LONDON'S GLOBAL UNIVERSITY

MSc Dissertation

Final Report

MSc in Telecommunications

Student: Jinhao Wu

Student number: 18110353

Project Title:

Reinforcement learning with routing

Supervisor: Prof. Miguel Rio

University College London

Dept. of Electronic and Electrical Engineering

2018

Abstract

How to implement efficient routing algorithms in complex and variable network environments currently is a hot research topic. Due to the high performance and high adaptability of machine learning, it has gradually come into the sights of many network engineers and scientists. Among the machine learning based algorithms, the routing algorithm based on reinforcement learning is considered to be the most suitable method for the current complex and variable network environment. This paper proposed and tested a deep reinforcement learning routing algorithm based on Q-learning and neural network. This algorithm was implemented in a Python program. In this algorithm, the router selects the forwarding path through the output of the neural network and returns the reward and minimum value function to the previous node, which is used by the previous node to update the neural network. The simulation results showed that the reinforcement learning routing algorithm can implement basic dynamic routing functions, and the algorithm can also achieve different requirements for different types of packets. However, the simulation results also showed that the current reinforcement learning routing algorithm is still in the development stage, and it only can be applied to small scale networks. If the reinforcement learning routing algorithm is applied to larger scale networks, the generalization of network parameters is an important issue that needs to be solved.

Table of Contents

Table of figure

Introduction and Problem Statement

Introduction

In recent years, as the scale of the network grows larger and larger, the network becomes more and more complex and variable, the traditional routing algorithms are gradually difficult to adapt to the current networks. To solve this thorny problem, many network engineers and scientists are committed to design new generation routing algorithms to adapt to the current growing networks. The difficulties of designing a new routing algorithm is how to adapt to the dynamic changes of network links and the complex uncertainty of the network load [1]. Therefore, how to solve these two problems is a challenge for researchers.

The traditional static routing algorithm performs routing according to some certain fixed rules, so these routing algorithms cannot response to the network changes and fluctuations in time to adjust the rules. It is necessary to manually update the rules to adapt to the new network environment. The dynamic routing algorithms can dynamically perform routing based on the current state of the network. The current dynamic routing algorithms can be divided into two categories, namely the distance vector based routing algorithm and the link-state based routing algorithm. The distance vector based routing algorithm calculates the distance vectors of the links in the network, and then performs routing according to the calculation results. These routing protocols only distribute routing information to neighbour nodes, so these protocols are easier to be implemented and maintained. However, for complex networks, their routing performance shows obvious deficiencies due to the limitation of the distance vector. The typical distance vector based routing protocols are IGRP and RIP. In a link-state based routing protocol, each node has a network topology of the entire network, so such algorithms can better adapt to different network environments to make optimal decisions. However, the routers in a link-state based algorithm need to obtain routing information from all other routes regularly by flooding, when the network becomes complex, such algorithms also show significant deficiencies due to the large network burden.

Since machine learning has developed rapidly in recent years, its high adaptability and high performance have quickly attracted the attention of many network engineers. They have designed many dynamic routing algorithms by using the machine learning methods. Among them, the routing algorithms with reinforcement learning were received considerable attention. In the field of the machine learning, the reinforcement learning is different from traditional supervised learning and unsupervised learning that require input of training data sets. It can automatically obtain the training samples through the interactions between agent and the environment. Therefore, the reinforcement learning has high biological relevance and high learning autonomy which is very suitable for the network routing [2]. There are seven important elements in the reinforcement learning, namely environment, agent, state, policy, action, reward and observation. The agent is learner and decision maker in the reinforcement learning, and everything that interacts with the agent is included in the environment. These interactions are continually ongoing, the agent makes actions, and the environment responds to these actions by returning rewards and observations, and then transfers the agent to new

states. A complete agent-environment pair defines a learning task, which is an instance of the reinforcement learning, see figure 1.

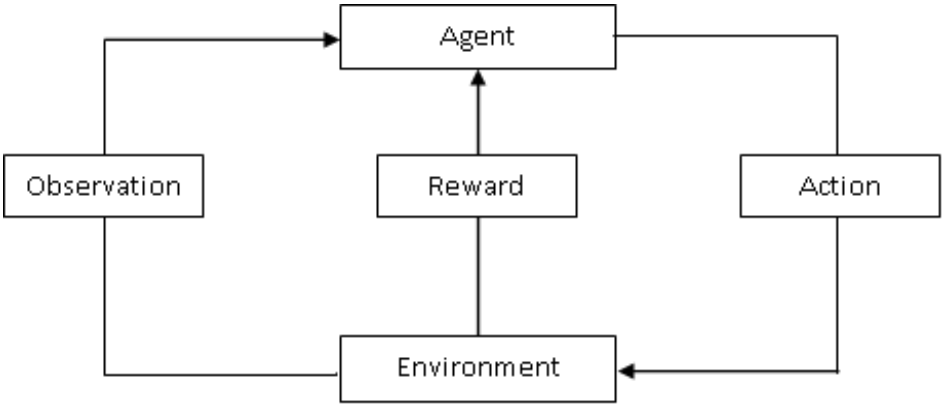


Figure 1: The reinforcement learning process

Problem statement

The aim of this project is to build a virtual network environment in a Python program and realize a reinforcement learning based dynamic routing algorithm. The specific requirements are shown as follow. Firstly, this virtual network environment should contain several virtual routers to form a network topology. These virtual routers should have basic routing functions such as receiving packets, forwarding packets, queuing packets, neighbour discovery and loop prevention. Secondly, the parameters of these virtual routers should be initialized differently, such as different link capacities, different processing performance, different buffer sizes, different number of ports, and so on. Thirdly, the reward and observation returning behaviour of the network environment should also be implemented, including detecting whether the packet is delivered, whether the packet times out, whether the link is congested, and so on. Fourthly, a network traffic model that balances authenticity and simulation friendliness should also be designed. Fifthly, the implementation of the routing algorithm should be based on a more advanced reinforcement learning method which is called deep reinforcement learning, that is, a reinforcement learning model combined with the supervised learning. Finally, the program should also include a statistics module which can record the routing performance of each virtual router and plot a series of routing performance diagrams at the end of the simulation.

Context, Background and Literature Review

Reinforcement learning in routing

The earliest proposed machine learning dynamic routing algorithm can be traced back more than 20 years ago. Boyan and Littman [3] proposed the prototype of the reinforcement Q-learning routing algorithm in 1994. Kennedy and Eberhart [4] proposed the swarm intelligence algorithm in 1995. Rojas [5] proposed the application of neural networks in network routing in 1996. Gen and Cheng [6] proposed the genetic algorithm in 1999, and Zhang and Fromherz [7] implemented the reinforcement Q-learning routing algorithm for practice in 2006. In 2007, Forster [8] summarized and compared several machine learning algorithms that were popular at that time. He stated that the reinforcement learning routing algorithm has smaller network overhead, so the reinforcement learning routing algorithm is more suitable for large scale and energy constrained networks. Yau et al. [9] in 2012 stated that the reinforcement learning can bring context awareness and intelligence to routing algorithms. The context awareness can enable network routers to quickly observe the state of the network environment, while the intelligence can enable network routers to learn the optimal routes by processing the observed network environment information [9]. In other words, the context awareness and the intelligence make the network nodes alive and unlike other traditional dynamic routing algorithms passively acquiring the network information, the routers in the reinforcement learning routing algorithm can actively acquire the network information and update the routing policy more efficiently.

Reinforcement learning fundamental

Model-based reinforcement learning

Most of the proposed reinforcement learning algorithms are based on the Markov decision process (MDP). The MDP is a mathematical model of sequential decision, which was proposed by the Russian mathematician Andrei Markov [10]. It refers to the processes in which an agent learns the most rewarding policy in an environment with Markov property [10]. Markov property indicates that the next state of the system is only related to the current state, but not to the previous state [10]. In an environment with Markov property, the set of all states that the agent experiences each time from the initial state to the terminal state is called a Markov process, which is also known as an MDP episode. The set of a finite number of Markov processes is called a Markov decision process, that is, learning the policy of obtaining the maximum reward from a finite number of Markov processes. A Markov decision process consists of five basic elements {S, A, T, R, P}. S is a set of all states in the environment; A is a set of all actions that the agent can make; T is a state transfer probability matrix, which describes that the probability matrix of agent moving from one state to another state; R is the reward matrix, which describes the set of all rewards given to the agent by the environment according to the actions of the agent; and P is the current policy matrix of the agent, which describes the set of the optimal actions of the agent for the each state in the environment.

Sutton and Barto [11] formally proposed a complete reinforcement learning theory in 1998, and they defined some basic concepts of the reinforcement learning. Firstly, they defined long-term reward, which is the discount sum of all rewards that the agent will get from the current state to the terminal state, see equation 1.

$$G = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \quad (1)$$

Where γ is a discount rate and 0 < γ <1, k is the iteration and when the MDP does not have a terminal state the k can increase to infinite, and r_{t+1} is the reward of the current state.

Secondly, with the definition of the long-term reward, other two important concepts were defined, which are the state-value function $v_{\pi(s)}$ and state-action-value function $q_{\pi(s,a)}$. They respectively describe the expected long-term reward that the agent can obtain from a certain state to the terminal state under the policy π, and the expected long-term reward that the agent can obtain from a certain state to terminal state under the policy π after making a specific action a, see equation 2 and 3.

$$V_\pi(s_t) = E_{s_{t+1}\sim s_{\infty},\pi}[\sum_{k=0}^{\infty}\gamma^k r_{t+1+k}] \quad (2)$$

$$q_\pi(s_t, a) = E_{s_{t+1},a\sim s_{\infty},\pi}[\sum_{k=0}^{\infty}\gamma^k r_{t+1+k}] \quad (3)$$

These two value functions are the objective functions of the reinforcement learning. Just like the weights of the neural network constantly updating in the supervised learning to minimize the objective error function, the goal of the reinforcement learning is to update the policy to minimize (or maximize) these two value functions. The easiest way to update a policy in the reinforcement learning is the iterative method, which requires the implementation of Bellman equations [12]. The Bellman equation was proposed by American mathematician Charlie Bellman, which is an important mathematical tool to solve the problem of dynamic programming. It is also an important theoretical basis for many engineering control theories and economic capital pricing theories [12].

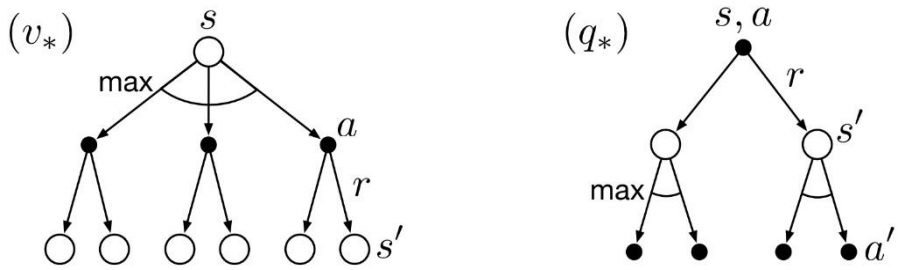


Figure 2: The backup figures of value function vπ(s) and qπ(s,a) [12]

Figure 2 are the backup diagrams of the state-value function and state-action-value function. The former describes the possible actions the agent can make when it locates at a state S, and the subsequential states to which it may be transferred. The latter describes the possible states to which the agent may be transferred when it locates at a state S after making an action a, and the subsequential actions that the agent may make. The Bellman function can be derived from these two backup diagrams. Firstly, the two value functions can be rewritten as follow

$$v_\pi(s) = \sum_{a\in A}\pi(a|s)q_\pi(s,a) \quad (4)$$

$$q_\pi(s,a) = \sum_{s'} P_{ss'}^{a}[r_{ss'} + \gamma v_\pi(s')] \quad (5)$$

where π(a|s) is the conditional probability of the agent that making an action a at the state S under the policy π, $r_{ss'}$ is the reward that the environment giving to the agent when agent transfers from state S to state S', and $P_{ss'}^{a}$ is the probability of the agent transferring from state S to state S' after making an action a.

