{P}_{i-1} - \boldsymbol{P}_{i-1}\boldsymbol{H}_i^\top(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^\top)^{-1}\boldsymbol{H}_i\boldsymbol{P}_{i-1} \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i\boldsymbol{H}_i^\top(\boldsymbol{t}_i - \boldsymbol{H}_i\boldsymbol{\beta}_{i-1}), \end{aligned} \tag{1}$$

where $\boldsymbol{H}_i$ is defined as $\boldsymbol{H}_i \equiv G(\boldsymbol{x}_i \cdot \boldsymbol{\alpha} + \boldsymbol{b})$ using an activation function G.

The initial values $\boldsymbol{P}_0$ and $\boldsymbol{\beta}_0$ are precomputed as follows.

$$\begin{aligned} \boldsymbol{P}_0 &= (\boldsymbol{H}_0^\top\boldsymbol{H}_0)^{-1} \\ \boldsymbol{\beta}_0 &= \boldsymbol{P}_0\boldsymbol{H}_0^\top\boldsymbol{t}_0 \end{aligned} \tag{2}$$

As shown in Equation 1, the output weight matrix $\boldsymbol{\beta}_i$ and its intermediate result $\boldsymbol{P}_i$ are computed from the previous training results $\boldsymbol{\beta}_{i-1}$ and $\boldsymbol{P}_{i-1}$. Thus, OS-ELM can sequentially update the model with a newly-arrived target chunk in one shot.

2.2. DQN

DQN [8] is known as a typical reinforcement learning algorithm. $Q_{\theta_1}(s_t, a_t)$ represents a Q-value in time step t when taking action a_t in state s_t. θ_1 represents a set of neural network parameters.

In DQN, a target signal can be computed as follows.

$$f(r_t, s_{t+1}, d_t) = r_t + (1 - d_t)\gamma \max_{a \in A} Q_{\theta_2}(s_{t+1}, a), \tag{3}$$

where $\gamma \in [0, 1]$ is a discount rate that determines the importance of the next step, r_t represents the reward for transitioning from s_t to s_{t+1}, and d_t indicates whether the episode is finished, which is expressed as 1 or 0. In addition, Equation 3 uses a fixed target Q-Network technique. If θ_1 is changed each time while it is used for predicting the Q-value, then the training process becomes unstable. To suppress this issue, a target Q-Network θ_2 is separated from the main Q-Network θ_1. θ_2 is used to generate the target Q-value for the reinforcement learning while it is periodically updated by θ_1. Then, the loss value $L(\theta_1)$ is computed with the following equation.

$$L(\theta_1) = \mathbb{E}[(Q_{\theta_1}(s_t, a_t) - f(r_t, s_{t+1}, d_t))^2], \tag{4}$$

where $\mathbb{E}$ means an expectation value. New parameters are computed using an optimization algorithm, and they are applied to θ_1.

In DQN, an experience replay technique [8] is used to acquire a set of experiences. In Figure 3(b), a replay buffer uses this technique. An experience means a set of s_t, a_t, r_t, s_{t+1}, and d_t. If a training is performed every time an experience is generated, the training is affected by a temporal dependence of the generated experiences. To suppress this negative impact, a batch of experiences is randomly picked up from the replay buffer.

2.3. Prioritized Replay Buffer

In addition to the experience replay buffer, a recent reinforcement learning uses a prioritized experience replay buffer technique [9].

The random sampling method described in Section 2.2 is sometimes inefficient for preferentially training certain transitions of high importance. To address this issue, the prioritized experience replay buffer technique assigns weights to the sampling probability of each experience based on a priority. In Figure 3(c), a replay buffer uses this prioritized sampling.

A sampling probability of an experience i is calculated based on priority p of the experience as follows.

$$P_i = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \qquad (p_k \neq 0), \tag{5}$$

where α is a hyper-parameter that weights the priority; if α is 0, it is equivalent to a random sampling.

2.4. Distributed Reinforcement Learning

Distributed Prioritized Experience Replay (Ape-X) [10] is one of well-known distributed reinforcement learning systems. By decoupling actors or agents from a learner, an agent can get experiences more effectively than previous distributed learning methods.

Figure 2 shows Ape-X architecture. The agents take actions and observe the environments to obtain the rewards. After that, they add states, actions, and rewards into a global buffer. A learner samples the experiences from the global buffer to learn. We use this algorithm as a baseline. We will describe the detail in Section 4.1.

