# 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

1. **Abstract**

Text here, maximum of 1 a page

**Table of Contents**

Add a table of contents including section numbers and page numbers. You can also include a nomenclature table and a list of figures and tables, as appropriate to your project.

# 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 current growing network. 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 show 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 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 agents 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 transferring the agent to new states [3]. A complete agent and environment pair define a learning task, which is an instance of the reinforcement learning, see figure 1.

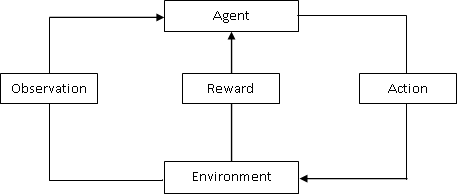


Figure 1: The reinforcement learning process

## 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 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 graphs 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 [5] proposed the prototype of the reinforcement Q-learning routing algorithm in 1994. Kennedy and Eberhart [6] proposed the swarm intelligence algorithm in 1995. Rojas [7] proposed the application of neural networks in network routing in 1996. Gen and Cheng [8] proposed the genetic algorithm in 1999, and Zhang and Fromherz [9] implemented the reinforcement Q-learning routing algorithm for practice in 2006. Forster [10] summarized and compared several machine learning algorithms that were popular at the time in 2007. 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. [11] 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 [11]. 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 that proposed by the Russian mathematician Andrei Markov [12]. It refers to the process in which an agent learns the most rewarding policy in an environment whose states have Markov property [12]. Markov property indicates that the next state of the system is only related to the current state, but not to the previous state [12]. In an environment whose states have 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 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 of the environment.

Sutton and Barto [13] 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

(1)

Where γ is the discount rate and 0 < γ <1, k is the iteration and when the MDP does not have terminal state the k can increase to infinite, and rt+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 vπ(s) and state-action-value function qπ(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.

(2)

(3)

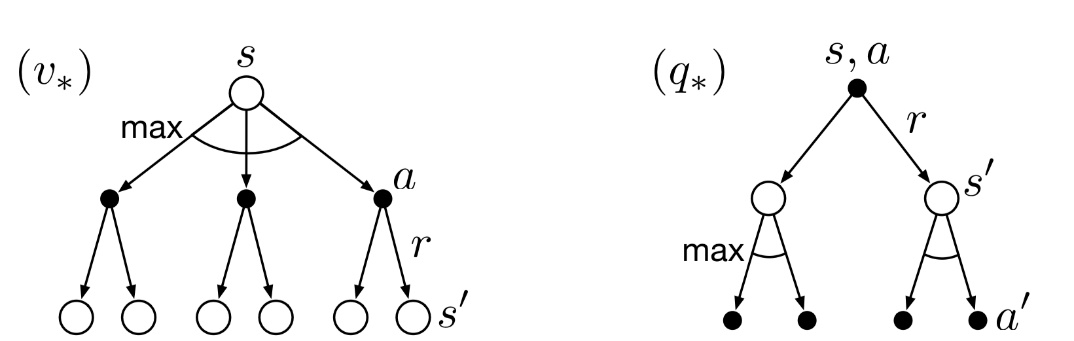
These two value functions are the objective functions of the reinforcement learning. Just like the weights of the neural network constantly updating in supervised learning to minimize the objective error function, the goal of the reinforcement learning is to update the policy to minimize (or maximize) the two value functions. The easiest way to update a policy in reinforcement learning is the iterative method, which requires the implementation of Bellman equations [14]. The Bellman equation was proposed by American mathematician Charlie Berman, 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 [14].

Figure 2: The backup figures of value function vπ(s) and qπ(s,a) [14]

Figure 2 are the backup diagram 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

(4)

(5)

where π(a|s) is the probability of the agent that making an action a at the state S under the policy π, is the reward that the environment giving to the agent when agent transfers from state S to state S’, and 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 vπ(s) can be rewritten as

(6)

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

(7)

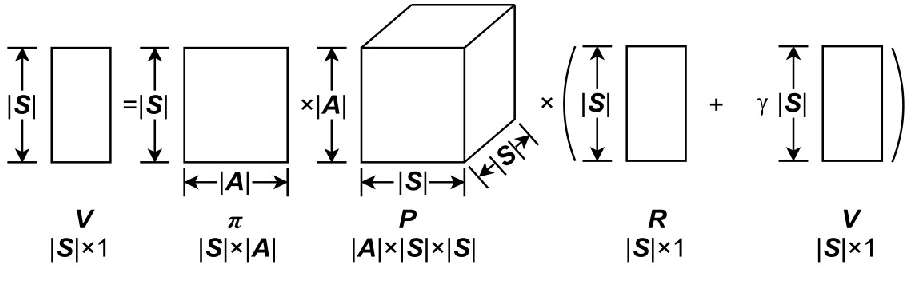
The equation 6 and 7 are the bellman functions in the reinforcement learning. There is an important assumption in MDP which is the time independency. The time independency refers to the value functions of two same states at different instants of times in an MDP will eventually become same with the continuous 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 [14]. 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 [14]