Substituting equation 5 to equation 4, the state-value function $v_\pi(s)$ can be rewritten as

$$v_\pi(s) = \sum_{a\in A} \pi(a|s) \sum_{s'} P^a_{ss'}[r_{ss'} + \gamma v_\pi(s')] \quad (6)$$

Substituting equation 4 into equation 5 the state-action-value function $q_\pi(s,a)$ can be rewritten as

$$q_\pi(s,a) = \sum_{s'} P^a_{ss'}[r_{ss'} + \gamma \sum_{a\in A} \pi(a|s') q_\pi(s',a)] \quad (7)$$

The equation 6 and 7 are the bellman functions in the reinforcement learning. There is an important assumption in MDP which is the time independency. The time independency refers to the value functions of two same states at different instants of times in an MDP will eventually become same with the continuous evolution of the MDP. This assumption is based on the condition that the state transition process of MDP lasts long enough, and eventually each state transition enters a stable state [12]. With this assumption, the Bellman function can be transferred into the form of matrix operation, see figure 3.

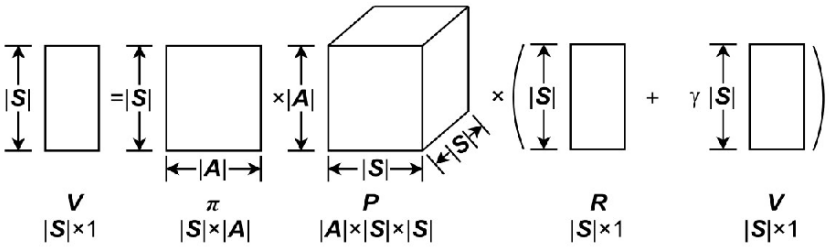


Figure 3: The matrix operation of Bellman function of state-value function [12]

Then the Bellman function of state-value function can be rewritten as

$$V = \Pi P(R + \gamma V) \quad (8)$$

The Bellman function of the state-action-value function can also be rewritten into the from of matrix, then the values of these two value functions can be derived by solving the matrix equations or using iteration method. In general, the iterative methods are easier to be implemented because the matrix equation is always difficult to be solved.

When the value of the state-value function is known, the state-action-value function can be derived, then the policy can be updated as follows

$$\pi(s) = argmax_a q(s,a), a \in A, s \in S \quad (9)$$

The policy of the agent at state s is the action with the maximum state-action-value function at state s. When the agent gets the new policy, the new value function will be derived again and the policy will also be updated again. It can be proved that the policy will become stable to an optimal policy within finite iterations [12]. This learning method is called the policy iteration method, which is the easiest method in the reinforcement learning. In this method, both the state transition probability matrix and the reward matrix which belong to the environmental information are known. The reinforcement learning model with known environmental information is called the model-based reinforcement learning model.

Model-free with Monte-Carlo method

The environmental information in most reinforcement learning instances is unknown, and these instances belong to model-free reinforcement learning model. For model-free reinforcement learning model, the policy iterations cannot be used directly. The Monte-Carlo method is a good approach for the model-free reinforcement learning problems. Because the value function describes the expectation of the long-term reward of a state or a state-action pair, the Monte-Carlo method can be used to estimate the value function. The idea of the Monte-Carlo method is to do trials and use the testing data to estimate the value functions, then the policy iteration method can be applied subsequently.

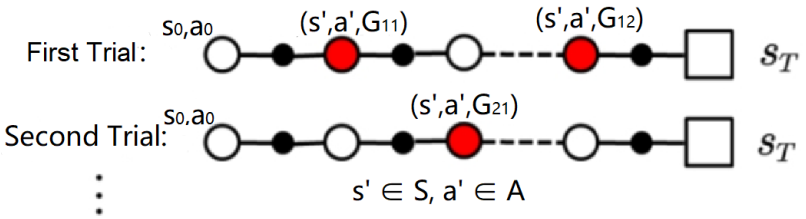


Figure 4: The trials in the Monte-Carlo method [13]

Figure 4 is the trials in the Monte-Carlo method, the agent firstly locates at a random state in the environment and makes an action under the current policy, then the agent will be transferred into a new state by the environment. The agent continuously moves in the environment until the agent enters the terminal state, which is called a trial sequence. When the model gets an enough number of trial sequences, the long-term reward of each state-action pair that appears in the trial sequences can be calculated, then the average value can be calculated as the estimation of the value function of these state-action pairs, see equation 10.

$$q_\pi(s',a') \cong \frac{1}{N(s',a')}\sum_{i=1}^{N(s',a')} G(s',a') \quad a' \in A, s' \in S \tag{10}$$

Where N(s',a') is the N^{th} appearance of the state-action pair (s',a') in the trial sequences.

When the model gets the estimation values of all state-action pairs, the policy can be updated as follow

$$\pi(s) = \begin{cases} \varepsilon + \frac{\varepsilon}{N(A)} \quad , & a = argmax_a q(s,a) \\ 1 - \varepsilon + \frac{\varepsilon}{N(A)} \, , & a \neq argmax_a q(s,a) \end{cases} \quad a \in A, s \in S \tag{11}$$

Where ε is an exploration rate that 0< ε<1. The policy updating in the model-free learning is ε-greedy method rather than the fully greedy method in the model-based learning. This means that for a state the action with the largest state-action-value function in the policy has a larger probability of being selected and other actions of this state have an identical smaller probability of being selected. The purpose of the ε-greedy is that the model needs to be explorative to all state-action pairs, otherwise there will be some state-action pairs that may never be experienced by the agent [13].

The Monte-Carlo method is a good approach, but it is not used commonly in most cases. The Monte-Carlo method is not efficient because it always requires full trial sequences to calculate the average values of state-action-value functions [13].

Model-free with Q-learning

The Q-learning is a more efficient way used in the model-free problems. The Q-learning also does trials, however, the value estimation of the state-action-value function in the Q-learning can be performed simultaneously with trials. The average of the state-action-value function in the Monte-Carlo method can be rewritten into the incremental form as follow

$$\begin{aligned} q_{\pi,k}(s',a') &\cong \frac{1}{k}\sum_{i=1}^{k} G_i(s',a') \\ &= \frac{1}{k}\left[G_k(s',a') + \sum_{j=1}^{k-1} G_j(s',a')\right] \\ &= \frac{1}{k}\left[G_k(s',a') + (k-1)q_{\pi,k-1}(s',a')\right] \\ &= q_{\pi,k-1}(s',a') + \frac{1}{k}\left(G_k(s',a') - q_{\pi,k-1}(s',a')\right) \end{aligned} \tag{12}$$

where k is the k^{th} appearance of the state-action pair (s',a') in the trials. Then 1/k in the equation 12 can be replaced by a constant α, see equation 13.

$$q_{\pi,k}(s',a') = q_{\pi,k-1}(s',a') + \alpha\left(G_k(s',a') - q_{\pi,k-1}(s',a')\right) \tag{13}$$

The equation 13 is the moving average of the state-action-value function. Then the state-action pair (s',a') can be replaced by (s_t,a_t).

$$q'_{\pi}(s_t,a_t) = q_{\pi}(s_t,a_t) + \alpha\left(G(s_t,a_t) - q_{\pi}(s_t,a_t)\right) \tag{14}$$

Next, the long-term reward G(s_t,a_t) in the equation 14 can be divided into two parts, the one-step reward of state-action pair (s_t,a_t) and the maximum state-action-value function of the next state-action pair (s_{t+1},a_{t+1}), see equation 15.

$$q_{\pi}{}'(s_t,a_t) = q_{\pi}(s_t,a_t) + \alpha(r_{t+1} + \gamma * \max_{a_{t+1}} q_{\pi}(s_{t+1},a_{t+1}) - q_{\pi}(s_t,a_t)) \tag{15}$$

The equation 15 is the main idea of Q-learning, which is using the state-action-value function of next state to estimate the value function of the current state. Because the updating of the policy and the evaluation of the value function can be performed simultaneously, the Q-learning is much more efficient than the Monte-Carlo method [14].

Reinforcement learning in routing

Q-routing algorithm

Network routing problems can be easily modeled in the reinforcement learning because the forwarding of packet is only related to the router that the packet is currently located and independent with the routers that the packet has been experienced before, which shows the Markov property in the MDP. Therefore, the network topology can be treated as the environment, the each packet forwarded in the network can be treated as each individual agent in the environment, the routers in the network can be treated as the sates, the forwarding behaviours of the router can be treated as the actions, and the routing table in each router can be treated as the policy. For different types of packet, the reinforcement

learning routing algorithm can have different policies. For example, for UDP packet the reward of each forwarding behaviour of the router can be the delay between two routers, and the value function then can be described as the end-to-end delay from the source router to the destination router, the policy here is the minimum delay routing algorithm. For TCP packet, the reward of each forwarding behaviour of the router can be the degree of congestion of the forwarding link, and then the value function can be described as the end-to-end packet loss rate between source router to destination router, the policy here is the minimum packet loss rate routing algorithm.

Boyan and Littman [3] firstly proposed the reinforcement learning routing algorithm in 1994 which is based on the Q-learning, so their routing algorithm is called Q-routing. See equation 16, which is the main idea of Q-routing

$$Q^i_{t+1\, j\in a^i_t}(s^i_t, j) = Q^i_t(s^i_t, j) + \alpha\left[r^i_{t+1}(i,j) + \min_{k\in a^j_{t+1}} Q^j_{t+1}(s^j_{t+1}, k) - Q^i_t(s^i_t, j)\right] \qquad (16)$$

Where t is the current instant of time, i,j and k are the serial numbers of the routers in the network, a^i_t and a^j_{t+1} are the sets of the next-hop routers of ith and jth routers, s^i_t and s^j_{t+1} are the destination of the packet which is unchanged all the time, the superscript is used to distinguish the located router, and the subscript is used to distinguish the instant of time.

The principle of the Q-routing is that the routers in the network store the value functions in their local memories as the routing tables, and continuously update them by forwarding and receiving packets. For example, see figure 5, which is a network topology with the minimum delay Q-routing algorithm.

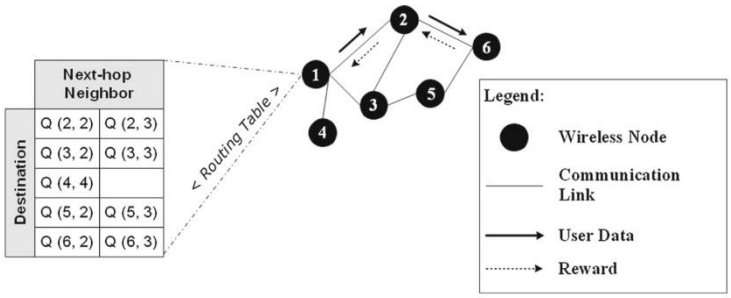


Figure 5: A network topology with Q-routing algorithm [15]

Just as shown in the figure 5, the router 1 has a routing table which contains the state-action-value function of each destination-next-hop pair. Assuming that router 1 receives a packet which needs to be sent to router 6, the router 1 will firstly look up the destination set in the routing table, then it compares all the state-action-value functions of the destination router 6. The router 1 will forward the packet to the next-hop router which has a smaller Q value. If Q(6,2) is smaller than Q(6,3), the router will forward the packet to router 2. When router 2 receives the packet, it will return the delay between router 1 and 2 which contains the transmission delay and the queuing delay. The router 2 will also return its minimum Q value of destination router 6. When router 1 receives the returned reward (delay) and the Q value it will update the value of Q(6,2) by using the equation 16.

Deep Q-routing

The Q-routing has a deficiency, which is the inefficiency in the large-scale network. When the scale of the network becomes large the size of the Q-routing table will increase exponentially, which will reduce the packet processing efficiency of the router. The neural network in the supervised learning can solve this problem. The reinforcement learning model accompanied with the supervised learning is called the deep reinforcement learning model. The Q-routing can become the deep Q-routing by replacing the Q table in each router into neural network, see figure 6.

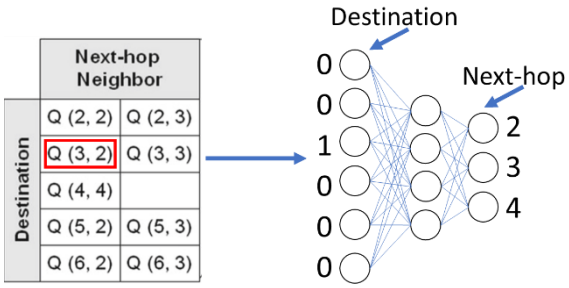


Figure 6: The Q table and Q neural network [15]

The input of the neural network is the one-hot form of the destination, the outputs are the end-to-end delays to all other routers. This kind of way can greatly reduce the time spent in the check-up process.