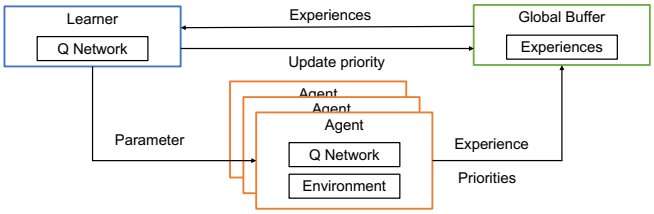

Figure 2. Learning model of Apex-DQN

3. Related Work

3.1. Reinforcement Learning using OS-ELM

As mentioned earlier, an OS-ELM-based reinforcement learning is proposed [6]. In this section, we describe the reinforcement learning techniques using OS-ELM and their shortcomings that degrade learning efficiency.

3.1.1. Simplified Output Model. The loss value of DQN is calculated by Equation 4. In typical DQNs, the i-th node of an output layer represents the Q-value of the i-th action, and the Q-Network is trained so that the i-th node can predict $Q(s, a_i)$. On the other hand, as Equation 1 shows, OS-ELM requires teacher data $\boldsymbol{t} \in \boldsymbol{R}^m$ to update $\boldsymbol{\beta}$. Since OS-ELM analytically derives the weight parameters, it is necessary to specify inputs and outputs explicitly. In the previous work [6], a set of state variables and a scalar variable that represents actions are given as inputs to the neural network. However, we consider that representing an action as a scalar value may not be scalable, and thus we use a vector variable instead of a scalar variable. We will describe the detail in Section 4.1.1.

3.1.2. Random Update. As shown in Section 2.2, DQN typically trains its neural network parameters in a batch manner and uses the experience replay technique to form a batch randomly. On the other hand, this previous work [6]

fixes the batch size to 1 and randomly decides whether to train using an incoming experience. Figure 3(a) illustrates the random update technique, in which Q-Network accepts the experiences by probability P.

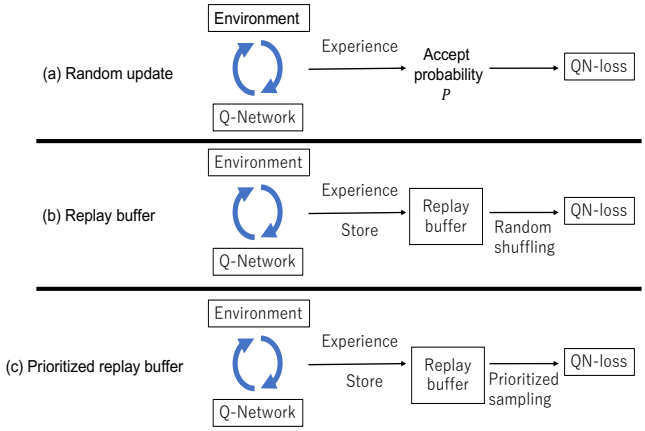

Figure 3. Three experience sampling techniques

By fixing the batch size to 1, an inverse matrix operation $(\boldsymbol{I} + \boldsymbol{H}_i \boldsymbol{P}_{i-1} \boldsymbol{H}_i^\top)^{-1}$ in Equation 1 can be interpreted as a simple division $\frac{1}{1+(\boldsymbol{H}_i \boldsymbol{P}_{i-1} \boldsymbol{H}_i^\top)}$ when training an incoming experience. This technique can eliminate the need for the pseudo-inverse operation that typically requires SVD or QR decomposition. However, the training may be affected by the temporal dependence since a replay buffer is not used. In this paper, we compare this technique with other sampling techniques.

3.1.3. Forget Rate. The distribution of data given by the environment may change as time goes by. To gradually reduce the impact of old trained data, $\boldsymbol{P}_i$ is recomputed before updating the parameters using Equation 1 as follows.

$$\boldsymbol{P}_{i-1} \leftarrow \boldsymbol{P}_{i-1} / \lambda^2, \tag{6}$$

where $\lambda \in [0, 1]$ is a forget rate. Thus, the impact of old trained data is adjusted by the λ parameter.

3.2. Packet Routing using Machine Learning

Supervised learning methods: In this section, we describe some previous research about packet routing methods using machine learning. In [11], a data-driven supervised learning model is designed to minimize maximum link utilization. It implies that well-predicting traffic conditions would be challenging when using existing methods. This problem is one of motivations to use supervised learning.

In [4], a supervised learning method is proposed to optimize routing efficiency using three phases: Initial phase, Learning phase, and Running phase. In Initial phase and Learning phase, each node collects traffic information by using existing routing methods (e.g., OSPF) and starts training. In Running phase, each node routes packets by using the trained neural networks. It achieves a higher forwarding efficiency than OSPF by reducing signaling overhead [4].

