A global multipath load-balanced routing algorithm based on Reinforcement Learning in SDN

Truong Thu Huong
School of Electronics and Telecommunications
Hanoi University of Science and Technology
Hanoi, Vietnam
huong.truongthu@hust.edu.vn

Ngo Do Dang Khoa
School of Electronics and Telecommunications
Hanoi University of Science and Technology
Hanoi, Vietnam
khoa.ndd142350@sis.hust.edu.vn

Nguyen Xuan Dung
School of Electronics and Telecommunications
Hanoi University of Science and Technology
Hanoi, Vietnam
dung.nguyenxuan@hust.edu.vn

Nguyen Huu Thanh
School of Electronics and Telecommunications
Hanoi University of Science and Technology
Hanoi, Vietnam
thanh.nguyenhuu@hust.edu.vn

***Abstract*— Routing and load balancing are two important factors that guarantee Quality of Service (QoS) in the Internet. Meanwhile Software Defined Networking (SDN) has introduced a new networking paradigm that make routing and load balancing policies implemented more quickly and flexibly in the system. In the SDN architecture, the "intelligence" of the network is centralized in the application layer and control layer instead of distributing in the network device in traditional network. Current studies using SDN-based routing and load balancing still stop at the scope of finding appropriate network resource for one single data flow but not considering network resource for all flows coming from different sources as the whole. In this paper, we propose a global load-balanced routing scheme, which can take advantages of global view of the SDN controller to make a global policy for routing and load balancing. The solution has been shown to outperform traditional algorithms and recent SDN-based artificial intelligence algorithms in terms of delay and network utilization.**

Keywords— Load balancing, SDN, global routing, Reinforcement Learning, Neural Network

I. INTRODUCTION

To assure the Quality of Service in the Internet, a good routing and load balancing scheme is required. In traditional network, OSPF and RIP routing protocols, which are based on Dijkstra and Bellman Ford algorithms respectively, are still widely applied [1]. Meanwhile, most of the current load balancing algorithms are still static, such as Equal-Cost Multipath (ECMP) [2], Valiant Load Balance (VLB) [3] and Round Robin [4] which are independent of variant traffic.

On the other hand, thanks to the introduction of Software Defined Networking SDN [5], a new approach of network management has been promted. The main idea of SDN is to separate the control layer from the infrastructure layer (layer of real devices such as router, switch, etc.). In the SDN architecture, control algorithms and schemes, instead of being implemented dispersedly in routers/switches, can be now flexibly and dynamically developed at the centralized controller which has a global view of the whole network. There are some proposed SDN-based load balancing methods, such as ANNLB [6], DNQ [7] which applied Artificial Intelligence and Machine Learning in their algorithms. However, their proposed routing and load balancing scheme are designed to optimize for each single flow under a cetain network circumstance without considering optimization for all flows coming from different sources. Our target is, at each control period, the network resource should be managed in a way that gives an appropriate policy for all possible incoming flows at once.

In this paper, we propose a SDN-based load-balanced routing scheme combining Reinforcement Learning and Deep Neural Network called RLLR to improve both routing and load balancing policies in each control period. Every flow arriving within this period will then have to follow the policy. By utilizing global information in the SDN controller, RLLR can get enough information on traffic states to establish an appropriate policy. Traffic states include traffic, links, bandwidth. To the best of our knowledge, it is the first attempt to make a SDN-based load balanced routing solution for all possible flows from all sources within the network. The emulation results have shown a promising performance in minimizing delay and improving network utilization.

The rest part of our paper is designed as follows: section II presets related work. Section II elaborates problem formulation. Then our load balancer architecture is described in Section IV. Section V describes how the RLLR algorithm works. Performance evaluation of the proposed method versus other methods are illustrated in Section VI. And finally, Section VII presents conclusion and future work.