The deep reinforcement learning not only introduces neural networks, it also has many other features. The DeepMind proposed the deep Q Network (DQN) in 2015, which is the first formal definition of the deep reinforcement learning model [16]. In addition to the introduction of the neural networks, DeepMind also introduced other two features which are experience replay and target network into the reinforcement learning model [16]. After DeepMind, Hasselt [17] and Schaul [18] proposed their improved version of the DQN, respectively. They introduced the new features double Q learning and priority replay into the DQN.

The experience replay refers to storing the returned reward in a buffer and updating the policy by uniformly selecting a batch of samples in the buffer. The experience replay is similar to the mini-batch gradient descent method in the supervised learning. If a supervised learning model uses the stochastic gradient descent method the objective error function will be difficult to converge to a minima. The mini-batch gradient descent method can make the objective error function converge to the minima more stably and smoothly. The function of the experience replay is similar to the mini-batch gradient descent method, if the reinforcement learning model updates the policy every time when it receives a reward, the policy will be difficult to converge to optimal because there may be correlation between consecutive samples [16]. The random selection in the replay buffer can break the correlation among consecutive samples [16]. In the deep Q-routing algorithm, routers can store the returned delays and value functions in its buffer and update the routing policy when the buffer is full.

The priority replay is an improved version of experience replay, it selects the samples in the buffer with weighted probabilities rather than using uniform selection. The principle of the priority replay is that before the reward is stored in the buffer, the agent will calculate a Q-

error and then store the reward accompanied by the Q-error [18]. The Q error is shown as below

$$p_i = |\delta_i| = R_{t+1} + \gamma \max_{a_{t+1}} q_\pi(S_{t+1}, a_{t+1}) - q_\pi(S_t, a_t) \quad (17)$$

When the agent starts to select samples in the buffer, it will firstly calculate the weight of each reward in the buffer, see equation 18

$$P_i = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (18)$$

Where α is a constant parameter to equalize the probability and 0< α <1, and k is the number of the rewards in the buffer. Because the weighted selection of the reward will make the evaluation of the state-action-value function become biased, the selected reward needs to multiply an importance sampling factor to solve this problem, see equation 19

$$w_i = (\frac{1}{k} \cdot \frac{1}{P(i)})^\beta \quad (19)$$

Where β is a constant parameter to equalize the factor and 0< β <1.

The priority replay can let routers update the routing entries with larger errors more frequently, which can improve the efficiency of the router to learning and optimize the routing table.

The target network in the DQN refers to an additional network that has the same structure with the main network but the weight updating of it is lagging behind the main network. The target network in the Q-routing is used to replace the main network to return the minimum state-action-value function to the previous router. The neural network in the router is continuously updated so the state-action-value function of the neighbour nodes of a router will change frequently, which may increase the variance of the samples and influence the learning effect [16]. The purpose of the target network is to reduce the variance of the samples. At the beginning, the target network duplicates the weights of the main network and then the target network will be frozen for a fixed period of time. After that, the target network will be unfrozen and duplicate the main network's weights again and then it will be frozen again. Because the weights of the target network do not change frequently, the returned rewards and value functions can have small variances.

The double Q-learning refers to splitting the action evaluation and the action selection in the Q-learning. In the Q-learning, the agent uses the returned reward and next state's maximum (or minimum) state-action-value function to update the current state's state-action-value function, see equation 15. However, the max (or min) operation here may make an over (or under) estimation to the next state's state-action-value function [17]. The double Q-learning solves this problem by splitting action evaluation and action selection by using different state-action-value functions. The returned reward and value function are called Q-target in the Q-learning, see equation 20.

$$Y_t^Q = R_{t+1} + \gamma \max_{a'} q_\pi(S_{t+1}, a') \quad (20)$$

The action selection refers to finding the action a' which has the largest state-action-value function in the next state S_{t+1}, and the action evaluation refers to construct the returned Q-

target by using the one-step reward and the state-action-value function of action a'. It can be seen from equation 20 that the action selection and evaluation use the same value function. In the double Q-learning the action selection and evaluation use different value function, see equation 21.

$$Y_t^{Double\ Q} = R_{t+1} + \gamma q_{\pi'}(s_{t+1}, \max_{a'} q_{\pi}(S_{t+1}, a')) \quad (16)$$

Because there are two neural networks in the DQN, the target network can be used to separate the action selection and evaluation. The weights of the target network are different from the main network, so it can reduce the over (or under) estimating effect of the value function.

Results

Methodology/Design Description

This project used two Python files to implement a deep reinforcement learning routing model, which are **env.py** and **Agent.py**. In the **env.py**, a network environment was constructed, including network topology, routers, forwarding rules, receiving rules, queuing rules, and the packet generation rules. In the **Agent.py**, the algorithm of the reinforcement learning was implemented in this file, including the establishment of the neural network, the weight update of the main network, the return of the value function of the target network, the interaction between the target network and the main network, and so on. The complete codes for these two files will be shown in the appendices.

Network environment in env.py

In the **env.py**, the network environment is defined in a class **Network**, and the packet is also defined in a class **Packet**. In the class **Packet**, the source node of the packet (**source)**, the destination node (**dest)**, the birth time of the packet (**birth)**, the total number of hops (**hops)**, the time of entering the router queue (**qtime**), the size of the packet (**size**), the type of packet (**type**), the next hop of the packet (**next**), the one-step delay carried by the packet (**reward**), the port number of the next-hop node that receives the returned Q-target (**prereward**), and the flag of returned Q-target (**back**) are defined, see figure 7.

```
class Packet:
    def __init__(self, source, dest, btime, reward, prereward,backQ, preforward_port,predest,back):
        self.dest = dest
        self.source = source
        self.node = source
        self.birth = btime
        self.hops = []
        self.qtime = timeit.default_timer()
        self.size = int(x.rvs(1))
        self.type = np.random.choice(['TCP', 'UDP'], p = np.array([0.5, 0.5]).ravel())
        self.next = None
        self.reward = reward
        self.prereward = prereward
        self.backQ = backQ
        self.forward_port = None
        self.preforward_port = preforward_port
        self.predest = predest
        self.back = back
        self.delay = 0
        self.delay_1 = 0
```

Figure 7: The attribute definitions in the class Packet

The size of the packet is subject to a truncated normal distribution, and the type of the packet is subject to a binomial distribution. In the class **Network,** there are more attributes defined. Among them, the important attributes are the packet receiving queue (**packet_queue**) in each router, the packet forwarding queue in each router (**packet_forward_queue)**, the experience replay pool in each router (**replay**), the learning sample pool in each router (**sample**), the total number of nodes (**n_nodes**), the total number of links (**n_links**), the correspondence between ports (**port)**, link capacity (**linkc**), router maximum buffer (**buffer**), packet delay statistics (**UDP_delay** and **TCP_delay**), and packet loss rate statistics (**UDP_loss** and **TCP_delay**). They basically define a complete topology of a network environment and also the basic properties of network router. The complete and detailed attributes defined in the class **Network** are shown in the figure 8.

```
class Network():
    def __init__(self):
        self.done = False                           # simulation done
        self.packet_queue = defaultdict(Queue)      # packet receiving queue in a router
        self.packet_queue_size = defaultdict(Queue) # size of packet receiving queue in a router
        self.packet_forward_queue = defaultdict(dict) # packet forwarding queue in a router
        self.n_receive_queue = defaultdict(Queue)   # number of packets in the packet receiving queue in a router
        self.replay = defaultdict(list)             # experience replay pool in a router
        self.sample = defaultdict(list)             # traning sample pool
        self.n_forward_queue = defaultdict(Queue)   # number of packets in the packet forwarding queue in a router
        self.port = defaultdict(dict)               # correspondence of ports
        self.n_nodes = 0                            # number of nodes
        self.n_edges = 0                            # number of links
        self.links = defaultdict(dict)              # correspondence of nodes
        self.linkc = defaultdict(dict)              # capacities of links
        self.buffer = defaultdict(int)              # size of the buffer in a router
        self.t_routing_time = 0.0                   # total routing time
        self.succeed_packets = 0                    # total number of successfully transferred packets
        self.t_hops = 0                             # total number of hops
        self.process = 0.2                          # packet processing time in a router
        self.active_packets = 0                     # total number of active pakcets
        self.send_fail = 0                          # total number of transferred failed packets
        self.arrivalmean = defaultdict(int)         # Lambda in the Poisson process
        self.agent = defaultdict()                  # routing algorithms in a router
        self.iteration = 0                          # simulation iteration
        self.latency_UDP = defaultdict(dict)        # statistics of delay of UDP packet
        self.latency_TCP = defaultdict(dict)        # statistics of delay of TCP packet
        self.packet_loss_TCP = defaultdict(dict)    # statistics of packet loss rate of UDP packet
        self.packet_loss_UDP = defaultdict(dict)    # statistics of packet loss rate of TCP packet
        self.back = defaultdict(dict)               # flag of returning Q-target
        self.recovery_rate = defaultdict(dict)      # link recovery rate
        self.last = defaultdict(dict)               # time when the last packet was sent
        self.neighbour = defaultdict(dict)          # neighbour relations between routers
        self.tcp_loss = defaultdict(bool)           # tcp loss flag for time out
        self.tcp_loss_1 = defaultdict(bool)         # tcp loss flag for full queue
        self.tcp_flag = defaultdict(bool)           # tcp transfer flag for suboptimal route
        self.tcp_flag_1 = defaultdict(bool)         # another tcp transfer flag for suboptimal route
        self.tcp_flag_count = defaultdict(int)      # number of tcp packet which is transferred via suboptimal route
```

Figure 8: The attribute definitions in the class of Network

After the attribute definitions in the class **Network**, the initialization of these attributes was performed subsequently in **rest**. These attributes were initialized by a CSV file which contains the information of the network environment. See figure 9, which is the CSV file and the network topology used in this project.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	1	N	3	4	C	2.60E+03	2.00E+03	1.60E+03	B	2.54E+04	M	6.00E-01		
2	2	N	3	4	C	2.60E+03	3.00E+03	3.00E+03	B	2.67E+04	M	6.00E-01		
3	3	N	1	2	5	C	2.20E+03	1.80E+03	1.20E+03	1.40E+03	B	2.67E+04	M	6.00E-01
4	4	N	1	2	5	C	3.20E+03	1.00E+03	2.80E+03	1.00E+03	B	2.68E+04	M	6.00E-01
5	5	N	3	4	C	2.80E+03	2.00E+03	1.50E+03	B	2.52E+04	M	6.00E-01		

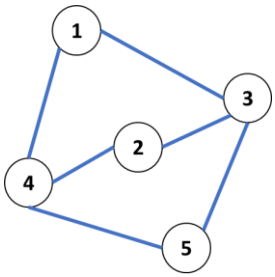


Figure 9: The network parameter CSV file and the network topology

In this CSV file, each line contains the information of a router, so there are 5 routers in the network environment. The first digit of each line represents the serial number of the router in the network. The digits after the capital letter N represent the serial numbers of the neighbour routers. The numbers after the capital letter C represent the capacities of the link connecting to the neighbour routers. The number after the capital letter B represents the size of the buffer of the router. The number after the capital letter M represents the time interval at which the router generates packets.