Deep reinforcement learning methods: As mentioned in [12], DRL methods are superior to supervised methods in some respects. For example, supervised training methods require labeling a large amount of information in the network, which is an arduous task. In addition, DRL can autonomously monitor and control networks by training.

In [5], a system model of the reinforcement learning (i.e., states, actions, and rewards) is defined and a central controller which routes all the packets is designed. It shows a higher forwarding efficiency than OSPF.

4. Proposed Routing Method

4.1. OS-ELM Q-Network

In this section, we propose OS-ELM QN (OS-ELM Q-Network), which is an improved version of the reinforcement learning method using OS-ELM [6]. OS-ELM QN uses Ape-X as a baseline and replaces the neural network model and optimization algorithm with OS-ELM as we introduced in Section 2.1. The agent, learner, and optimization algorithms are shown in Algorithms 1, 2, and 3.

4.1.1. Agent's Behavior. Algorithm 1 shows the behavior of the agent. It is running on the agents in Figure 2.

There are multiple agents. Each agent has an environment and a neural network (lines 2-3). The agent chooses an action according to the ϵ-greedy policy (line 5) [8]. That is, an action is chosen randomly with probability ϵ or using an inferred result with probability (1-ϵ).

The action is represented as a vector (lines 6-9). Compared to the previous method [6], we change the input format. As we described in Section 3.1.1, a set of state variables and a scalar action variable is used as an input data for the Q-Network [6]. However, if a single scalar is used to represent multiple actions, different scalar values are defined for different actions (e.g., 0.5 for a_0 and -0.5 for action a_1). The mapping between the scalar values and corresponding actions may affect the results. In this paper, the action is given as a vector instead of a scalar. For example, assuming that the number of input states is 3 and the number of input actions is 2. In this case, the neural network model is designed as shown in Figure 4. The action vector is fed to this single neural network as $[1, 0]$ for action $\hat{a}_0$ and $[0, 1]$ for action $\hat{a}_1$, in addition to the state variables. The output of the neural network is the Q-value corresponding to the given action and state. Using this neural network, we get two Q-values, $Q(s, \hat{a}_0)$ and $Q(s, \hat{a}_1)$, by inferring twice with different action vectors. The action that outputs the largest Q-value is selected as the next action.

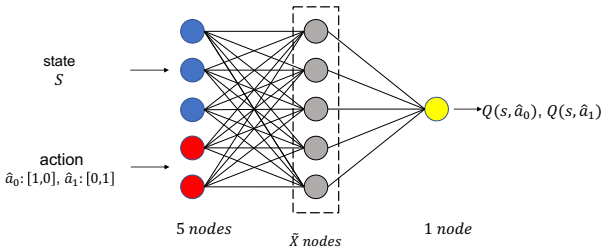

Figure 4. OS-ELM QN model

Then, an agent takes the inferred action and acquires the next state s_t, reward r_t, and finish flag d_t (line 12). Each agent stores the experience in the global buffer (line 13) and calculates its priority (lines 14-17). The parameters θ_1 are periodically updated by given global parameters (line 18). The above operations can be performed asynchronously.

4.1.2. Learner's Behavior. Algorithm 2 shows a training algorithm that updates the Q-Network parameters. It is running on the leaner in Figure 2.

The experiences generated by the agents are stored in a fixed-length global buffer, and the learner samples the experiences from the global buffer (line 4). It updates the priorities of the experiences in the global buffer (line 5) and then it updates the neural network of the learner (line 6). The sequential learning method for updating the parameters is shown in Algorithm 3. Periodically, the learner shares the updated neural network parameters with the agents.

As we described in Section 3.1.2, a random update technique is used in the previous work [6]. Compared to the previous work, we use a prioritized experience replay inspired from Ape-X. This is because the previous work [6] targets an edge environment with a limited memory capacity. However, in this paper, our targets are network routers which are expected to have more memory capacity than tiny edge devices. Therefore, we introduce a prioritized experience replay buffer into OS-ELM QN. This modification is expected to improve learning performance.

4.1.3. Online Sequential Learning. Algorithm 3 shows an algorithm of online sequential learning of OS-ELM QN. Algorithm 2 uses this algorithm to update the parameters $\boldsymbol{\beta}$. If the Q-Network is trained at the first time, an initial learning is executed (lines 3-7). After that, a sequential learning is executed to train the Q-Network (lines 9-12).