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

(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

, , (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 [15]. This learning method is called the policy iteration method, which is the easiest way 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

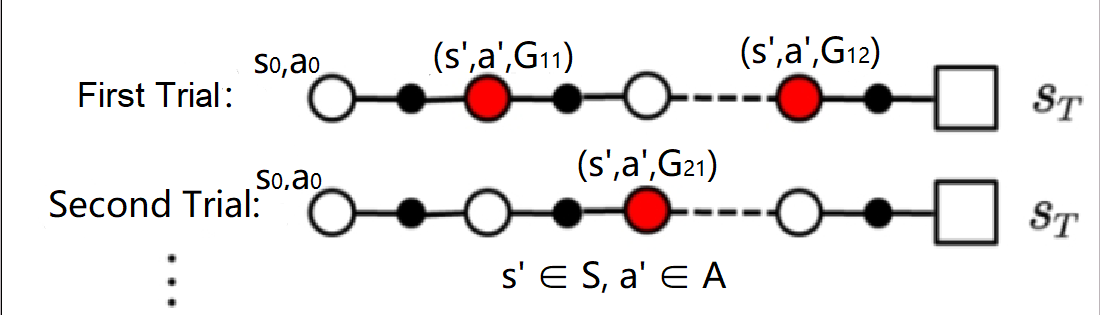
The environmental information in most reinforcement learning instances is unknown, and these instances belong to model-free reinforcement learning model. For model-free reinforcement learning model, the policy iterations cannot be used directly. The Monte-Carlo method is a good approach to the model-free 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 of the value functions, then the policy iteration method can be applied subsequently.

Figure 3: The trials in the Monte-Carlo method [16]

Figure 3 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 rewards of each state-action pairs that appear in the trial sequences can be calculated, then the average values can be calculated as the estimations of the value functions of these state-action pairs, see equation 10.

𝑎’∈𝐴, s’∈𝑆 (10)

Where N(s’,a’) is the Nth 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

𝑎∈𝐴, s∈𝑆 (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 [16].

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 value of state-action-value functions.

### 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

(12)

where k is the kth 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

(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 (st,at).

(14)

Next, the long-term reward G(st,at) in the equation 14 can be divided into two parts, the one-step reward of state-action pair (st,at) and the maximum state-action-value function of the next state-action pair (st+1,at+1), see equation 15

(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 value function can be performed simultaneously, the Q-learning is much more efficient than the Monte-Carlo method [17].

## 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 [5] 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

(16)

Where t is the current instant of time, i,j and k are the serial numbers of the routers in the network, and are the sets of the next-hop routers of ith and jth routers, and 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.

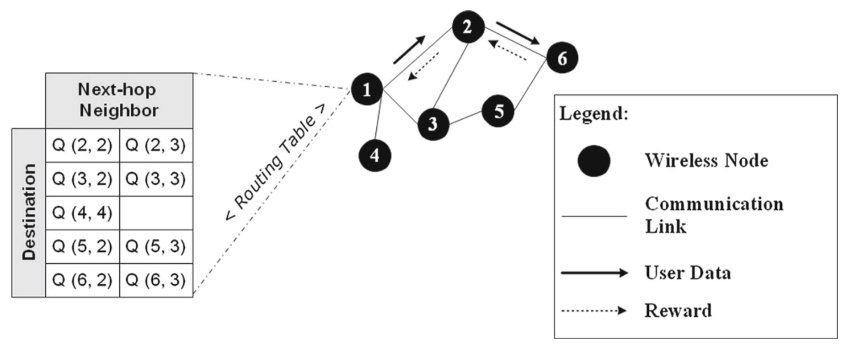
The principle of the Q-routing is that the routers in the network store the value functions in their local memories as the routing tables, and continuously update them by forwarding and receiving packets, and eventually the. For example, see figure 4, which is a network topology with the minimum delay Q-routing algorithm.

Figure 4: A network topology with Q-routing algorithm [18]

Just as shown in the figure 4, the router 1 has a routing table which contains the state-action-value function of each destination and 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 the state-action-value functions with 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 the minimum Q value to destination router 6. When the router 1 receives the returned reward (delay) it will update the value of Q(6,2) by using the equation 16.

### Deep Q-routing

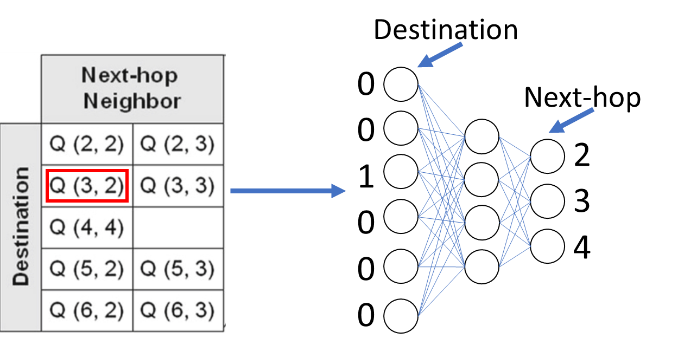
The Q-routing has a deficiency, which is the inefficiency in the large-scale network. When the scale of the network becomes large the size of the Q-routing table will increase exponentially, which will reduce the packet processing efficiency of the router. The neural network in the supervised learning 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 5.