After initializing the attributes of the network environment, the class **Network** defined three important functions subsequently, which are **_receivequeue**, **_forwardqueue**, and **_get_new_packet**. **_receivequeue** defines the rules for the router to receive the packet, **_forwardqueue** defines the rules for the router to forward the packet, and **_get_new_packet** defines the rules for the router to generate new packets. In the **_get_new_packet** function, in order to simplify the complexity of the model, new packets are generated at fixed time intervals, the lengths of the time interval are obtained from the initialization CSV file in the previous step. In order to balance the load of each router, each router always generates new packets which are sent to other routers evenly, see the red box in figure 10.

```
def _get_new_packet(self, node):
    callmean = self.arrivalmean[node]
    j = 1
    n_new_list = []
    p_list = []
    for i in range(1, self.n_nodes+1):
        if i != node:
            p_list.append(1/(self.n_nodes-1))
        else:
            p_list.append(0)
    p = np.array(p_list)
    dest_list = []
    dest_temp = []
    for i in range(1, self.n_nodes+1):
        dest_list.append(i)
    for i in range(1, self.n_nodes + 1):
        if i != node:
            n_new_list.append(int((self.iteration / (self.n_nodes - 1))))
            dest_temp.append(i)
        else:
            n_new_list.append(0)
    dest_temp_1 = dest_temp.copy()
    clear_all = open(str(node) + '_queue.csv', 'w+')
    clear_all.truncate()
    clear_all.close()
    while True:
        time.sleep(callmean)
        source = node
        while True:
            dest = np.random.choice(dest_list, p=p.ravel())
            if node != dest and n_new_list[dest - 1] != 0 and dest in dest_temp_1
                n_new_list[dest - 1] -= 1
                dest_temp_1.remove(dest)
                if not dest_temp_1:
                    dest_temp_1 = dest_temp.copy()
                break
            #print('node new ', node)
            if sum(n_new_list) == 0:
                break
```

Figure 10: Part of the code of the **_get_new_packet** function

Additionally, the **_get_new_packet** function sends the packets to the receiving queue of the router, and when the router's receiving queue is full this function will record the loss the packet. If the type of the packet is TCP, this function will also set up a flag to tell **_receivequeue** and **_forwardqueue** to send TCP packets along suboptimal route and drop a set of UDP packets to reduce TCP packet loss rate, see figure 11.

```
if current_buffer + packet.size > node_buffer:
    self.send_fail += 1
    self.packet_queue_size[node].put(current_buffer)
    if packet.type == 'TCP':
        self.packet_loss_TCP[node][dest].append((node,0,'s_f'))
        self.tcp_loss_1[node] = True
        self.tcp_flag_1[node] = True
    else:
        self.packet_loss_UDP[node][dest].append((node,0,'s_f'))
    #print('new', node, i, arrival * 10, 'new packet fail')
else:
    self.active_packets += 1
    self.packet_queue[node].put(packet)
    self.packet_queue_size[node].put(packet.size + current_buffer)
```

Figure 11: Part of the code of the **_get_new_packet** function

The **_receivequeue** function reads the packets in the receiving queue one by one in the first-in-first-out order, and then processes the packets through four if statement to determine the type of the packet, see figure 12.

```
if (elapse_time > 15 and packet.type == 'UDP') or (elapse_time > 20 and packet.type == 'TCP'):
if packet.next != dest and packet.next != None:
if packet.next == dest:
if packet.next == None:
```

Figure 12: The four if-condition judgments in **_receivequeue**

The first if statement determines whether the packet has timed out when it reaches the router. Because UDP packets are more sensitive to latency than TCP packets, the timeout threshold for UDP packets is smaller than TCP packets. The second if statement determines whether the packet is an undelivered packet that needs to be forwarded. The third if statement determines whether the packet is a delivered packet, and the fourth if statement determines whether the packet is a new packet generated by the router. Additionally, the first if statement will drop the packet, the second and the forth will find the next-hop of the packet and forward it to the forwarding queue of the corresponding forwarding port.

In this model, the Q-target is returned in the form of piggyback, which means that the router will send a Q-target message to the next-hop router accompanied with the forwarding packet. This Q-target message corresponds to the packet sent by the next-hop router to the local router previously. This piggyback mechanism does not require additional packet to transfer Q-target, which can save the link capacity and reduce the load of the router. Therefore, after the second and fourth if statements, there is a module that reads the piggyback Q-target information in the packet, and there is also a module that adds the piggyback Q-target information of next-hop router into the packet, see figure 13.

```
if packet.back:
    f_port = self.port[prenode][preforward_port]
    backQ = packet.backQ
    Q_actual = prereward + 0.9 * backQ
    TD_target = Q_actual - Q_estimate_list[f_port-1]
    delta_r = TD_target/(timeit.default_timer() -self.last[node][f_port])
    self.recovery_rate[node][predest][preforward_port] = delta_r
    weight = abs(TD_target)
    sample = [predest, f_port, prereward, Q_estimate_list, backQ, weight]
    self.replay[node].append(sample)

if not self.back[node][next_hop].empty():
    back_infor = self.back[node][next_hop].get()
    packet.prereward = back_infor[0]
    packet.backQ = back_infor[1]
    packet.preforward_port = back_infor[2]
    packet.predest = back_infor[3]
    packet.back = True
else:
    packet.prereward = None
    packet.backQ = None
    packet.preforward_port = None
    packet.predest = None
    packet.back = False
```

Figure 13: The reading and adding modules of piggyback Q-target

In addition to the above four if statements, there are other two if statements in the **_receivequeue** that control the functions of the router. The first if statement controls the learning of the router. When the router's experience replay pool is full, the router will update the neural network. Just like the priority replay described above, the router will first read the Q-error of each sample in the pool and then calculate the probability of being selected of each sample. Finally, the router will select samples from the experience replay pool based on these probabilities and update the network. In addition, the target network will duplicate the weights of the main network after every time the main network is updated twice, see figure 14.

```
if len(self.replay[node]) == 20:
    #print(node,'learn')
    self.sample[node] = []
    importance_sampling_factor = []
    pr_list = []
    TD_list = []
    for i in range(20):
        TD_list.append(self.replay[node][i][5])
    TD_sum = sum(TD_list)
    for i in range(20):
        pr_list.append(self.replay[node][i][5]/TD_sum)
    p = np.array(pr_list)
    for i in range(20):
        importance_sampling_factor.append((0.2/pr_list[i])**0.01)
        self.replay[node][i].append(importance_sampling_factor[i])
    for i in range(100):
        alt_sample_list = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
        chosed_sample_index = np.random.choice(alt_sample_list,p=p.ravel())
        self.sample[node].append(self.replay[node][chosed_sample_index])
    self.agent[node].learn(self.sample[node])
    self.replay[node] = []
    j += 1
    if j % 2 == 0 and j != 0:
        #print(node,'update')
        beta = self.agent[node].update_network(beta)
        print('node ',node, ' NN output')
        self.agent[node].show_routing_table()
    if j % 2 == 0:
        epsilon = self.agent[node].update_epsilon(epsilon)
        print(node,epsilon)
```

Figure 14: The learning module in **_receivequeue**

The second if statement controls the behaviour of the router when the TCP packet is lost. When the TCP packet is lost, **_get_new_packet** will set up a TCP loss flag. This flag will tell the router to drop some UDP packets and forward TCP packets along the suboptimal route to save link capacity and reduce router load. **_receivequeue** will detect this flag by an if statement and drop the UDP packets with a probability, see figure 15.

```
if self.tcp_flag_1[node] and packet.type == 'UDP':
    sacrifice = np.random.choice([True, False], p=np.array([sacrifice_p, 1-sacrifice_p]).ravel())
    if sacrifice:
        self.send_fail += 1
        current_node_current_buffer = self.packet_queue_size[node].get()
        current_node_changed_buffer = current_node_current_buffer - packet.size
        self.packet_queue_size[node].put(current_node_changed_buffer)
        self.packet_loss_UDP[packet.source][dest].append((node, 0, 'u_s'))
        continue
```

Figure 15: Dropping UDP packets in **_receivequeue**

After the packets are processed by the **_receivequeue** function in the receiving queue, they will be sent to the forwarding queue of the corresponding forwarding port. The **_forwardqueue** function is used to handle these packets subsequently. **_forwardqueue** first calculates the required transmission time based on the size of the packet and the capacity of the forwarding link. This function then pauses for the transmission time to simulate the transmission. Finally, this function will send packets to an environment processing function namely **_step**, see figure 16.

```
def _forwardqueue(self, node, port):
    while True:
        packet = self.packet_forward_queue[node][port].get()
        forward_port = packet.forward_port
        transdelay = packet.size / self.linkc[node][forward_port]
        time.sleep(transdelay)
        #print(transdelay)
        self._step(packet,transdelay)
```

Figure 16: The **_forwadqueue** function

After receiving the packet, the environment processing function **_step** will first check the forwarding port recorded in the packet. If the port is 0, it means that the packet has been delivered. So this function will record the total time spent on the packet transmission and then delete the packet from the forwarding queue, see figure 17.

```
def _step(self, packet,transdelay):
    port = packet.forward_port
    if port == 0:
        #print(packet.node,'routed')
        self.succeed_packets += 1
        self.active_packets -= 1
        current_node_current_buffer = self.packet_queue_size[packet.node].get()
        current_node_changed_buffer = current_node_current_buffer - packet.size
        self.packet_queue_size[packet.node].put(current_node_changed_buffer)
        packet.delay = packet.delay + transdelay
        print(packet.type,packet.source,packet.dest,packet.hops)
        if packet.type == 'TCP':
            self.latency_TCP[packet.source][packet.dest].append(packet.delay)
            # self.tcp_loss[packet.node] = False
        if packet.type == 'UDP':
            self.latency_UDP[packet.source][packet.dest].append(packet.delay)
```

Figure 17: Part of the code of **_step**

If the forwarding port of the packet is not 0, it means that the packet is sent to the next-hop router. Then this function will first check the size of the receiving queue of the next-hop router and if the next-hop router's receiving queue is full this function will drop the packet and set up the TCP loss flag if the type of the packet is TCP, and if the next-hop router's receiving queue is not full this function will add the packet into the receiving queue and record the entry time into the packet, see figure 18.

```
else:
    next_hop = packet.next
    next_hop_current_buffer = self.packet_queue_size[next_hop].get()
    if next_hop_current_buffer + packet.size > self.buffer[next_hop]:
        self.send_fail += 1
        self.packet_queue_size[next_hop].put(next_hop_current_buffer)
        current_node_current_buffer = self.packet_queue_size[packet.node].get()
        current_node_changed_buffer = current_node_current_buffer - packet.size
        self.packet_queue_size[packet.node].put(current_node_changed_buffer)
        if packet.type == 'TCP':
            self.packet_loss_TCP[packet.source][packet.dest].append((next_hop,0,'n_f'))
            self.tcp_loss_1[packet.node] = True
        else:
            self.packet_loss_UDP[packet.source][packet.dest].append((next_hop,0,'n_f'))
        #print(next_hop,'full')
    else:
        packet.hops.append(next_hop)
        current_time = timeit.default_timer()
        packet.reward = packet.delay_1 + transdelay
        packet.delay = packet.delay + transdelay
        packet.qtime = current_time
        self.packet_queue[next_hop].put(packet)
        self.packet_queue_size[next_hop].put(packet.size + next_hop_current_buffer)
        current_node_current_buffer = self.packet_queue_size[packet.node].get()
        current_node_changed_buffer = current_node_current_buffer - packet.size
        self.packet_queue_size[packet.node].put(current_node_changed_buffer)
```

Figure 18: Part of the code of **_step**

Finally, after defining the three function functions that control the behaviour of the router, the class **Network** defines a function namely **router** to instantiate the network environment. Because in the network, all routers are running simultaneously, this function will set the three control functions of each router in different threads, see figure 19.

```
def router(self):
    rece_list = []
    forw_list = []
    get_new_packet_list = []
    for i in range(self.n_nodes):
        rece_list.append(threading.Thread(target=Network._receiveinqueue, args=(self, i + 1)))
    print(len(rece_list))
    for i in range(self.n_nodes):
        for j in range(len(self.links[i + 1])):
            forw_list.append(threading.Thread(target=Network._forwardqueue, args=(self, i + 1, j)))
    for i in range(self.n_nodes):
        get_new_packet_list.append(threading.Thread(target=Network._get_new_packet, args=(self, i + 1)))
```

Figure 19: The thread setting in function **router**

After that, the router will start these threads one by one. These threads will run until each router's **_get_new_packet** completes the defined iteration. The router will call an external thread stopping function **stop_thread** to stop each router's **_receivequeue** and **_forwardqueue**, see figure 20.

```
for i in range(self.n_nodes):
    rece_list[i].start()
    print( i + 1, 'rece start')
for i in range(len(forw_list)):
    forw_list[i].start()
    print( i + 1, 'forward start')
time.sleep(10)
for i in range(self.n_nodes):
    get_new_packet_list[i].start()
for i in range(self.n_nodes):
    get_new_packet_list[i].join()
    print('node ', i+1, ' new packet end')
for i in range(self.n_nodes):
    stop_thread(rece_list[i])
    print('rece',i+1,'stopped')
for i in range(len(forw_list)):
    stop_thread(forw_list[i])
    print('frow', i+1, 'stopped')
```