Algorithm 1 Agent

1: **procedure** Actor(B, T)
2: $\theta_0 \leftarrow$ initial parameters
3: $s_0 \leftarrow$ initial state of environment
4: **for** $t = 1 \ldots T$ **do**
5: **if** random value $\hat{r} < \epsilon$ **then**
6: **for** $i = 1 \ldots |\mathbf{A}|$ **do**
7: add $Q(s_{t-1}, \hat{a}_i)$ into $ActionList$
8: **end for**
9: $a_{t-1} \leftarrow \arg\max(ActionList)$
10: **else**
11: $a_{t-1} \leftarrow$ random action
12: take a_{t-1} and acquire (s_t, r_t, d_t) from environment
13: add τ $(s_{t-1}, a_{t-1}, s_t, r_t, d_t)$ into a local buffer
14: **if** size of local buffer $> B$ **then**
15: get B experiences τ from local buffer
16: compute priority p based on τ
17: add (τ, p) into global buffer
18: periodically $\theta_t \leftarrow$ global parameters
19: **end for**

Algorithm 2 Learner

1: **procedure** Learner(T)
2: $\theta_0 \leftarrow$ initial parameters
3: **for** $t = 1 \ldots T$ **do**
4: $\tau, id \leftarrow$ experiences and their indexes sampled from global buffer
5: update priority of global buffer using id, p
6: perform online sequential learning using τ
7: **end for**

4.2. Packet Routing using Reinforcement Learning

In this section, we describe the packet routing method using a reinforcement learning.

4.2.1. System Model. The proposed method considers the following model. Here we assume the network is an undirected graph $G = \{\boldsymbol{V}, \boldsymbol{E}\}$, and N is the number of nodes in the network. Each node has a Q-Network for each destination node. For example, as shown in Figure 6, node v_0 has five Q-networks, Q_{v_0,v_1}, Q_{v_0,v_2}, Q_{v_0,v_3}, Q_{v_0,v_4}, and Q_{v_0,v_5}. Below, we describe the design of the state, action, and reward in our reinforcement learning system.

Algorithm 3 Online Sequential Learning

1: **procedure** SequentialLearning($s_{t-1}, a_{t-1}, r_t, s_t, d_t$)
2: $\hat{t} \leftarrow r_t + (1 - d_t)\gamma max_{a \in A} Q_{\theta_2}(s_t)$
3: **if** initial learn **then**
4: $\boldsymbol{P}_0 \leftarrow (\boldsymbol{H}_0^\top \boldsymbol{H}_0)^{-1}$
5: $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{P}_0 \boldsymbol{H}_0^\top \hat{\boldsymbol{t}}_0$
6: update OS-ELM QN with parameter $\boldsymbol{\beta}_0$
7: $i \leftarrow 0$
8: **else**
9: $\boldsymbol{P}_{i-1} \leftarrow \boldsymbol{P}_{i-1}/\lambda^2$
10: $\boldsymbol{P}_i \leftarrow \boldsymbol{P}_{i-1} - \boldsymbol{P}_{i-1}\boldsymbol{H}_i^\top(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^\top)^{-1}\boldsymbol{H}_i\boldsymbol{P}_{i-1}$
11: $\boldsymbol{\beta}_i \leftarrow \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i\boldsymbol{H}_i^\top(\hat{\boldsymbol{t}} - \boldsymbol{H}_i\boldsymbol{\beta}_{i-1})$
12: update OS-ELM QN with parameter $\boldsymbol{\beta}_i$
13: $i \leftarrow i + 1$

The state s is an N-dimensional vector that ranks the network nodes based on their buffer occupancy. For example, in Figure 6, the number of packets held by the six nodes are 10, 15, 5, 30, 0, and 10, respectively. In this case, the input ranked vector is $[3, 5, 2, 6, 1, 4]$. The reason for using the ranked data is to stabilize the training. This is based on prior research in which the training was unstable when the actual packet counts were used as input data. The state is updated periodically as we describe below.

The action space a of each Q-Network is defined as the next hop nodes that can take the shortest path for a given destination node. For example, in Figure 6, the action space of Q_{v_0,v_5} is $[v_1, v_3]$ because node v_1 or v_3 can be selected as a next hop when node v_0 transfers packets to node v_5.

The reward r is the negative signed latency of each packet to the destination node. Figure 7 shows a packet routing example. Here, we assume a packet is forwarded from node v_0 to node v_4. Q_{v_0,v_4} forwards the packet p_0 from node v_0 to node v_1 at cycle 0. Then, node v_1 receives p_0 at cycle 4, and Q_{v_1,v_4} forwards p_0 to node v_4 at the same cycle. At last, node v_4 receives p_0 at cycle 6. These latencies and a routing path are stored in the packet header of p_0. As a result, Q_{v_0,v_4} will get the reward -6, and Q_{v_1,v_4} will get the reward -2. The method of acquiring the rewards is described below.