Figure 5: The Q table and Q neural network [18]

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 [19]. 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 [19]. After DeepMind proposing the DQN, Hasselt [20] and Schaul [21] proposed their improved version of 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 [19]. The random selection in the replay buffer can break the correlation among consecutive samples [19]. 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 [21]. The Q error is shown as below

(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

(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 can 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

(19)

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

The priority replay can let router 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 [19]. 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 [20]. 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.

(20)

The action selection refers to finding the action a’ which has the largest state-action-value function in the next state St+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.

(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 the 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 were 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

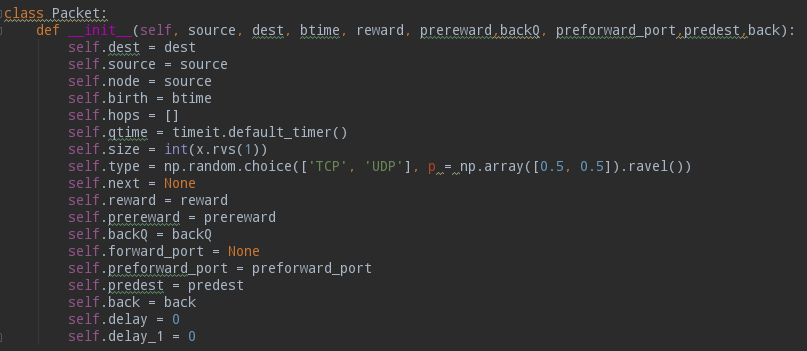
In the env.py, the network environment is defined in a class **Network**, and the packet is also defined in a class **Packet**. In the packet class, the source node of the packet (**source)**, the destination node (**dest)**, the birth time of the packet (**birth)**, the total number of hops (**hops)**, 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 in this class are defined, see figure 6.

Figure 6: The attribute definitions in the class of Packet

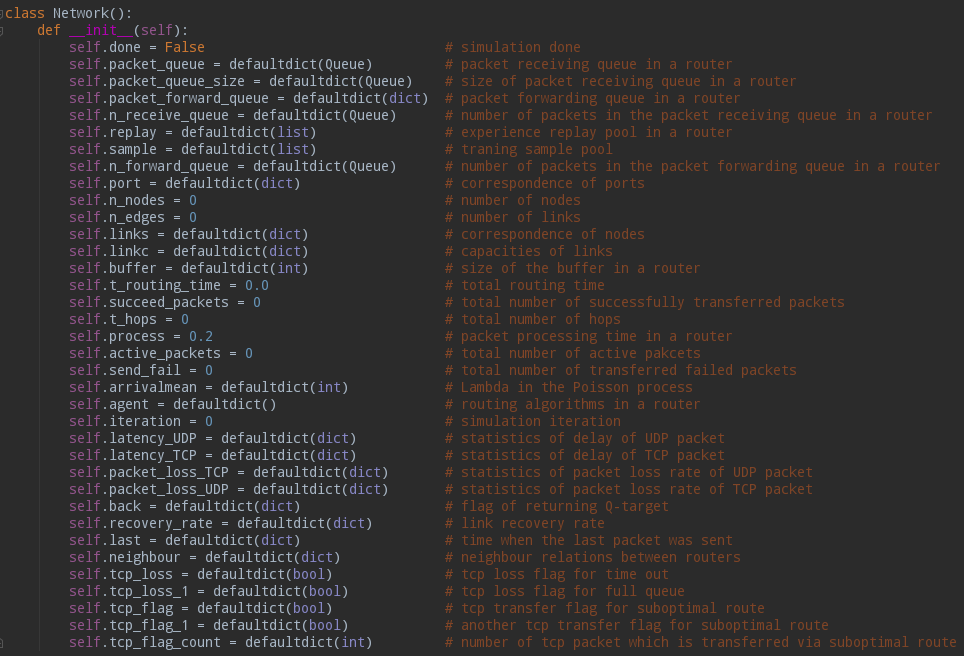
The size of the packet is subject to a truncated normal distribution, and the type of the packet is subject to a binomial distribution. In the Network class, there are more attributes defined. Among them, the important attributes are the packet receiving queue (**packet\_queue**) in each router, the packet forwarding queue in each router (**packet\_forward\_queue)**, the experience playback pool replay 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 buffer maximum (**buffer**), packet delay statistics (**UDP\_delay** and **TCP\_delay**), and packet loss rate statistics (**UDP\_loss** and **TCP\_delay**). They basically defined a complete topology of a network environment and also the basic properties of routers. The complete and detailed attributes defined in the Network class are shown in the figure 7.