II. RELATED WORK AND BACKGROUD

Traditionally, the algorithms are deployed distributedly among the routers, so as it is not easy to implement a dynamic scheme, thus almost algorithms in this architecture are often static. A static load balancing scheme is only based on the given topology. To the best of our knowledge, there are three widely-use static algorithms which are ECMP [2], VLB [3] and Round Robin [4]. In ECMP, it can be seen from its name that the main idea is to divide loads equally in possible paths. In VLB, the concept is to split traffic randomly. And finally, in Round Robin, paths are chosen in round. There are also some variant algorithms such as Weight Round Robin [8] but again, all of those are static and will not change no matter how the traffic varies.

The appearance of SDN opened an evolution that the implementation of an algorithm can be centralized in the

controller. With the SDN networking paradigm, we can collect network status from every SDN switch and control the network from the control layer.

There are some proposed static load balancing strategies, for example in [9][10]. However, it is clearly a static algorithm usually gives worse results than a dynamic one. Only few researchers proposed dynamic load balancing in SDN. One of the first researches on this issue is the DLB algorithm [11] in which the authors design an algorithm which will choose the path with least data transmitted, based on a greedy-like scheme. Nevertheless, this algorithm cannot take the advantage of global view in SDN. A proposed algorithm for this problem can be found in [6], in which they use a neural network to make decision. The system collects information about bandwidth ratio, packet loss rate, delay and length of paths to predict which path will have the least load. Although it achieved an impressive result that could decrease 19.3% of delay at most, some drawbacks can be pointed out. Firstly, they did not optimize directly delay or utilization but use a parameter called load condition to give decisions. Secondly, if the neural network is trained only one time or collected data in the training phase cannot generalize the traffic patterns, it cannot adapt well to the network environment. Recent researches in [12] and [7] extended the idea of using Neural Network in [6]. In [12], they collected two more factors, which are trust and overhead to calculate the load condition. In [7], they used the same Neural Network in [6] but adding a Q-learning phase to choose the best path. Although all of those approaches had made some breakthrough in dynamic load balancing, the routing scheme is still just implemented for each single flow without considering an optimization scheme for all flows coming from different sources within the network.

In summary, this paper will give a new approach which can make use of the SDN global view to improve directly delay for all flows from different sources and network utilization and can be well-adapted to the network environment.

III. Problem Formulation

In this section, we will formulate the network problem and introduce our proposed algorithm as well as the architecture of our load balancer deployed in the SDN network.

A. Network context and Notations

Traffic context

Given a network topology of n nodes, traffic generated from n nodes within the network is simulated by a matrix as follows:

$$T = \begin{pmatrix} \lambda_{11} & \lambda_{12} & \cdots & \lambda_{1n} \\ \lambda_{21} & \lambda_{22} & \cdots & \lambda_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n1} & \lambda_{n2} & \cdots & \lambda_{nn} \end{pmatrix}_{n \times n} \quad (1)$$

Where:

- n is the number of nodes in the network
- λ_{ij} is the arrival rate from node i to node j (non-existing links will be represented as 0).

Load balancing scenario

In this paper, we want to cope with the load balancing issue, it means we have to find multiple paths between each pair of source – destination nodes. It is obviously unnecessary to find all possible paths between a pair of source-destination nodes due to NP-hard complexity. Therefore, we suggest load balancing to be implemented in only "k best paths" in terms of the number of hops. In detail, with any two given nodes, we will find out best k paths with the least number of hops, and the length (or depth) of the path must be smaller than a threshold. There are two ways to definite the threshold: one is the maximum length of a path (denote as depth_max) and other is the maximum proportion of a path to the shortest path (denote as pro_max). This can be easily achieved by Breadth First Search (BFS) algorithm [13], due to the fact that the first k paths found by BFS is the least-hop paths.