4.2.2. Learning Flow. Figure 5 shows the learning flow in this paper. The state and experience are updated at a prespecified interval.

Update state: The state s is updated periodically. For example, let us assume that the update interval is 50 cycles. In Figure 5, the state s_0 is updated with the latest ranked buffer occupancies at 0 cycle. Then, to infer a next hop router, every Q-Network in every node uses the same state s_0 during 0 to 49 cycles. After that, the state is updated to s_1 with the latest buffer occupancies at 50 cycle, and every Q-Network uses the same state s_1 during 50 to 99 cycles. The above steps are repeated until one episode is completed.

Get experiences: The latencies data at intermediate hops are recorded in the packets during their flight to the destination. In Figure 7, p_0 has two latencies when it arrives at node v_4 from node v_0. That is, Q_{v_0,v_4} took 6 cycles to forward p_0 and Q_{v_1,v_4} took 2 cycles under s_0. At the same time as the states are updated, experiences are also sent to

the nodes along the routing path. In this case, Q_{v_0,v_4} gets an experience $\{s_0, v_1, -6, s_1, 0\}$ and Q_{v_1,v_4} gets an experience $\{s_0, v_4, -2, s_1, 0\}$ at 50 cycle.

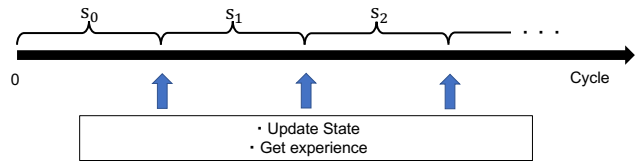

Figure 5. Learning flow where time goes by from left to right.

5. Evaluations

5.1. Simulation Environment

All the programs are executed on a computer with an Intel Core i7-10700 CPU, 32GB RAM, and NVIDIA GTX 3060 Ti GPU. The operating system is Ubuntu 20.04.

We build a network simulator with Python 3.8.10. We use the Python library Gym 0.18.3 [13] and NetworkX 2.6.3 [14] to build the environment. To build a neural network, we use Tensorflow 2.9.1 and NumPy 1.23.1. The network topology of this simulation is a 4x4 mesh topology. The reason for using this small topology is that we consider it is sufficient to validate the proposed method. All the nodes and edges in the network have the same link bandwidth. Each node can forward a single packet in each cycle under a round-robin scheduling; thus in total 16 packets are forwarded in each cycle in the network. Each node has an infinite FIFO (First-In First-Out) queue as a packet buffer. The latency for the packet transfer is 2 cycles per a hop. All the generated packets have the same size. A certain amount of packets are injected at randomly-selected (discrete uniform distribution) nodes each cycle. One episode is finished when all the packets have been delivered completely. A destination node of packets is selected randomly. The update interval we described in Section 4.2.2 is set to 50 cycles.

5.2. Learning Performance

In this section, we compare the performance of DQN and OS-ELM QN with the three different sampling techniques (a), (b), and (c) in Figure 3. That is, the following five methods are compared.

1) DQN (Replay buffer)
2) DQN (Prioritized replay buffer)
3) OS-ELM QN (Random update)
4) OS-ELM QN (Replay buffer)
5) OS-ELM QN (Prioritized replay buffer)

In this experiment, only a single agent is used in these methods. We set the batch size to 64. The interval of Q-learning is 50 cycles. The synchronization interval between the target Q-Network and the main Q-Network as mentioned in Section 2.2 is set to 1000 cycles. The random update probability P is 0.5. The sampling probability parameter α is 0.3. Table 1 shows the neural network models of DQN and OS-ELM QN. In this experiment, 8000 packets are injected into the network in one episode. This means 8 packets are injected per 1 cycle until 1000 cycles.

Figure 8 shows the total rewards of all the nodes in the network. First, we compare (3) Random update method with the replay buffer methods. OS-ELM QN with an experience

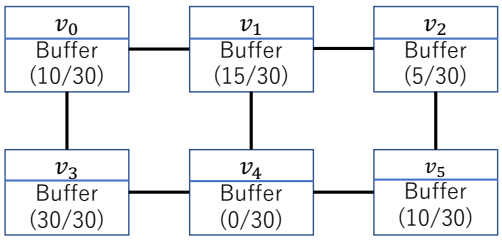

Figure 6. Network example of six nodes, where values under "Buffer" represent (occupied capacity / total capacity).