Figure 7: The attribute definitions in the class of Network

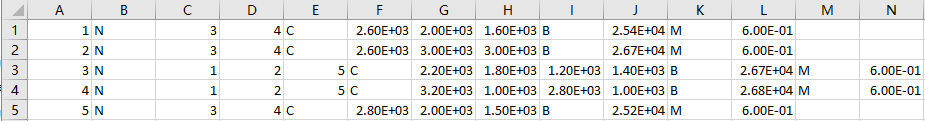
After the attribute definitions in the class of Network, the initialization of these attributes was performed subsequently in **rest**. These attributes were initialized by a CSV file which contains the information of the network environment. See figure 8, which is the CSV file used in this project.

Figure 8: The CSV file used to initialize network environment attributes

In this CSV, 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 Network class 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 length of the time interval is obtained from the initialization CSV file in the previous step. In order to balance the load of each router, each router always generates a new packet that is sent to other routers evenly, see the red box in figure 9.

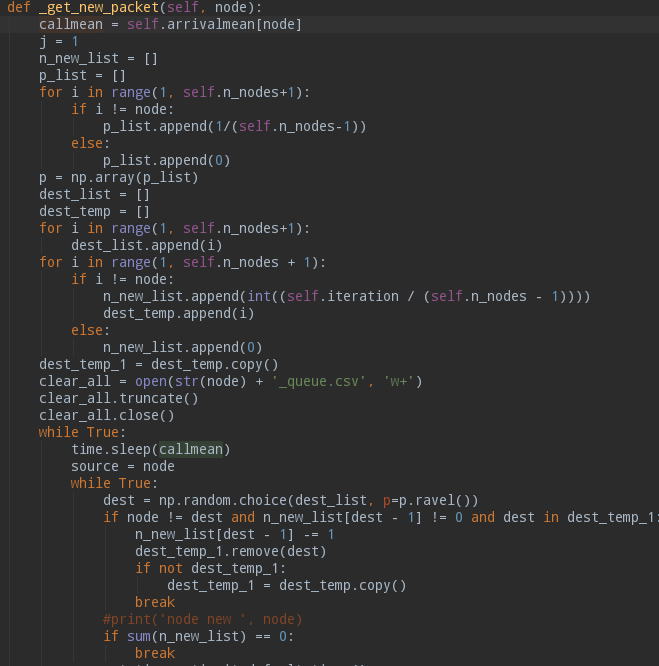
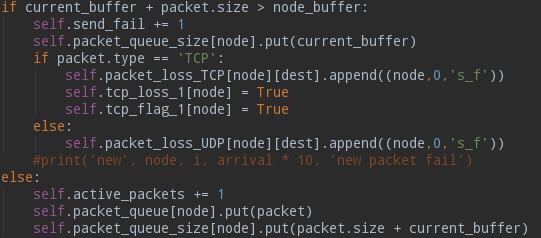


Figure 9: Part of the code of the **\_get\_new\_packet** function

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 via suboptimal route and drop a set of UDP packets to reduce TCP packet loss rate, see figure 10.

Figure 10: 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 proceeds the packet through four if statement to determine the type of the packet, see figure 11.



Figure 11: 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.

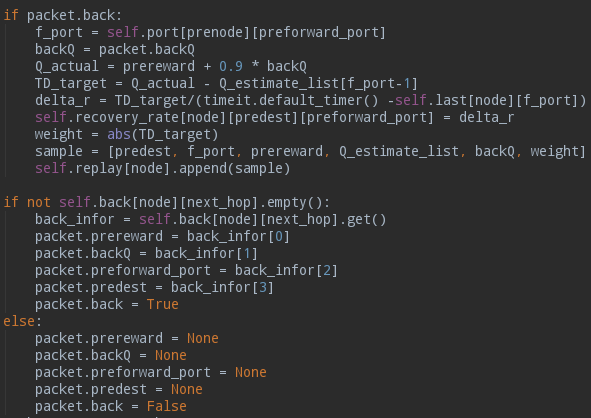
In this model, the Q-target is returned in the form of piggyback, which means that the router will send a Q-target message to the next-hop router accompanied with the forwarding packet. This Q-target message corresponds to the packet sent by the next-hop router to the local router previously. This piggyback 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 this is also a module that adds the piggyback Q-target information of next-hop router into the packet, see figure 12.

Figure 12: 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 13.