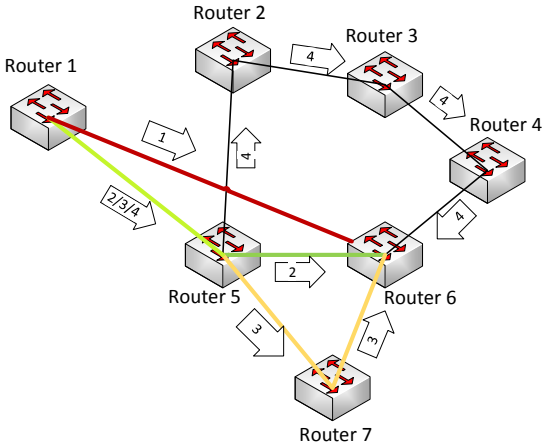

Fig. 1. An example of all paths in the NetRail Topology

In our experiment, we use the NetRail Topology described in Figure 1. As can be seen, if $k = 3$, $pro\_max = 2$, two paths from node 1 to node 6. The data structure of k shortest paths between each source and destination is dictionary. In general, given key (i, j), dictionary of path *dict_path* stores k best paths between node i and j:

$$dict\_path(i,j) = \{P_1^{(ij)}, P_2^{(ij)}, \dots, P_k^{(ij)}\} \quad (2)$$

Each $P_m^{(ij)}$ saves all nodes in the path:

$$P_m^{(ij)} = [i, i_1, \dots, i_l, j] \quad (3)$$

It means the m-th path $P_m^{(ij)}$ of the top-k shortest paths between i and j has length of $l + 1$, every packet following` $P_m^{(ij)}$ will go in order $i \rightarrow i_1 \rightarrow \dots \rightarrow i_l \rightarrow j$.

B. Control Objectives

Given a network topology, traffic conditions, k-best paths found by BFS, we want to find out a policy of load balancing in top-k best paths for every pairs of nodes (i.e. sources) that minimizes delay and improves utilization at the same time. The load balancing policy is presented by weights of each path w.

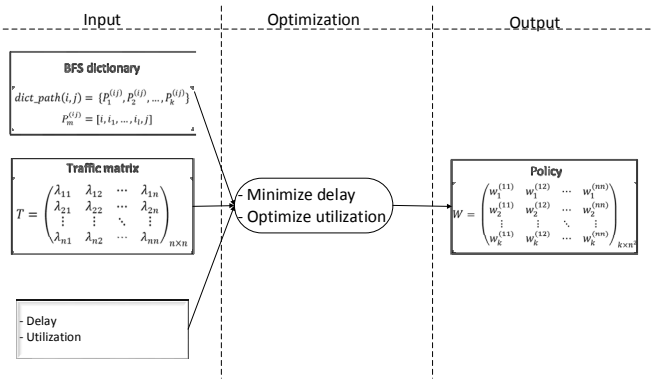

Fig. 2. Network control goal

The weights of the corresponding paths are represented in matrix *W:*

$$W = \begin{pmatrix} w_1^{(11)} & w_1^{(12)} & \cdots & w_1^{(nn)} \\ w_2^{(11)} & w_2^{(12)} & \cdots & w_2^{(nn)} \\ \vdots & \vdots & \ddots & \vdots \\ w_k^{(11)} & w_k^{(12)} & \cdots & w_k^{(nn)} \end{pmatrix}_{k \times n^2} \quad (4)$$

Where:

$(w_1^{(ij)}, w_2^{(ij)}, \dots, w_k^{(ij)})$ are weights for load balancing in k paths between *(i, j)*

The weights have to satisfy:

$$w_1^{(ij)} + w_2^{(ij)} + \cdots + w_k^{(ij)} = 1 \quad (5)$$

IV. Our Load Balancer Architecture

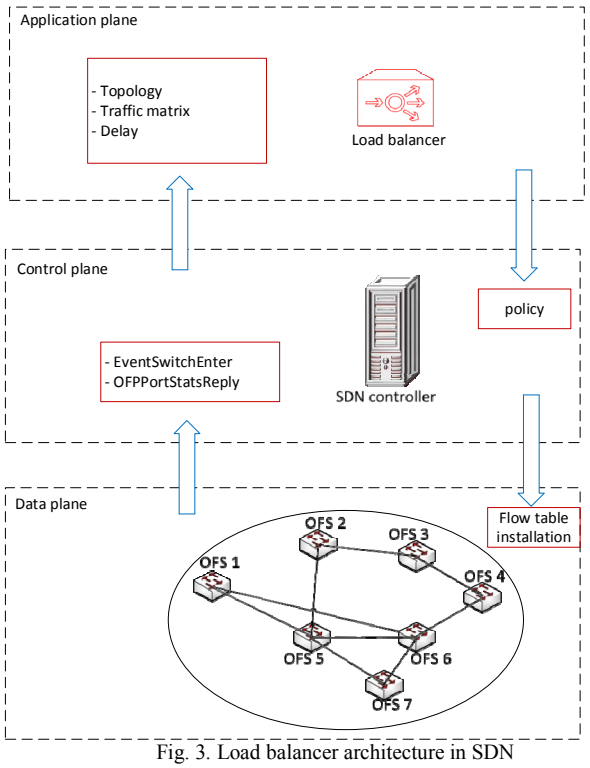

Fig. 3. Load balancer architecture in SDN

In this section, we will describe our Software-Defined Networking- based load balancing architecture. In this architecture, the SDN controller is in charge of forwarding, dropping, routing packets while the load balancer works in synchronization with but separately from the SDN controller to avoid overloading it.

The load balancer is implemented with the RLLR algorithm (described in section V) in order to balance load based on a balanced metric of delay and utilization. For it, the load balancer, therefore, needs input parameters coming from the OpenFlow switches (OFS) in the data plane such as: traffic matrix and delay periodically. In fact, traffic information (i.e. Utilization) can be acquired by parsing the OFPPortStatsReply message from OpenFlow Switches to the Controller. All Fields of this message in OpenFlow 1.3 protocol is shown as below:

Index	Fields	Description
1	port_no	Port number
2	rx_packets	Number of received packets
3	tx_packets	Number of transmitted packets
4	rx_bytes	Number of received bytes
5	tx_bytes	Number of transmitted bytes
6	rx_dropped	Number of packets dropped by RX
7	tx_dropped	Number of packets dropped by TX
8	rx_errors	Number of receive errors.
9	tx_errors	Number of transmit errors
10	rx_frame_err	Number of frame alignment errors
11	rx_over_err	Number of packet with RX overrun
12	rx_crc_err	Number of CRC errors
13	collisions	Number of collisions

Fig. 4. OFPPortStatsReply message format

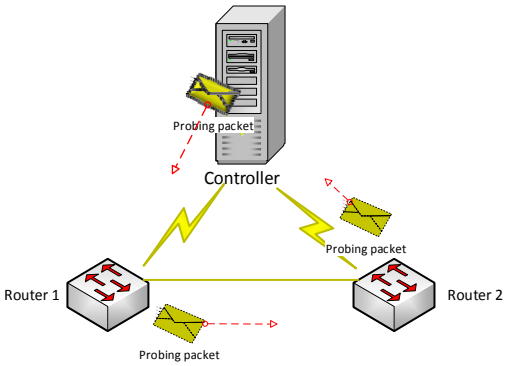

Fig. 5. Delay measurement

But information of delay is not available in the current standard OpenFlow switch since there are no current supported message or protocol to calculate the delay. To deal with this, we create a special probing packet and inject it to the measured links

whenever it is needed (as described in Figure 5). In our research, delay in the control plane is assumed to be negligible. Upon receiving the inputs of delay and utilization, the load balancer deploys the RLLR algorithm to find out a load balancing decision for the whole network.

V. RLLR ALGORITHM

In this section, we will describe our RLLR algorithm (named after Reinforcement Learning based Load-Balanced Routing). There are two main components in our algorithm: the policy maker and the estimator which are built based on two neural networks. The basic idea of RLLR is: with the current network information collected periodically, the mission of the policy maker is outputting a global load balancing strategy, then the policy will be estimated by the estimator component. Based on the estimated results, the policy maker will update their "knowledge" to make policies better and better. A neural network contains three factors: input, output and objective. Whenever we feed the input, neural network will try to give the output that optimizes the given objective.