Figure 20: The starting and stopping of threads in **router**

After the simulation is finished, the router will print the transmission delay and the delivery of all the packets. The router will then create a folder to store the figures which created latter. The name of the folder is defined by the time and date at the time, see figure 21.

```
print(self.latency_UDP)
print(self.latency_TCP)
print(self.packet_loss_UDP)
print(self.packet_loss_TCP)
path = os.getcwd()
Time = time.ctime()
Time = Time.replace(' ', '_')
Time = Time.replace(':', '_')
path = path + '\\' + Time
path = path.replace('\\', '\\\\')
os.makedirs(path)
```

Figure 21: The print of packet's delay and delivery and the creation of figure folder

Next, the router will plot three figures for the UDP packet and the TCP packet respectively according to the statistical data, which are the delay figure, the average delay figure and the packet loss rate figure. When each figure is generated, the figure will be automatically saved

into the previously created figure folder. The code for plotting the delay figure of the UDP packet is shown in figure 22. The codes for plotting other figures are same as the code in figure 22.

```
for i in range(len(self.latency_UDP)):
    for j in range(len(self.latency_UDP[i + 1])+1):
        if j + 1 != i + 1:
            x_list_len = len(self.latency_UDP[i + 1][j + 1])
            x = []
            for q in range(x_list_len):
                x.append(q)
            plt.ylim(0,int(max(self.latency_UDP[i + 1][j + 1]))+1)
            plt.plot(x, self.latency_UDP[i + 1][j + 1])
            plt.title('node ' + str(i + 1) + ' to node ' + str(j + 1) + '_UDP delay')
            plt.xlabel('iteration')
            plt.ylabel('delay')
            plt.savefig('.\\' + Time + '\\' + str(i + 1) + '_' + str(j + 1) + '_UDP delay')
            plt.show()
```

Figure 22: The code for plotting delay figure of the UDP packet

Deep reinforcement learning in Agent.py

The deep reinforcement learning algorithm is defined in this file. In the **Agent.py**, the deep reinforcement learning algorithm is also defined in a class namely **ADQN**. In class **ADQN**, there are 6 defined functions, which are **estimate**, **target**, **learn**, **update_network**, **update_epsilon**, and **show_routing_table**. The function **reset** in the env.py uses the class **ADQN** to instantiate the algorithm in each router. The function **_receivequeue** uses the function **estimate** to find the next-hop router of packet, uses the function **target** to return Q-target, uses the function **learn** to learn the routing, uses the function **update_network** to duplicate weights from main network to target network, uses the function **update_epsilon** to adjust the exploration rate, and uses the function **show_routing_table** to print current routing table, see figure 23.

```
for i in range(1, net_topo.shape[0] + 1):
    print(i)
    self.agent[i] = Agent.ADQN(self.n_nodes, i, len(self.links[i]) - 1)
MinQ_port_eval, forward_port, Q_estimate_list = self.agent[node].estimate(dest,receive_port,epsilon)
MinQ = self.agent[node].target(dest, MinQ_port_eval)
self.agent[node].update_network(beta)
epsilon = self.agent[node].update_epsilon(epsilon)
self.agent[node].show_routing_table()
```

Figure 23: The utilizations of class ADQN in **env.py**

The class **ADQN** firstly uses the input number of nodes, the input number of ports and the input router serial number to define the attributes of the algorithm, which are the neural networks in the deep reinforcement learning. Because there are two networks in the deep reinforcement learning, the class **ADQN** initializes two neural networks for the main network and the target network, respectively. Because the target network needs to duplicate the weights from the main network, when the main network is initialized, the weights of the main network will be stored in a neural network weight file suffixed with h5, see figure 24.

```
class ADQN:
    def __init__(self,n_nodes,node,n_port):
        self.n_nodes = n_nodes
        self.node = node
        self.n_port = n_port
        with graph.as_default():
            self.model = Sequential()
            self.model.add(Dense(16, activation='relu',input_shape=(self.n_nodes+1,)))
            self.model.add(Dropout(0.2))
            self.model.add(Dense(8,activation='relu'))
            self.model.add(Dropout(0.1))
            self.model.add(Dense(n_port,activation='linear'))
            self.model.compile(loss='mean_squared_error', optimizer=RMSprop(lr=0.01), metrics=['accuracy'])
            a = random.randint(1,self.n_nodes)
            state_input = to_categorical(a,num_classes=self.n_nodes+1)
            state_input[0] = 1
            state_input_array = np.array(state_input)
            state_input_array = state_input_array.reshape(1,self.n_nodes+1)
            self.model.predict(state_input_array)
            self.model._make_train_function()
            weight_name = str(self.node) + '_weights.h5'
            self.model.save_weights(weight_name)
        with graph.as_default():
            self.target_model = Sequential()
            self.target_model.add(Dense(16, activation='relu',input_shape=(self.n_nodes+1,)))
            self.target_model.add(Dropout(0.2))
            self.target_model.add(Dense(8,activation='relu'))
            self.target_model.add(Dropout(0.1))
            self.target_model.add(Dense(n_port, activation='linear'))
            self.target_model.compile(loss='mean_squared_error', optimizer=RMSprop(lr=0.01), metrics=['accuracy'])
            a = random.randint(1,self.n_nodes)
            state_input = to_categorical(a,num_classes=self.n_nodes+1)
            state_input[0] = 1
            state_input_array = np.array(state_input)
```

Figure 24: The initialization of neural networks in class **ADQN**

The **estimate** function selects the forwarding port of the packet based on the input destination, the input receiving port, and the input exploration rate. This function will first convert the destination node into a one-hot form, then call the prediction function of the main neural network with inputting the one-hot destination node, then get the delay from each port to the destination node. This function then will select a forwarding port with the ε-greedy method. Additionally, If the port with the smallest delay is the same as the port that receives the packet, the port with the second smallest delay will be set as the smallest one. The purpose of this process is similar to the split horizon in the RIP routing protocol, which means that the router does not forward the packet back to the router that sends the packet to it, see figure 25.

```
def estimate(self,dest,receive_port,epsilon):
    state_input = to_categorical(dest,num_classes=self.n_nodes+1)
    state_input[0] = 1
    state_input_array = np.array(state_input)
    state_input_array = state_input_array.reshape(1,self.n_nodes+1)
    with graph.as_default():
        Q_estimate_array = self.target_model.predict(state_input_array)
    Q_estimate_list = Q_estimate_array.tolist()
    while True:
        Q_min_port = Q_estimate_list[0].index(min(Q_estimate_list[0])) + 1
        if Q_min_port != receive_port:
            break
        else:
            index = Q_estimate_list[0].index(min(Q_estimate_list[0]))
            Q_estimate_list[0][index] = max(Q_estimate_list[0]) + 1
    p_greedy = 1 - epsilon + epsilon / len(Q_estimate_list[0])
    p_not_greedy = epsilon / len(Q_estimate_list[0])
    p_list = []
    for i in range(1,self.n_port+1):
        if i != Q_min_port:
            p_list.append(p_not_greedy)
        else:
            p_list.append(p_greedy)
    p = np.array(p_list)
    port_list = []
    for i in range(self.n_port):
        port_list.append(i+1)
    while True:
        Q_egreddy_port = np.random.choice(port_list, p=p.ravel())
        if Q_egreddy_port != receive_port:
            break
    return Q_min_port,Q_egreddy_port, Q_estimate_list[0]
```

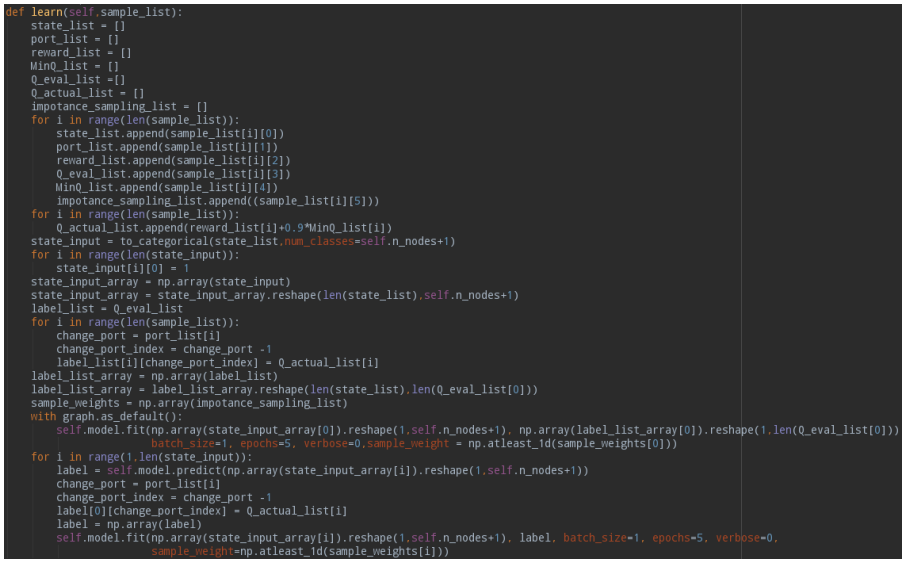
Figure 25: The **estimation** function of **ADQN**

The **target** function calculates and returns the minimum state-action-value function of the target network based on the input destination node and the input minimum delay port, which is the action evaluation in the double Q-learning. The target function will first read the stored neural network weight h5 file and then load this file to the target network. Next it will convert the destination node to the one-hot form and call the prediction function of the target network with inputting the one-hot destination node. After getting the delay from each port to the destination node, this function will return the minimum delay port's minimum state-action-value function, see figure 26.

```
def target(self,dest,MinQ_port_eval):
    weight_name = str(self.node) + '_weights.h5'
    with graph.as_default():
        self.target_model.load_weights(weight_name)
    state_input = to_categorical(dest, num_classes=self.n_nodes+1)
    state_input[0] = 1
    state_input_array = np.array(state_input)
    state_input_array = state_input_array.reshape(1, self.n_nodes+1)
    with graph.as_default():
        Q_estimate_array = self.target_model.predict(state_input_array)
    Q_estimate_list = Q_estimate_array.tolist()
    Q_min_actual = Q_estimate_list[0][MinQ_port_eval-1]
    return Q_min_actual
```

Figure 26: The **target** function of **ADQN**

The **learn** function uses the input sample set to update the weights of the main network. This function will first read the components in the samples, which are the destinations, the forwarding ports, the one-step rewards, the current network's Q value output list, the minimum Q values of next-hop routers, and the importance sampling parameters. Then this function will calculate the updated Q value for each sample and use the updated Q values to replace the Q values of the corresponding forwarding port in the current network's Q value output list. Then the new Q value output lists are used as the labels for the samples in the supervised learning. After that, this function will train the main network accompanied with the importance sampling factor, see figure 27.



```
def learn(self,sample_list):
    state_list = []
    port_list = []
    reward_list = []
    MinQ_list = []
    Q_eval_list =[]
    Q_actual_list = []
    impotance_sampling_list = []
    for i in range(len(sample_list)):
        state_list.append(sample_list[i][0])
        port_list.append(sample_list[i][1])
        reward_list.append(sample_list[i][2])
        Q_eval_list.append(sample_list[i][3])
        MinQ_list.append(sample_list[i][4])
        impotance_sampling_list.append((sample_list[i][5]))
    for i in range(len(sample_list)):
        Q_actual_list.append(reward_list[i]+0.9*MinQ_list[i])
    state_input = to_categorical(state_list,num_classes=self.n_nodes+1)
    for i in range(len(state_input)):
        state_input[i][0] = 1
    state_input_array = np.array(state_input)
    state_input_array = state_input_array.reshape(len(state_list),self.n_nodes+1)
    label_list = Q_eval_list
    for i in range(len(sample_list)):
        change_port = port_list[i]
        change_port_index = change_port -1
        label_list[i][change_port_index] = Q_actual_list[i]
    label_list_array = np.array(label_list)
    label_list_array = label_list_array.reshape(len(state_list),len(Q_eval_list[0]))
    sample_weights = np.array(impotance_sampling_list)
    with graph.as_default():
        self.model.fit(np.array(state_input_array[0]).reshape(1,self.n_nodes+1), np.array(label_list_array[0]).reshape(1,len(Q_eval_list[0])),
                       batch_size=1, epochs=5, verbose=0,sample_weight = np.atleast_1d(sample_weights[0]))
    for i in range(1,len(state_input)):
        label = self.model.predict(np.array(state_input_array[i]).reshape(1,self.n_nodes+1))
        change_port = port_list[i]
        change_port_index = change_port -1
        label[0][change_port_index] = Q_actual_list[i]
        label = np.array(label)
        self.model.fit(np.array(state_input_array[i]).reshape(1,self.n_nodes+1), label, batch_size=1, epochs=5, verbose=0,
                       sample_weight=np.atleast_1d(sample_weights[i]))
```