Figure 13: The learning module in **\_receivequeue**

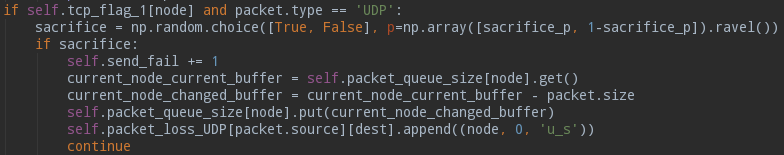
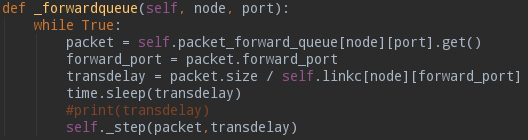
The second if statement controls the behaviour of the router when the TCP packet is lost. When the TCP packet is lost, **\_get\_new\_packet** will set up a TCP loss flag. This flag will tell the router to drop some UDP packets and forward TCP packets along the suboptimal route to save link capacity and reduce router load. **\_receivequeue** will detect this flag by an if statement and drop the UDP packets with a probability, see figure 14.

Figure 14: 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 15

Figure 15: The **\_forwadqueue** function

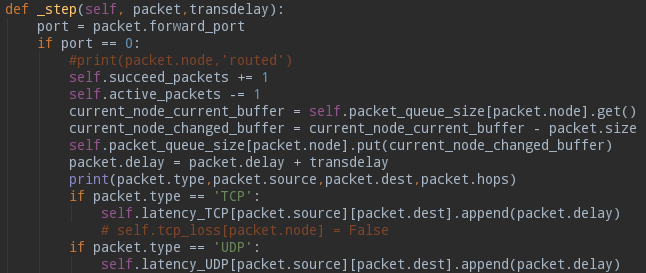
After receiving the packet, the environment processing function **\_step** will first check the forwarding port recorded in the packet. If the port is 0, it means that the packet has been delivered. So this function will record the total time spent on the packet transmission and then delete the packet from the forwarding queue, see figure 16.

Figure 16: Part of the code of **\_step**

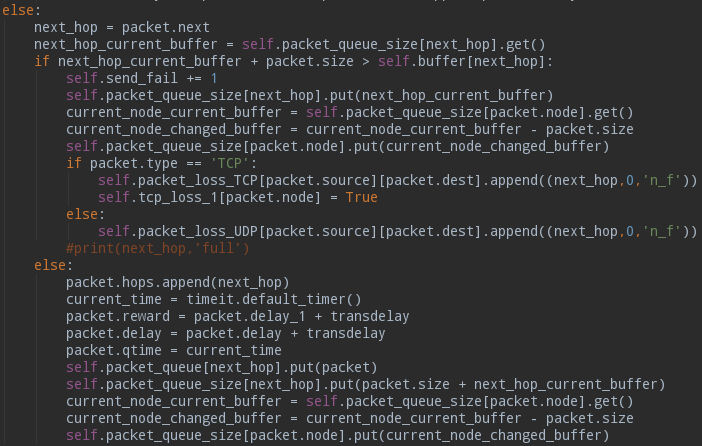
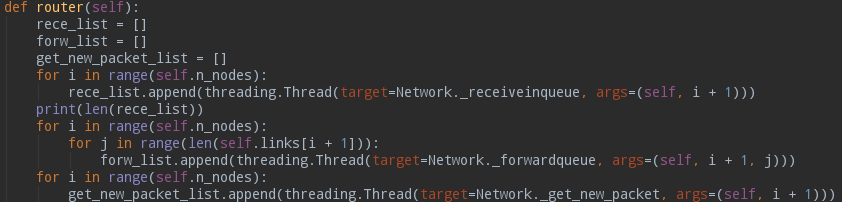
If the forwarding port of the packet is not 0, it means that the packet is sent to the next-hop router. Then this function will first check the size of the receiving queue of the next-hop router and if the next-hop router's receiving queue is full this function will drop the packet and set up the TCP 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 17.

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

Finally, after defining the three function functions that control the behaviour of the router, the router 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 18

Figure 18: The thread setting in function **router**

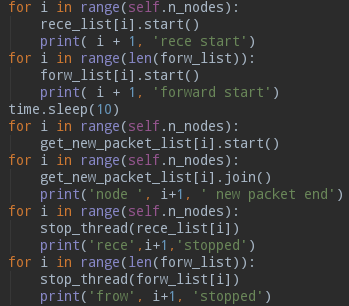
After that, the router will start these threads one by one. These threads will run until each router's **\_get\_new\_packet** completes the defined iteration. The router will call an external thread stopping function **stop\_thread** to stop each router's **\_receivequeue** and **\_forwardqueue**, see figure 19.

Figure 19: The starting and stopping of threads in **router**

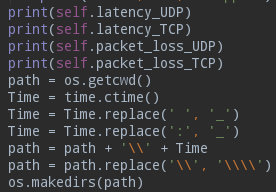
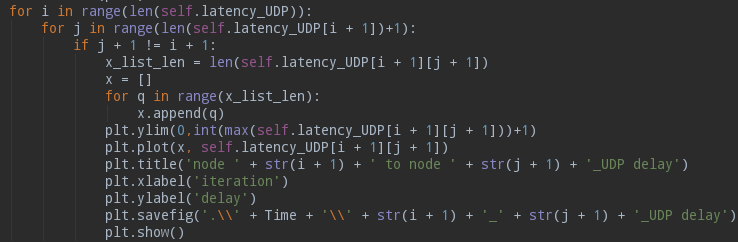
After the simulation is finished, the router will print the transmission delay and the delivery of all the packets. The router will then create a folder to store the figures which created latter. The name of the folder is defined by the time and date at the time, see figure 20

Figure 20: 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 delay figure of the UDP packet is shown in figure 21. The codes for generating other figures are same as the code in figure 21.