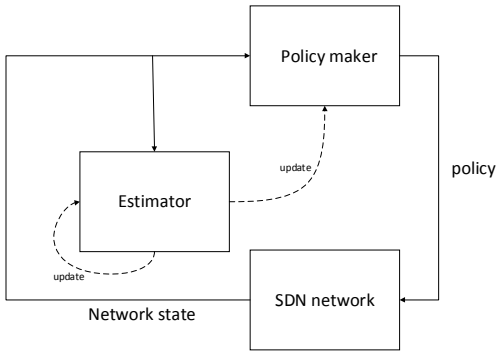

Fig. 6. Basic idea of the RLLR algorithm

Given the traffic matrix input T representing for traffic from all sources, the policy maker will output the estimated weight matrix W, each of whose columns represents the weights for k possible routes between a pair of source and destination. w_k^{ij} means weight between node i and j in k possible routes. Note that if $i = j$ then $w_k^{ij} = 0$.

This policy weight W is then evaluated by the Estimator to improve the decision for the next step. The estimator uses the Q value to back propagate to the policy maker in order to update its internal states to get the weights closer to the desired weights (i.e. outputs).

$$Q(T_t, W_t) = r + \gamma \times Q(T_{t+1}, W_{t+1}) \quad (6)$$

Where:

- T_t: the traffic condition retrieved from the previous period but representing for traffic condition in the current period t
- W_t: Load balancing policy at time t

$$W_t = W_t + \max(0, (\varepsilon - \beta t))\eta_t \quad (7)$$

- Constants $0 < \varepsilon < 1$, $0 < \beta << 1$ are the 2 constants for the initial exploration phase;
- η is the noise at the initial exploration phase. So overtime, noise η is reduced as policy W_t is improved.
- r : is the reward that is returned from the network environment after each action (i.e. a policy W_t) has been implemented.
- γ: the discount rate since immediate reward is often more important than long-term rewards

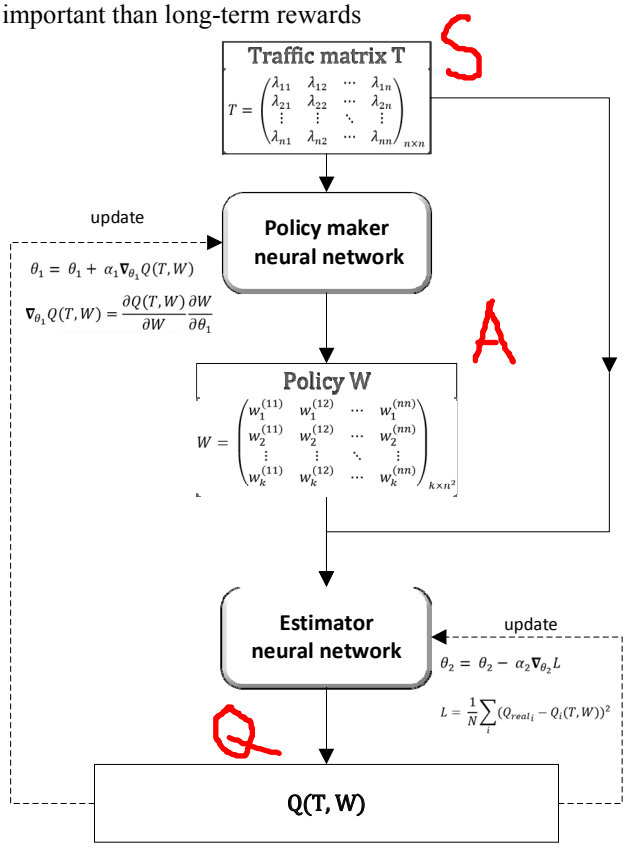

Fig. 7 Flow char of RLLR

As mentioned in equation (6), the reward function or the objective function of our load balancing strategy is defined as follows:

$$r = \frac{delay\_std}{\overline{delay}} + \delta_1 \frac{delay\_std}{std(delay)} + \delta_2[1 - std(utilization)] \quad (8)$$

Where:

- $delay\_std$ is a standard delay, which is a constant, implying the smallest delay of a link could be.
- $\overline{delay}$ is average delay in all links.
- $std(delay)$ is the standard deviation of delay which has to be small to avoid large difference between links in the network.
- δ_1, δ_2 are discount rates (smaller than 1), because average delay is priority (e.g. when load is small, there is no need to minimize $std(utilization)$).
- $std(utilization)$ is the standard deviation of utilization. In a good-policy network, the difference of utilization in

links will be small (to avoid congestion while network still remains resources).