Figure 27: The **learn** function of **ADQN**

The **update_network** and **update_epsilon** are two simple functions which are used to duplicate the weights from main network to target network and adjust the exploration rate of the deep reinforcement learning. The **update_network** function will store current main network's weights in a neural network weight h5 file, and if the file is already existing the new file will overwrite the old file as the update of the target network weights, see figure 28.

```
def update_network(self,beta):
    weight_name = str(self.node) + '_weights.h5'
    self.model.save_weights(weight_name)
```

Figure 28: The **update_network** of **ADQN**

The **update_epsilon** function is used in **_receivequeue** in **env.py** to reduce the exploration rate. In the reinforcement learning, the exploration rate ε is set to gradually decrease during the learning process. The purpose of it is that a decreasing exploration rate can accelerate the convergence of the policy. However, if the link in the network is congested, the already converged routing policy may need to be changed. At this time, a small exploration rate is inefficient for policy to converge. Therefore, in the **_receivequeue** when the TCP loss flag is

set up the attenuated ε will be reset to an enough scale for the reinforcement learning process, see figure 29.

```
def update_epsilon(self,epsilon):
    if epsilon > 0.1:
        epsilon -= 0.2
    return epsilon
```

```
if self.tcp_loss[node]:
    print(node, 'epsilon reset')
    self.tcp_loss[packet.node] = False
    epsilon = 0.5
elif self.tcp_loss_1[node]:
    print(node, 'epsilon reset')
    self.tcp_loss_1[packet.node] = False
    epsilon = 0.5
```

Figure 29: The **update_epsilo**n function of **ADQN** and epsilon reset in **_receivequeue**

The **show_routing_table** function is used in **_receivequeue** in **env.py** to print the current routing table. This function will fist reload the neural network weight h5 file and then generate an input set for all destination nodes in the form of one-hot. Next, it will call the prediction function of the target network to get the routing table, see figure 30.

```
def update_network(self,beta):
    weight_name = str(self.node) + '_weights.h5'
    self.model.save_weights(weight_name)
    if beta > 0.1:
        beta -= 0.3
    return beta
def update_epsilon(self,epsilon):
    if epsilon > 0.1:
        epsilon -= 0.2
    return epsilon
def show_routing_table(self):
    weight_name = str(self.node) + '_weights.h5'
    with graph.as_default():
        self.target_model.load_weights(weight_name)
    test_list = []
    for i in range(1,self.n_nodes+1):
        if i != self.node:
            test_list.append(i)
    for i in range(len(test_list)):
        state_input = to_categorical(test_list[i], num_classes=self.n_nodes + 1)
        state_input[0] = 1
        state_input_array = np.array(state_input)
        state_input_array = state_input_array.reshape(1, self.n_nodes + 1)
        with graph.as_default():
            Q_estimate_array = self.target_model.predict(state_input_array)
        print(state_input_array)
        print(Q_estimate_array)
```

Figure 30: The **show_routing_table** function of **ADQN**

Analysis of Results

When the program is running, the program first uses the class **Network** to instantiate a network environment object. This object then reads the pre-set network parameter CSV file and inputs a total iteration to initialize the network environment, see figure 31.

```
if __name__ == '__main__':
    N = Network()
    N.reset('network_sample.csv', 1500)
    print(N.n_nodes)
    print(N.packet_queue)
    print(N.packet_forward_queue)
    print(N.packet_queue_size)
    print(N.links)
    print(N.linkc)
    print(N.buffer)
    print(N.arrivalmean)
    print(N.agent)
    print(N.port)
    print(N.latency_UDP)
    print(N.back)
    print(N.recovery_rate)
    N.router()
```

Figure 31: The execution code of the program

The network then prints out all important network parameters after initialization. Finally, the program calls the **router** function to build the network topology and start the simulation see figure 32.

Figure 32: The outputs of the initialization of network parameters

When the simulation starts, the information of the delivered packets will be printed one by one in the Python console, including the type of the packet, the source node and the destination node of packet, and the routers that the packet passes through, see figure 33.

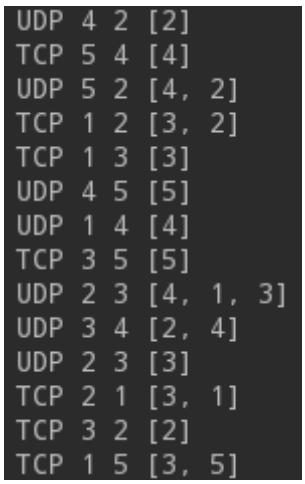


Figure 33: Printing of the information of delivered packets:

After every 50 iterations, the program will print the current iteration, see figure 34.

Figure 34: The program prints the current iteration after every 50 iterations

Each time the main network updates the weights, the program will print out the current routing table of the main network, that is, the delay to each destination node, see figure 35.

```
node  5  NN output
[[1. 1. 0. 0. 0. 0.]]
[[7.946504  4.4479365]]
[[1. 0. 1. 0. 0. 0.]]
[[8.1208   4.501824]]
[[1. 0. 0. 1. 0. 0.]]
[[5.002184  3.1709294]]
[[1. 0. 0. 0. 1. 0.]]
[[4.8285656 3.0983682]]
```

Figure 35: The example output routing table of the main network (node 5)

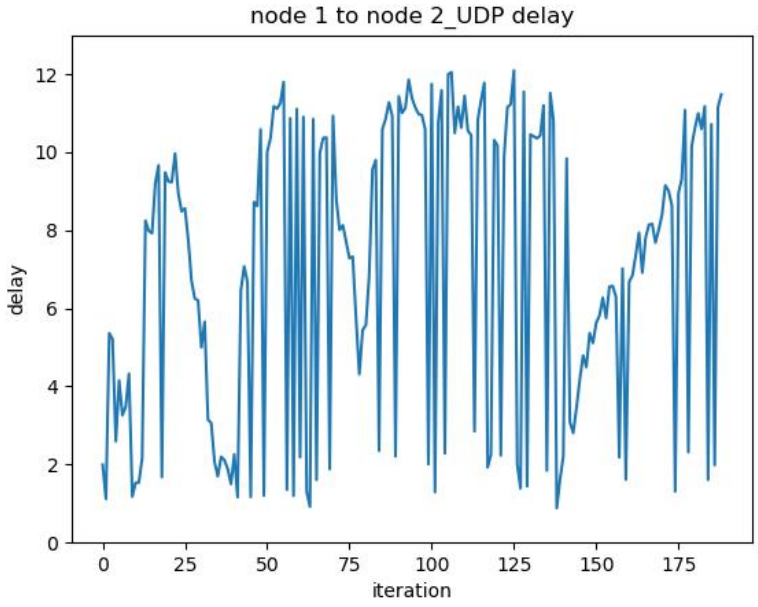
When the router detects a TCP packet loss, the router will reset the exploration rate, which is printed out in the Python console, see figure 36.

```
TCP 5 1 [3, 1]
2 epsilon reset
TCP 4 5 [5]
UDP 3 5 [5]
5 epsilon reset
4 epsilon reset
TCP 1 3 [3]
4 epsilon reset
UDP 5 3 [3]
TCP 3 1 [1]
TCP 1 3 [3]
TCP 5 4 [4]
```

Figure 36: The resets of the exploration rate

UDP packet analysis

In the reinforcement learning routing algorithm of this project, the UDP packets are used for exploration. Because UDP packets are not sensitive to packet loss and the exploration always requires sacrifice, UDP packets can be used to explore the routes. Therefore, the stability of the end-to-end delay of UDP packets is worse than that of TCP packets. Take node 1 as an example, see figure 37.



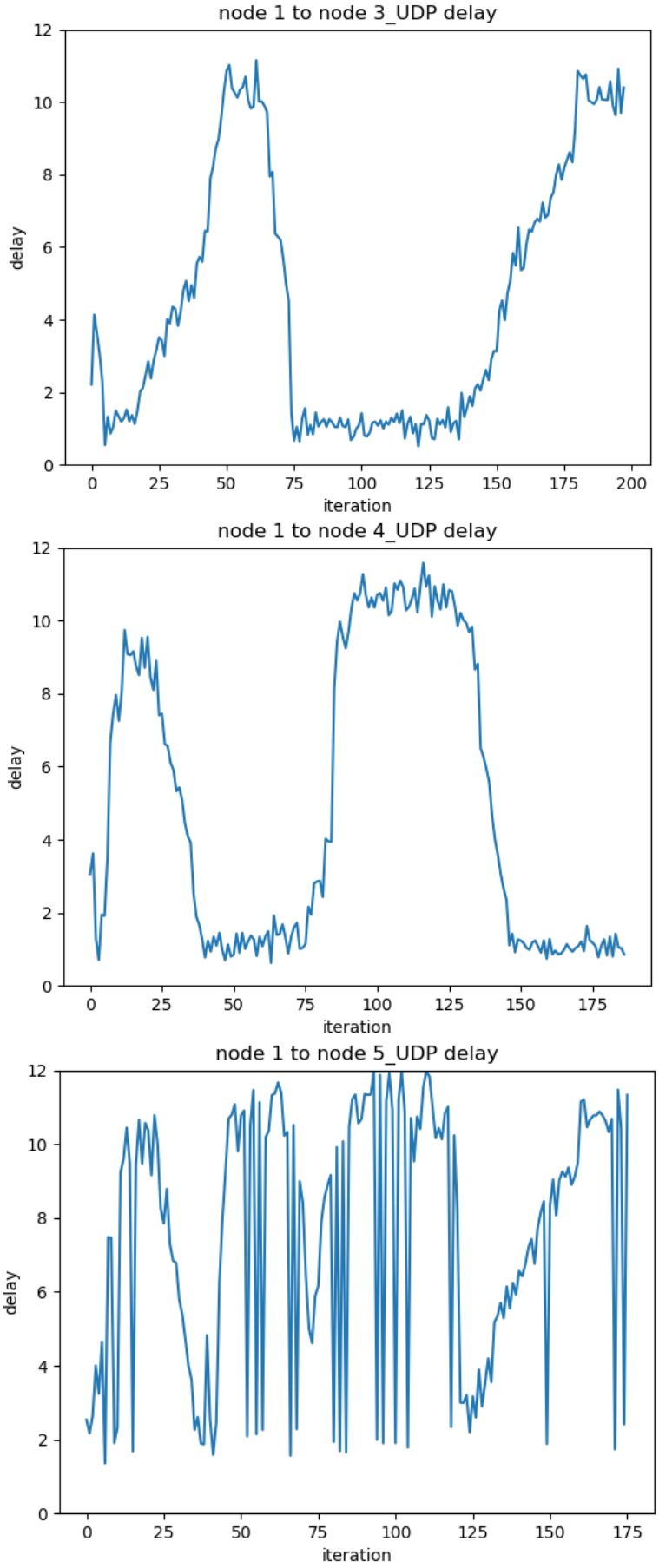
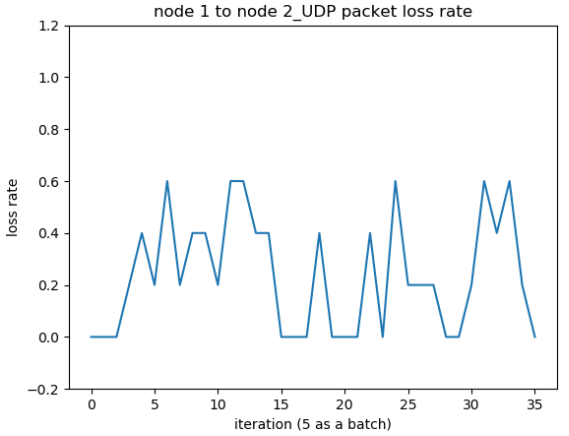