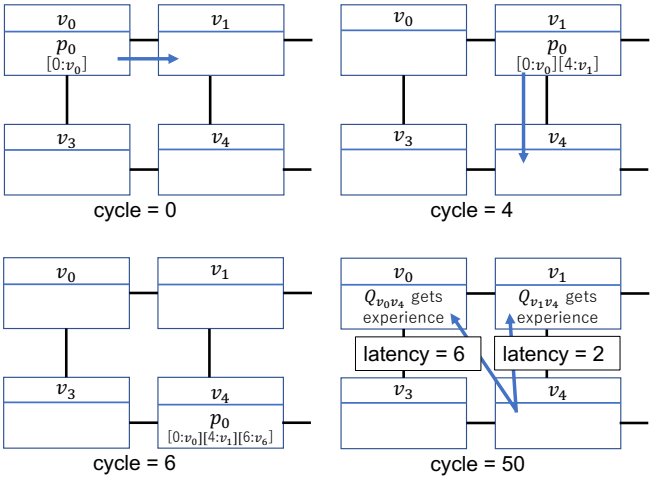

Figure 7. Routing example. $p_0[x{:}v_i]$ means p_0 reached v_i at x cycle.

replay buffer shows a better learning performance than the random update method, which means the experience replay buffer technique contributes significantly to the performance of OS-ELM QN.

We then focus on performance between DQN-based methods ((1) and (2)) and OS-ELM-based methods ((4) and (5)). The OS-ELM QN methods can be trained more efficiently than the DQN methods in the first 20 episodes. This is because the OS-ELM QN methods can optimize parameters in one-shot, while the DQN methods optimize

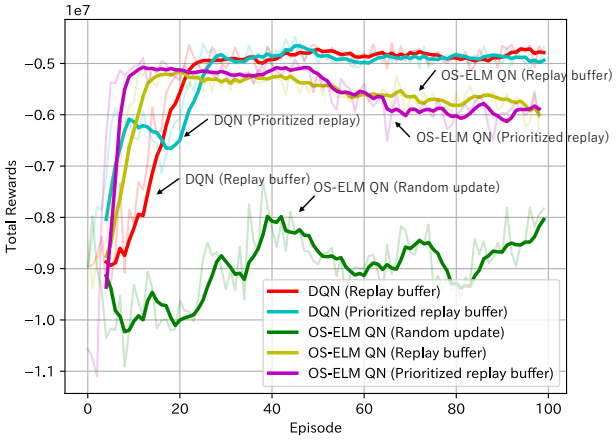

Figure 8. Total rewards vs. training episodes. X-axis shows the number of episodes. Y-axis shows the total rewards of all nodes. Light-colored lines show measured rewards. Heavy-colored lines show the 5-moving average.

TABLE 1. NETWORK MODELS OF DQN AND OS-ELM QN

	Input layer	Hidden layers		Output layer
DQN	18	64	64	$deg(v_n)$
OS-ELM QN	18	64		1

the parameters gradually. However, in subsequent episodes, the DQN methods acquire higher average reward values than the OS-ELM QN methods. This is because the OS-ELM QN methods use a 3-layer neural network and only β can be optimized.

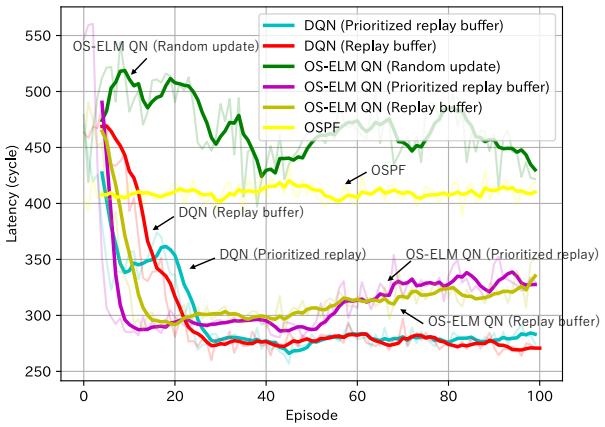

Figure 9. Latency comparison. Y-axis means the total number of cycles taken to transfer all packets.

Next, we consider the performance between the replay buffer technique ((1) and (4)) and the prioritized replay buffer technique ((2) and (5)). These two techniques show some differences in the early episodes of the training process for both the DQN and the OS-ELM QN. The total reward values increase quickly for the prioritized replay buffer methods in the first 10 episodes. This result indicates that the prioritized replay technique can optimize the parameters quickly. In the end, these two methods are converged to the stationary state.