Figure 21: The 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, usues the function **show\_routing\_table** to print current routing table, see figure 22.

Figure 22: The utilization of class **ADQN** in **env.py**

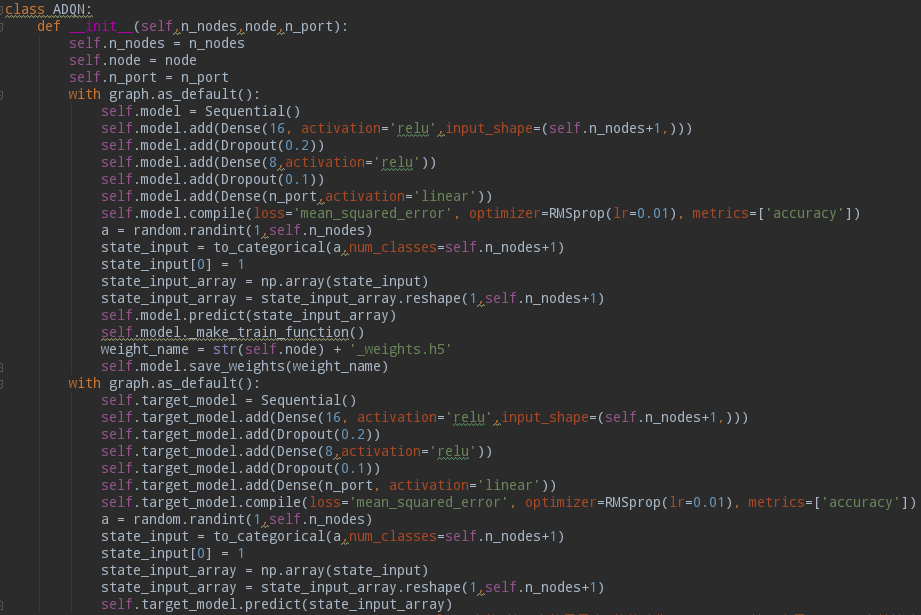
The class **ADQN** firstly uses the input number of nodes, the input number of ports and the input router serial number to define the attributes of the algorithm, which are the neural networks in the deep reinforcement learning. Because there are two networks in the deep reinforcement learning, the class **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 23.

Figure 23: The initialization of neural networks in class **ADQN**

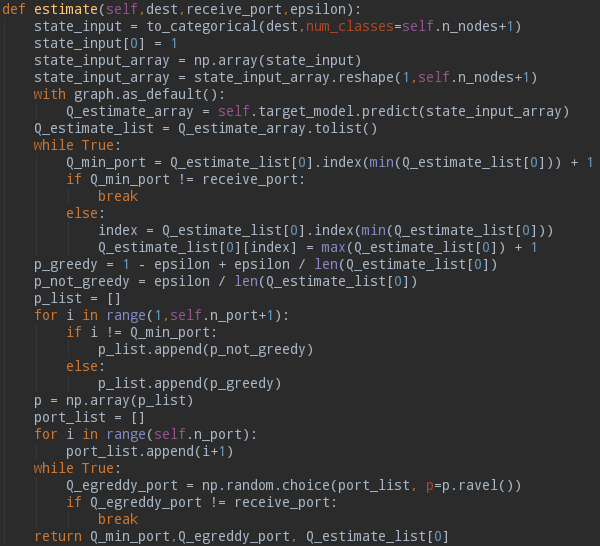
The **estimate** function selects the forwarding port of the packet based on the input destination, the input receiving port, and the input exploration rate. This function will first convert the destination node into a one-hot form, then call the prediction function of the main neural network with inputting the one-hot destination node, then get the delays from each port to the destination node. This function will find the port with the smallest delay and set it as the forwarding port of the packet. If the forwarding port is the same as the port that receives the packet, the port with the second smallest delay will be selected as the forwarding port. 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 packets back to the router that sends the packet to it, see figure 24.

Figure 24: The **estimation** function of **ADQN**

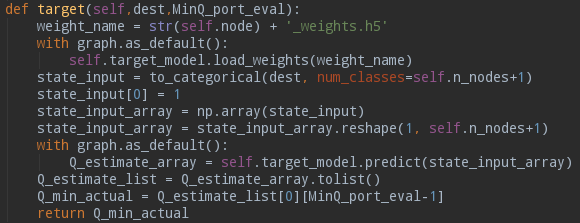
The **target** function calculates and returns the minimum state-action-value function of the target network based on the input destination node and the input minimum delay port, which is the action evaluation in the double Q-learning. The target function will first read the stored neural network weight 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 delays from each port to the destination node, this function will return the minimum delay port’s minimum state-action-value function, see figure 25.