Figure 37: The end-to-end delay from node 1 to other nodes for UDP packet

According to figure 9, the node 3 and node 4 are the neighbour nodes of node 1 and node 2 and node 5 are the far-end node of node 1. It can be seen from figure 36 that the end-to-end delay from node 1 to two far-end nodes on the delay figure have large fluctuations. This is the sign of the exploration of UDP packet, which means that the router will use the epsilon-greedy algorithm when forwarding UDP packets. Additionally, the delays from node 1 to all other routers are all wavy with respect to the iteration, which is the sign of the policy updating in the reinforcement learning routing algorithm. Take node 1 to node 2 can an example, from the network topology in figure 8, if node 1 wants to send a packet to node 2, there are 4 routes, which are 1 → 3 → 2, 1 → 4 → 2, 1 → 3 → 5 → 4 → 2, and 1 → 4 → 5 → 3 → 2. At the initial stage (before 25 iterations), the network does not know the optimal route from node 1 to node 2, then the network tried all the routes to node 2, so the delay from node 1 to node 2 is not stable. Additionally, at this stage node 1 found that node 3 and 4 are neighbour nodes, so the delay from node 1 to node 3 and 4 decreased. At the second stage (around 25 to 50 iterations), the network found the route 1 → 3 → 2 has the minimum end-to-end delay, so the policy was converged to this route, and the delay from node 1 to 2 decreased. because the packets were queued on this route, the delay from node 1 to node 2 and 3 then gradually increased, and because there are not many packets queued on route 1 → 4 → 2, so the delay from node 1 to 4 maintained at a low level. At the third stage (around 50 to 75 iterations), the route 1 → 3 → 2 was congested, and the network found that route 1 → 4 → 2 had a lower end-to-end delay, then the policy was switched to 1 → 4 → 2, so the delay from node 1 to 2 and 3 decreased, and the delay from node 1 to 4 increased because the packets were queued on this route. The routing policy from node 1 to node 2 switched back and forth between these two routes, which shows the adaptability of the reinforcement learning routing algorithm to the variable network conditions. The packet loss figures of node 1 of UDP packet are shown in figure 38.



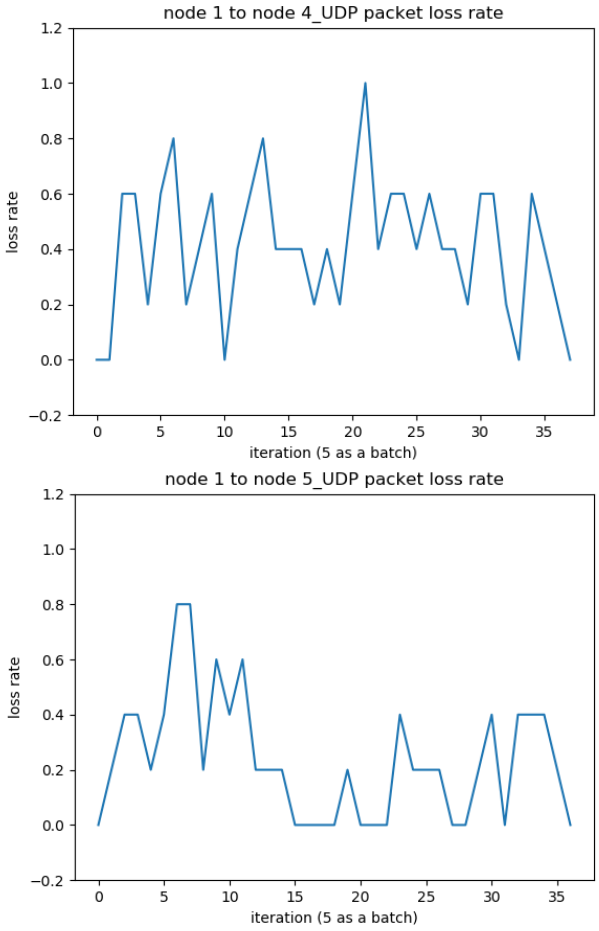


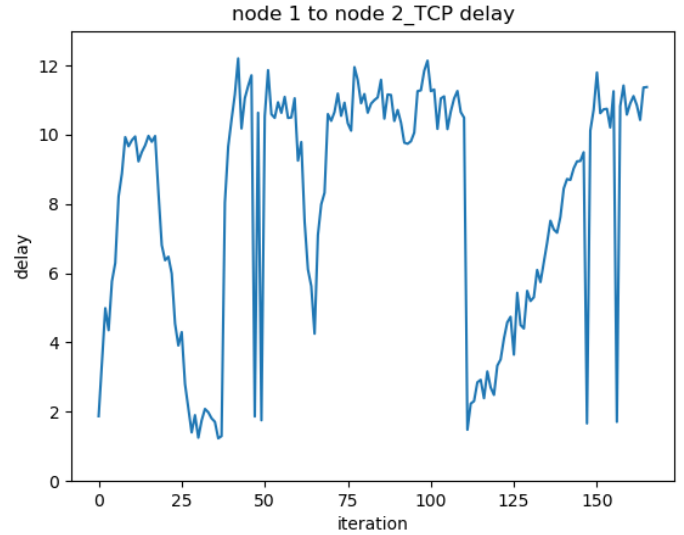
Figure 38: The packet loss rate figure of node 1of UDP packet

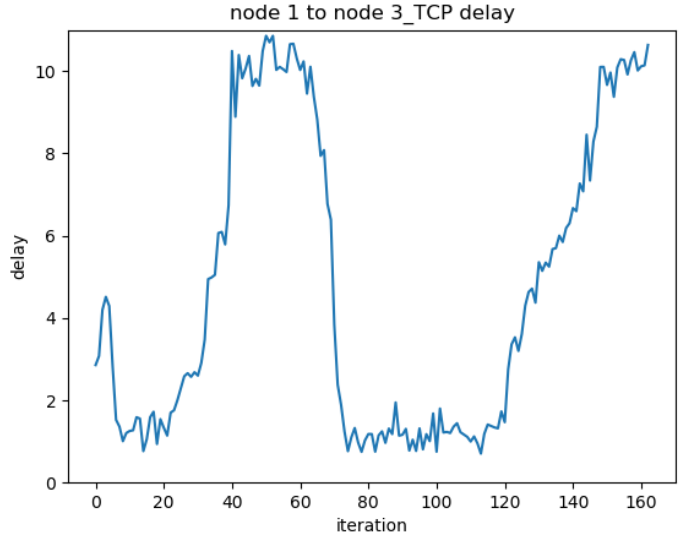
It can be seen from Figure 37 that the loss rate of UDP packet is relatively large, and the average packet loss rate is basically around 0.4. The UDP packet loss figures of node 1 also show rough wave shapes, and the reason is the same as the UDP packet delay figures of node 1, that is, the routing policy is constantly switching between different routes. Because in the setting of the network parameters, node 4 is set as a primary intermediate node to connect node 1,2 and 5, and node 3 is set as a secondary intermediate node. Therefore, the link capacities of node 3 are larger than that of node 4. When the network environment is running, node 1, 2 and 5 will always select node 3 as the intermediate node when sending packets to the far-end nodes, and when node 3's links are congested, they will choose to send packets through node 4. Once the congestion of node 3's links are relieved, they will choose node 3 again. Therefore, the node 3 as a primary intermediate node, its network traffic is always limited within a range in the environment, so node 1 to node 3's packet loss rate is always small in the figure 38.

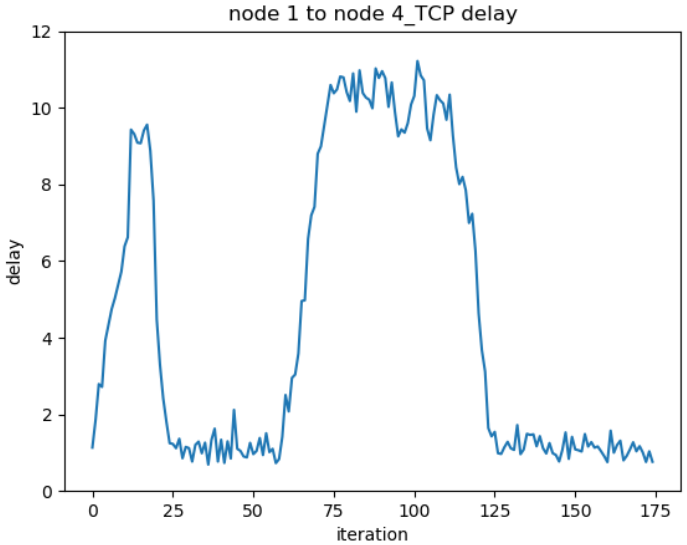
TCP packet analysis

Since the TCP packet and the UDP packet share a common routing policy, the delay of TCP packet is not much different from the delay of UDP packet. The differences between UDP packet and TCP packet in forwarding is that firstly the TCP packet is very sensitive to packet loss because it has high requirements for information integrity. Therefore, the TCP packet does not participate in the exploration, that is, the router uses full-greedy method when forwarding TCP packet instead of ε-greedy method. Secondly, the router's routing algorithm

has some TCP packet loss flags. When these flags are set up, the router will forward some TCP packets along the suboptimal port to ensure that TCP packets are not lost on the link of the optimal port. Thirdly, when these TCP loss flags are set up, the router will also randomly drop a part of the UDP packets to relieve the link congestion. In general, the delay of TCP packet is more stable than UDP packet, and the total number of deliveries of TCP packet is larger than that of the UDP packet. Also take node 1 as an example, see figure 39







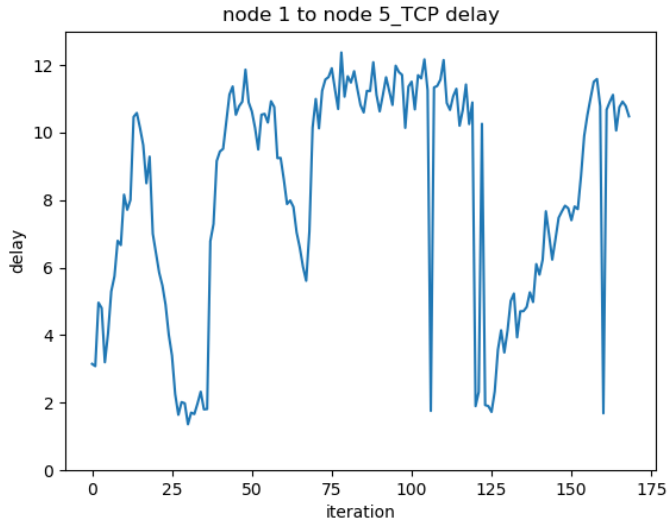
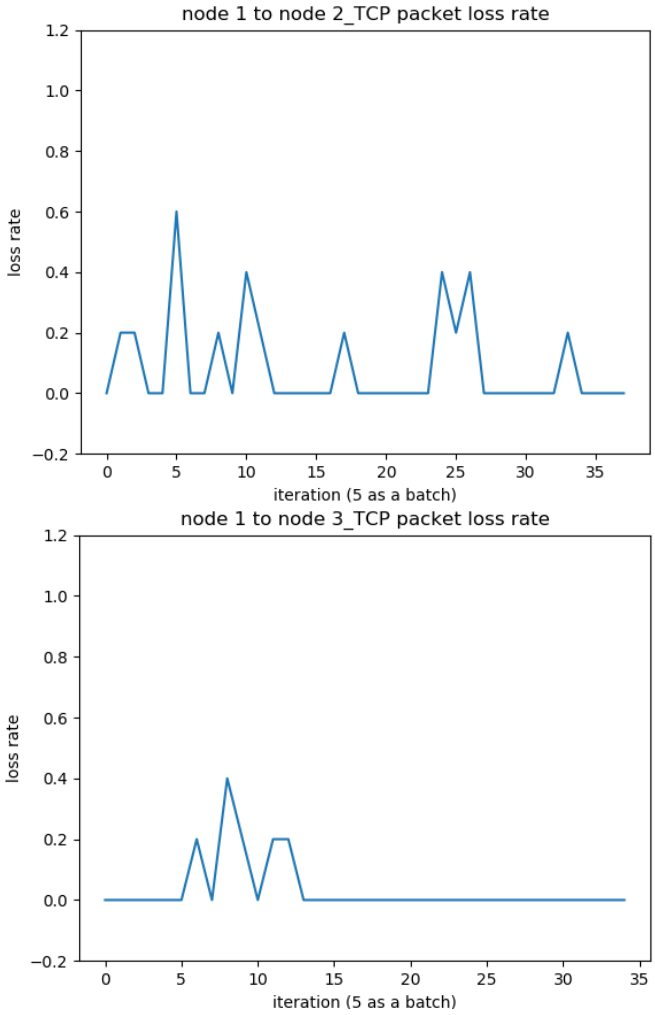


Figure 39: The end-to-end delay from node 1 to other nodes for TCP packet

It can be seen from Figure 39 that although the TCP packet does not participate in the exploration, there are still some fluctuations in its delay figures, which is the sign that TCP packets are forwarded along the suboptimal path. This also illustrates that there is a delay between the router's policy and the real-time network condition. How to shorten this delay so that the router can quickly adapt to the changes in the network environment is an important issue that the routing algorithm needs to consider.