Finally, we evaluate these methods in terms of the packet transfer latency. Figure 9 shows the relationship between the packet latency and training length in episodes. In addition to the reinforcement learning methods ((1) to (5)), we evaluate the latency of OSPF in the same environment for comparison. Comparing the reinforcement learning methods with OSPF, the latency in the reinforcement learning methods decrease significantly. We set the negative number of the latency as the reward value as proposed in Section 4.2.1. Therefore, the total latencies of the reinforcement learning methods (except (3) Random update) decrease as the number of episodes increases. We will discuss the latency more detail in Section 5.4.

5.3. Learning Time

In this section, we evaluate the learning time of OS-ELM QN and DQN. Table 2 shows the inference and learning times of the DQN and OS-ELM QN methods in 100 episodes.

The values in the table are in seconds. The running time of the network simulator and experience sampling are not included. The comparison result shows that the OS-ELM QN method is about twice as fast as the DQN method. As we mentioned in Section 5.2, this is because OS-ELM QN does not use the backpropagation algorithm but one-hot optimization. As a result, OS-ELM QN can learn faster.

TABLE 2. COMPARISON OF LEARNING AND INFERENCE TIME

	Inference time (100 ep)	Learning time (100 ep)
DQN	1192.4	2999.8
OS-ELM QN	1020.5	1539.9
DQN / OS-ELM QN	1.168	1.948

5.4. Latency in Different Injection Rates

We evaluate the packet transfer latency under different packet injection rates. Figure 10 shows the latencies when the packet injection rate is varied from 1 to 10 by 1 in the cases of (2) DQN, (5) OS-ELM QN, and OSPF.

The DQN and OS-ELM QN methods use their models that have learned 100 episodes in Section 5.2. As we can see, there is little difference in the latency when the injection rate is less than 4. However, after that, the two reinforcement learning methods achieve lower latencies than OSPF. These results demonstrate that the reinforcement learning methods choose a routing path intelligently when the packet injection rate is increased. Comparing the two reinforcement learning methods, the DQN method shows a better performance than the OS-ELM QN method. This is due to the difference in the final reward acquired in Figure 8.

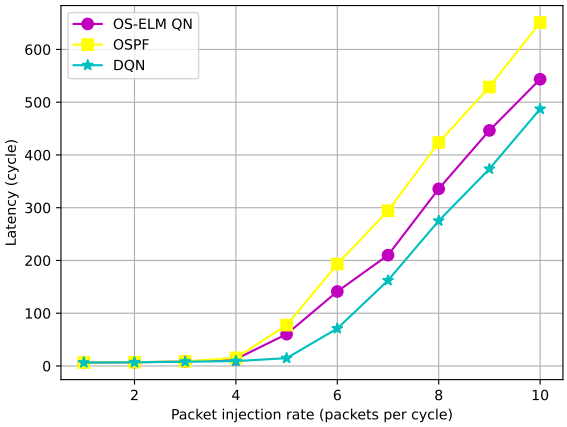

Figure 10. Packet transfer latency with different injection rates

5.5. Performance of Multi-agent Learning

Finally, we evaluate the performance of the multi-agent learning method. Figure 11 shows the total reward versus the training time in the cases of 1, 4, and 8 agents. The executing time is 120 seconds in total. They are compared to the DQN method. That is, the light-blue dotted line indicates the final total rewards of (2) DQN (Prioritized replay buffer) in Figure 8.

The current implementation of multi-agent learning of the OS-ELM QN methods works well in improving the learning speed but the final rewards still do not reach that of the DQN method. As we mentioned in Section 5.2, this slightly lower final reward may come from the representational ability of the 3-layer neural networks. However,

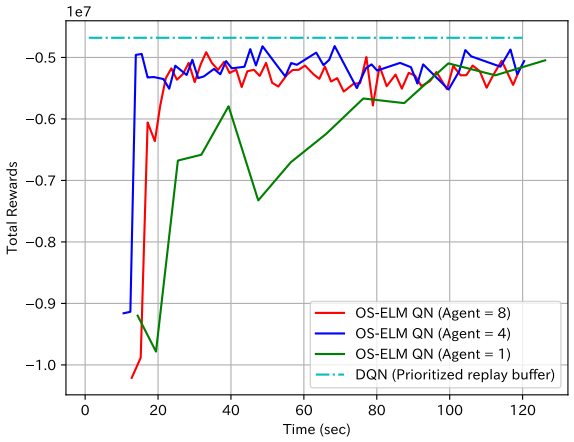

Figure 11. Total rewards per running time

comparing the multi-agent cases (4 and 8 agents) to the single agent case, the multi-agent cases actually improve the learning efficiency per second, which demonstrates the benefit of the multi-agent learning in the OS-ELM QN methods.