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

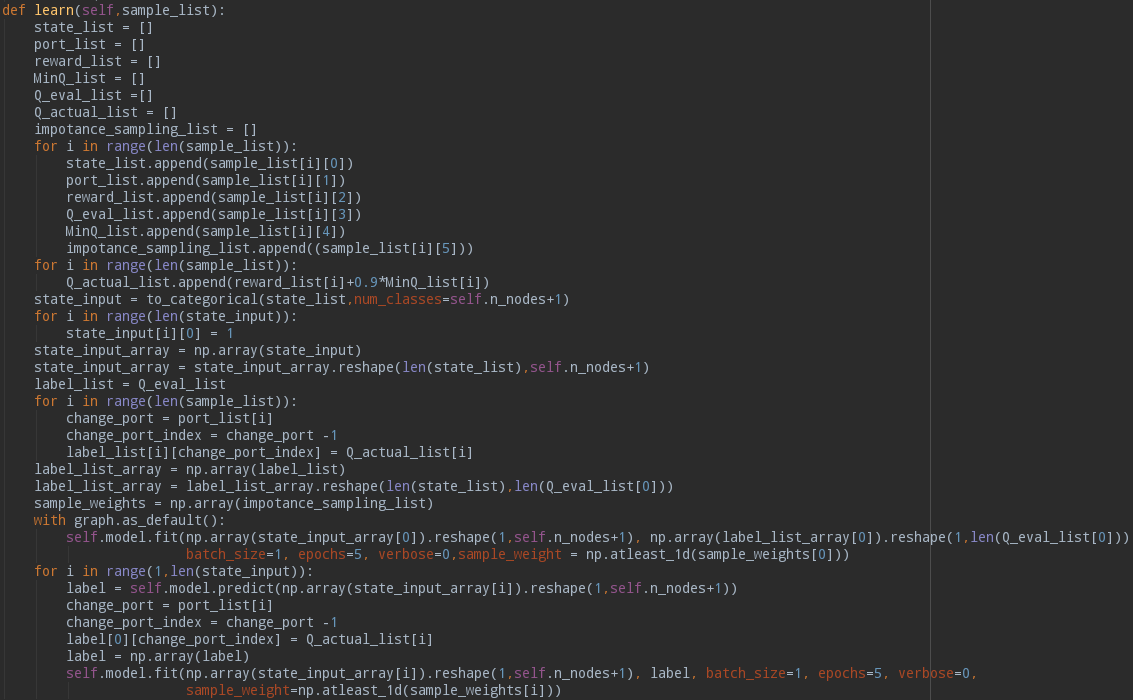
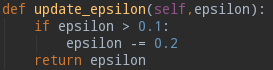
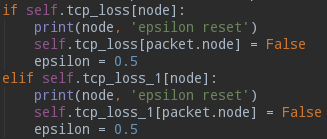
The **learn** function uses the input sample set to update the weights of the main network. This function will first read the components in the samples, which are the destinations, the forwarding ports, the one-step rewards, the current network’s Q value output list, the minimum Q values of next-hop 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 26.

Figure 26: 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 27.

Figure 27: 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 **\_receivequeu** 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 28.

Figure 28: The **update\_epsilon** function of **ADQN** and epsilon reset in **\_receivequeue**