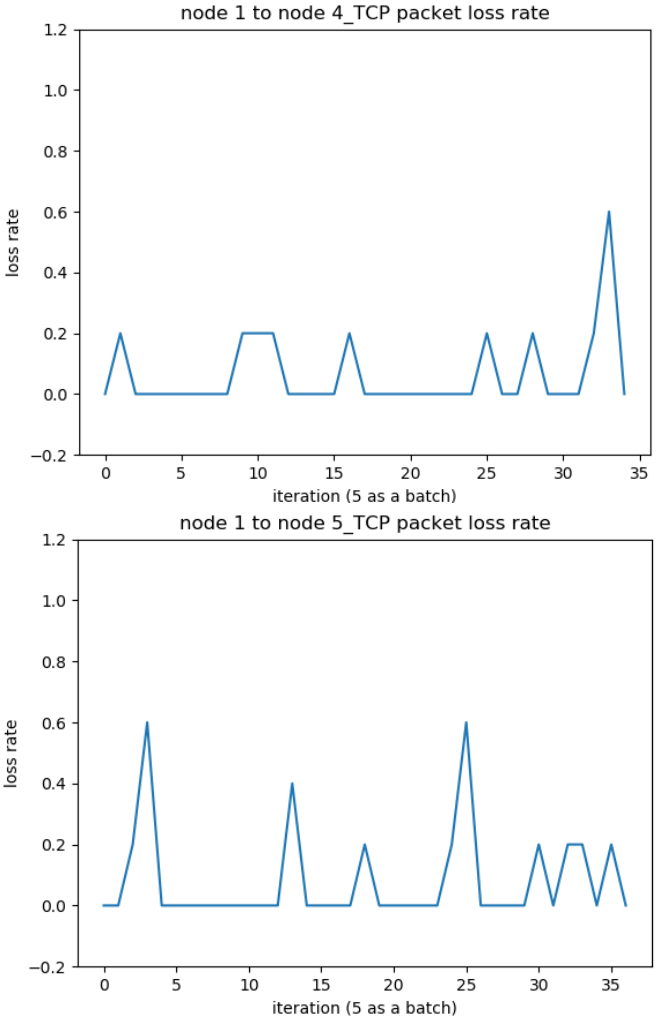


Figure 40: The TCP packet loss figures of node 1

The TCP packet loss figures of node 1 are shown in figure 40. It can be seen from the figure that the packet loss of TCP packet is less than that of the UDP packet. This shows that in the routing algorithm, forwarding TCP packets along the suboptimal routes and probabilistically dropping UDP packets can efficiently improve the delivery rate of the TCP packet.

Limitation analysis of the reinforcement learning routing algorithm

It can be seen from the simulation results that the reinforcement learning routing algorithm in this program can provide basic dynamic routing functions like general dynamic routing algorithms, and it also has different forwarding rules for different types of packets, which can better adapt to the complex network environments. However, it also has many deficiencies. Firstly, although the routing algorithm in this program has different forwarding rules for UDP packets and TCP packets, they are both based on the outputs of the same neural network. Ideally, the forwarding of UDP packets and TCP packets should be based on different neural networks. The UDP packet's neural network should be used to find the minimum end-to-end delay, so the Q value represents the end-to-end delay from the local node to all other nodes, and the returned reward is the one-step delay of the two neighbour nodes. The TCP packet's neural network should be used to find the minimum end-to-end packet loss rate, so its Q value represents the end-to-end packet loss rate from the local node to all other nodes. The returned reward is the packet loss rate between the two neighbour nodes. However, this is

not possible under the existing hardware conditions of this project. In this program, each router is placed in a thread, and each thread has two neural networks, which are the main network and the target network. All threads are running at the same time during the simulation progresses. This program used a network environment setting with five routers, so a total number of 10 neural networks are running simultaneously during the simulation. After many tests, the computer hardware used in this project can support up to 10 to 15 neural networks running simultaneously. Additionally, Because Python's multi-threading cannot make good use of the performance of multi-core CPUs, when the number of threads increases the speed of the training of the neural network will become relatively slow. Therefore, the TCP packet routing rules used in this program is a compromising solution.

Secondly, the router needs to know some global network environment parameters to learn the routing, such as the total number of routers in the network and the port correspondence among routers. However, in the real cases the router may not be able to obtain these pieces of information immediately at the beginning, the learning efficiency of the router may be much lower than that of the program in the simulation. These problems can be solved by the following methods. Firstly, rather than using the one-hot form, other representation methods can be used to represent the destination node. Hence there is no need to consider the total number of routers in the entire network. However, the new method also needs to ensure the discontinuities between discrete features. Secondly, for the port correspondence, the router can use the state request packet and the state update packet to obtain the port correspondence like other general dynamic routing algorithms.

Thirdly, in order to simplify the complexity of the model, the network traffic model used in this project is a simple uniform model, that is, each router generates new packets at regular time intervals. The advantage of this model is that each simulation result can tend to be the same, which can help the designer to check the functionality of the routing algorithm. However, this also makes the model lose the authenticity. The balance between the model authenticity and the simulation friendliness is a trade-off problem. As an improvement, the Poisson network traffic model can be used, which is a more authentic model and easy to be implemented. However, the parameters of the Poisson process are not easy to be determined. Inappropriate parameters can make the model become unstable, so it requires a lot of tests to determine the appropriate parameters.

Conclusions

Traditional routing algorithms have gradually become unable to adapt to the current complex and variable network environment. To solve this problem, many network engineers and scientists committed themselves to develop new generation routing algorithms. The machine learning-based routing algorithm has quickly come into people's sight and received a lot of attentions. Among them, the routing algorithm based on the reinforcement learning was regarded the most suitable algorithm for the current network environment. It gives routers context awareness and intelligence which can help routers to better adapt to the variable and complex network environment. This paper proposed a deep routing algorithm based on the Q-learning and the neural network, which can use the neural network to learn the routing and make optimal routing according to network conditions.

The deep reinforcement learning routing algorithm in this project was implemented in a Python program. This Python program contains two files, one for defining and initializing the network environment and the other for defining and initializing the neural networks. In the first file, the network environment is defined by a class, which includes the network parameter initialization function, the network topology building function, the packet receiving and forwarding function, and the packet generation function. In the second file, the neural networks in the deep reinforcement learning are also defined by a class, which includes the neural network building function, the neural network learning function, the neural network parameter updating function, and the interaction function between the main network and the target network. The program in this project instantiated a network environment to implement the Q-learning and neural network based routing algorithm by calling the above two classes.

It can be seen from the experimental results that the deep reinforcement learning routing algorithm in this project can implement the basic dynamic routing functions. The router can dynamically select the optimal route based on the current states of the network links. The routing algorithm in this project uses the same neural network for forwarding UDP packet and TCP packet, but it has special forwarding rules for TCP packets. This is a compromise solution due to the limitation of the hardware conditions. This program can be improved by following methods. Firstly, two different neural networks can be used for forwarding UDP packet and TCP packet, which can improve the forwarding efficiency and reduce the loss rate. Secondly, the program can use other representation methods to represent the network parameters, which can solve the problem that the router needs to know the global network parameters before the simulation. Thirdly, a more authentic network traffic model can be used to replace the uniform model used in the program, which can improve the authenticity of the model.

In summary, the routing algorithm based on the deep reinforcement learning is still in the development stage. Due to its high performance and high adaptability, it has very strong practicability and broad application prospects. It can be seen from the examples in this project that the existing reinforcement learning routing algorithms are only suitable for small networks. For large networks, because the number of network parameters is too large, it is necessary to generalize these large number of parameters. This is the problem that many network engineers and scientists are currently researching. In addition, the deep learning used in this project is only a simple fully connected network. However, some more advanced neural networks such as convolutional neural network or recurrent neural network can be used in the routing algorithm, which may solve the generalization problem of the parameters.

References

[1] Y. Wang et al., "A Rank-based Q-routing Algorithm", *Computers and modernization,* vol. 278, 2018, DOI: 1006-2475.2018.10.001.

[2] E. O. Neftci and B. B. Averbeck, "Reinforcement learning in artificial and biological systems", Nature Machine Intelligence, vol. 1, pp. 133-143, 2019, DOI: 10.1038/s42256-019-0025-4.

[3] J. Boyan and M. Littman, "Packet routing in dynamically changing networks: a reinforcement learning approach", *Proceedings of NIPS Adv neural information processing systems*, pp. 671-678, 1994.

[4] J. Kennedy and R. Eberhart, "Particle swarm optimization", *Proceedings of IEEE international conference neural networks*, PP. 1932-1948, 1995. DOI: 0-7803-2768-3/95.

[5] R. Rojas, Neural Networks: a systematic introduction, Berlin, New York, 1996.

[6] M. Gen and R. Cheng, "Genetic algorithms and engineering optimization", Wiley, New York, 1999.

[7] Y. Zhang and M. Fromherz, "Constrained flooding: a robust and efficient routing framework for wireless sensor networks", *Proceedings of AINA 20th international conference advanced information networking and applications*, 2006, DOI: 10.1109/AINA.2006.132.

[8] A. Foster, "Machine learning techniques applied to wireless ad-hoc networks: guide and survey", Proceedings of ISSNIP 3rd international conference intelligent sensors, Networks and information, pp.365-370, 2007, DOI: 10.1109/ISSNIP.2007.4496871.

[9] K-LA. Yau et al., "Reinforcement learning for context awareness and intelligence in wireless network: review, new features and open issues", Journal of Network Computer Applications, vol. 35, issue 1, pp. 253-267, 2012, DOI: 10.1016/j.jnca.2011.08.007.

[10] O. Alagoz et al., *"Markov Decision Processes: A Tool for Sequential Decision Making under Uncertainty"*, Medical Decision Making, 2010, DOI: 10.1177/0272989X09353194.

[11] R. S. Sutton and A. G. Barto, Reinforcement Learning: An introduction, The MIT Press, 2nd edition, pp. 23-44, 2018.

[12] R. S. Sutton and A. G. Barto, Reinforcement Learning: An introduction, The MIT Press, 2nd edition, pp. 47-60, 2018.

[13] R. S. Sutton and A. G. Barto, Reinforcement Learning: An introduction, The MIT Press, 2nd edition, pp. 91-115, 2018.

[14] R. S. Sutton and A. G. Barto, Reinforcement Learning: An introduction, The MIT Press, 2nd edition, pp. 131-132, 2018.

[15] H. A. A. Al-Rawi et al., "Application of reinforcement learning to routing in distributed wireless networks: a review", Artif Intell Rev, vol. 43, pp. 381-416, DOI: 10.1007/s10462-012-9383-6.

[16] V. Mnih et al., "Human-level control through deep reinforcement learning", Nature, vol. 518, pp. 529-533, 2015, DOI: 10.1038/nature14236.

[17] H. V. Hasselt et al. (8 Dec 2015), Deep Reinforcement Learning with Double Q-learning [Online]. Available: https://arxiv.org/pdf/1509.06461.pdf [Accessed on 19/06/2019]

[18] T. Schaul et al., "Prioritized Experience Replay", International Conference on Learning Representations, 2016.

Appendices

Appendix 1: The full codes in this project

The full codes in this project can be obtained in my Github repository: https://github.com/JinhaoWu/Msc-Project

Appendix 2: The demonstration of the program

The demonstration of the program and the presentation of the results were recorded and uploaded to my Youtube channel: https://www.youtube.com/watch?v=2XMXZTM1bts&t=8s