6. Discussion

In this section, we discuss the usefulness of OS-ELM QN and its related techniques.

First, we compare OS-ELM QN with DQN in packet routing. In terms of learning performance, while OS-ELM QN performed better than DQN in the early stages of learning, the final reward of DQN was better than OS-ELM QN in Figure 8. These results imply that OS-ELM QN can adapt to traffic patterns quickly, while DQN still provides a stable and high-performance routing. In terms of learning time, OS-ELM QN could be trained approximately two times faster than DQN. OSPF showed a lower transfer efficiency than the reinforcement learning methods, though OSPF does not require a training time of neural networks. These results demonstrated that there is a tradeoff between performance and learning speed.

The three sampling techniques were compared in this paper. In this environment, although a random update technique did not work well, experience replay and prioritized experience replay techniques improved the learning efficiency. Especially, the prioritized experience replay technique achieved higher efficiency than the experience replay technique in the first 10 episodes of Figure 8.

Regarding the reinforcement learning settings (e.g., state, action, reward) introduced in Section 4.2.1, we consider that these settings would be meaningful based on the results from Figures 9 and 10. Further tuning on the reinforcement learning settings is our future work.

A scalability analysis of the proposed algorithm using larger network sizes is our future work. Each node has $(N-1)$ neural networks, so the number of neural networks in a network is quadratically increased when the network size is increased. To alleviate this problem, we are considering combining state (i.e., buffer data) and one-hot encoded destination node as an input state. This modification may affect the accuracy while it can compress the number of neural networks in each node.

7. Conclusions

In this paper, we proposed OS-ELM QN as an OS-ELM-based reinforcement learning method for intelligent packet routing. We evaluated the performance of OS-ELM QN by using a network simulator to measure the packet transfer latency. Compared to an existing work [6], we improved the sampling technique and introduced the multi-agent learning function.

Experimental results showed that DQN achieved a slightly higher packet transfer efficiency than OS-ELM QN, while OS-ELM QN could learn approximately twice as fast as the DQN. Compared to the random update technique, the experience replay techniques contributed greatly to improving the network performance. The multi-agent learning further increased the learning speed of OS-ELM QN. Further performance improvement of the multi-agent learning is our future work.

Acknowledgement

This work was supported by JST CREST Grant Number JPMJCR20F2, Japan.

References

[1] "Cisco Annual Internet Report," https://www.cisco.com/c/ja_jp/solutions/executive-perspectives/annual-internet-report/index.html, (Accessed on 8/20/2022).

[2] J. Moy, "OSPF Version 2," RFC 2178, July 1997. [Online]. Available: https://rfc-editor.org/rfc/rfc2178.txt

[3] "Routing Information Protocol," RFC 1058, June 1988. [Online]. Available: https://rfc-editor.org/rfc/rfc1058.txt

[4] N. Kato, Z. M. Fadlullah, B. Mao, F. Tang, O. Akashi, T. Inoue, and K. Mizutani, "The deep learning vision for heterogeneous network traffic control: Proposal, challenges, and future perspective," *IEEE Wireless Communications*, vol. 24, no. 3, pp. 146–153, 2017.

[5] R. Ding, Y. Xu, F. Gao, X. Shen, and W. Wu, "Deep reinforcement learning for router selection in network with heavy traffic," *IEEE Access*, vol. 7, pp. 37 109–37 120, 2019.

[6] H. Watanabe, M. Tsukada, and H. Matsutani, "An FPGA-Based On-Device Reinforcement Learning Approach using Online Sequential Learning," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'21) Workshops, pp.96-103, May 2021.*, 2021.

[7] N. Liang, G. Huang, P. Saratchandran, and N. Sundararajan, "A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks," *IEEE Transactions on Neural Networks*, vol. 17, no. 6, pp. 1411–1423, Nov 2006.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *NeurIPS Deep Learning Workshop 2013*, 2019, NeurIPS Deep Learning Workshop 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[9] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in *Proceedings of the International Conference on Learning Representations (ICLR'16)*, May 2016.

[10] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed Prioritized Experience Replay," in *Proceedings of the International Conference on Learning Representations (ICLR'18)*, May 2018.

[11] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to Route," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 185–191.

[12] Y. Xiao, J. Liu, J. Wu, and N. Ansari, "Leveraging Deep Reinforcement Learning for Traffic Engineering: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2064–2097, 2021.

[13] "Gym," https://gym.openai.com/, (Accessed on 8/20/2022).

[14] "NetworkX," https://networkx.org/, (Accessed on 8/20/2022).