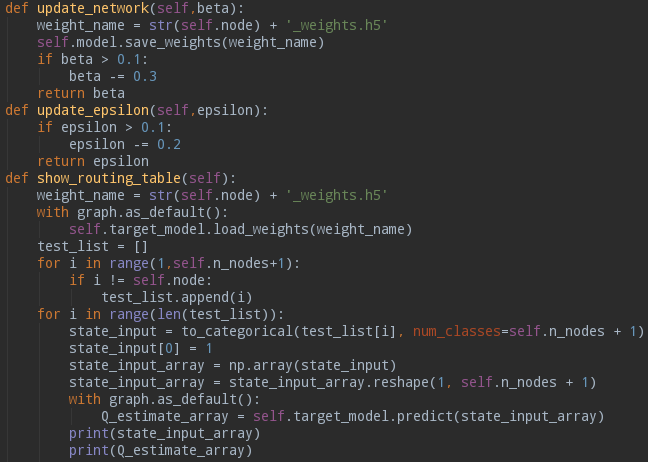
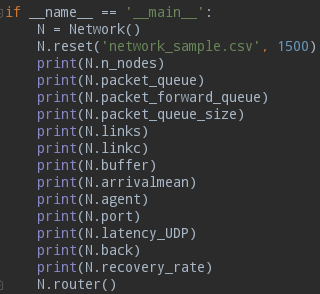
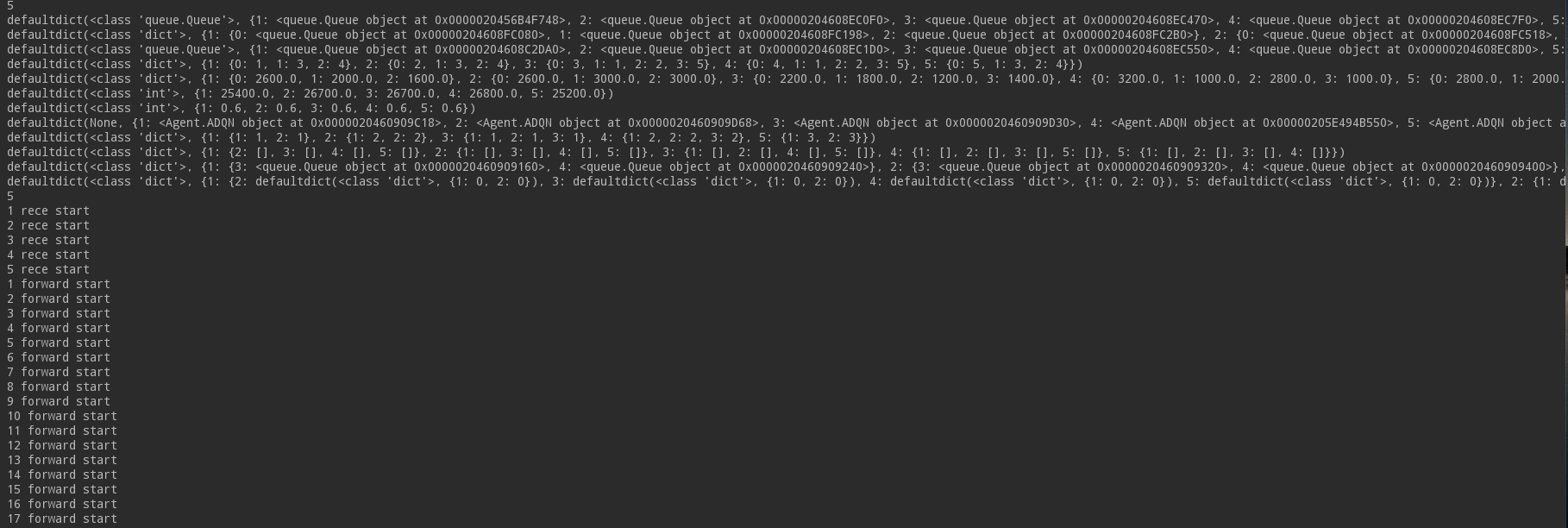
The **show\_routing\_table** function is used in **\_receivequeue** in **env.py** to print the current routing table. This function will fist reload the neural network weight h5 file and then generate an input set for all destination nodes in the form of one-hot. Next, it will call the prediction function of the target network to get the routing table, see figure 29.

Figure 29: The **show\_routing\_table** function of **ADQN**

## Analysis of Results

When the program is running, the program first uses the class **Netowrk** 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. 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 30.

Figure 30: The execution code of the programme and the initialization outputs

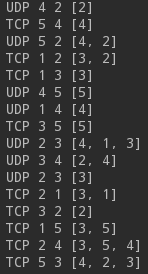
When the simulation starts, the information of the delivered packets will be printed one by one in the Pyton console, including the type of the packet, the source node and the destination node of packet, and the routers that the packet passes through, see figure 31.

Figure 31: Printing of the information of delivered packets

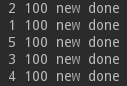
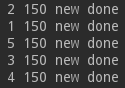
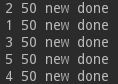
After every 50 iterations, the program will print the current iteration, see figure 32.

Figure 32: program prints the current iteration after every 50 iterations

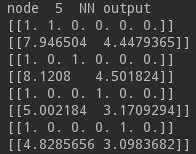
****Each time the main network updates the weights, the program will print out the current routing table of the main network, that is, the delay to each destination node, see figure 33

Figure 33: The example output routing table of the main network (node 5)

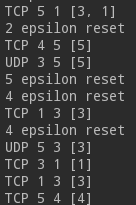
When the router detects a TCP packet loss, the router will reset the exploration rate, which is printed out in the Python console, see figure 34.

Figure 34: The resets of the exploration rate

### UDP packet analysis

1. **Conclusions**

Text here, should not be more than 2 pages.

**References**

(no limit on the number of references)

1. A. Smith and B. Jones, “Method to derive a reference list,” *Journal of References*, vol. 10, no. 3, pp. 34-38, Jan. 2010.
2. A. Smith, B. Jones, and C. Watson, “An improved method to derive a reference list,” *Proc. Internat. Conf. on References*, ICR 2011, pp. 45-50, Feb. 2011.

**Appendices**

No limit to the number of pages.

Add appendices with additional results, proofs, etc., as appropriate to your project. Listing of source codes is discouraged unless the particular portions of the source code are critical to the development of your project and they are discussed in the main part of your text.