This reward helps the system to optimize its decision in such as way that harmonize delay and utilization, taking into account that delay among different links should not be too large.

To make the calculated output of the Estimator as close as possible to the desire output, the internal states of the Estimator's neural network is updated by backpropagating the Loss function *L* (loss = desired output – actual output). In other words, the internal states should be modified to minimize the total loss function.

$$L = \frac{1}{N}\sum_i (Q_{real_i} - Q_i(T, W))^2 \quad (9)$$

Again, to avoid the risk of having to optimize millions of weights under the condition where the number of inputs of the neural network is a lot, we can use the derivative of the Loss function at a certain point that gives the rate or the speed of which this function is changing its value at this point.

$$\theta_2 = \theta_2 - \alpha_2 \nabla_{\theta_2} L \quad (10)$$

Where:

- θ_2: weights of the estimator
- α_2: learning rate of the estimator

The internal states of the policy maker are updated by using the sample policy gradient:

$$\nabla_{\theta_1} Q(T, W) = \frac{\partial Q(T,W)}{\partial W}\frac{\partial W}{\partial \theta_1} \quad (11)$$

The objective of the policy maker is to maximize Q. As a result, the weights of policy maker neural network can be updated by gradient ascent in the back propagation phase, which is inverse to the gradient descent:

$$\theta_1 = \theta_1 + \alpha_1 \nabla_{\theta_1} Q(T, W) \quad (12)$$

Where:

- θ_1: weights of the policy maker neural network
- α_1 : learning rate of the policy maker to force the weight to get updated very smoothly and slowly in order to avoid big steps and chaotic behavior.

The whole algorithm can be summarized in the pseudo code as follows:

Algorithm 1 RLLR algorithm

1. Initialize policy maker and estimator neural network with random weights θ_1 and θ_2.
2. Initialize learning rate α_1, α_2 for 2 neural network.
3. Initialize Buffer.
4. Initialize $0 < \varepsilon < 1$, $0 < \beta << 1$ (exploration parameters).
5. **In period *t-th* (*t* > 1):**
6. Initialize random process noise η for exploration.
7. Policy in period t. Calculate
$$W_t = W_t + max(0, (\varepsilon - \beta t))\eta_t$$
8. Measure reward r_{t-1} and traffic matrix T_t
9. Add transition (T_{t-1}, W_{t-1}, r_{t-1}, T_t) to Buffer.
10. Randomly sample a batch of N transitions (T_i, W_i, r_i, T_{i+1}) from Buffer.
11. Set $Q_{real_i} = r_i + \gamma \times Q(T_{i+1}, W_{i+1})$
12. **for** *i* = 1, M **do**:
13. Update estimator network by minimizing loss function $L = \frac{1}{N}\sum_i (Q_{real_i} - Q_i(T, W))^2$ and do gradient descent $\theta_2 = \theta_2 - \alpha_2 \nabla_{\theta_2} L$.
14. Update policy maker network by using the chain rule $\nabla_{\theta_1} Q(T, W) = \frac{\partial Q(T,W)}{\partial W}\frac{\partial W}{\partial \theta_1}$ and do gradient ascent $\theta_1 = \theta_1 + \alpha_1 \nabla_{\theta_1} Q(T, W)$.
15. **end for**
16. **end of period *t-th***

VI. PERFORMANCE EVALUATION

In this section, we will conduct an emulation to evaluate the efficiency of our RLLR algorithm versus 3 other algorithms Shortest Path [1], Round Robin [4], ANNLB [6].

A. Experiment set up

Tools: In our SDN network context, we use the Ryu/SDN controller [14] and emulate the SDN network of Openflow switch by Mininet [15]. We use D-ITG tool [16] to generate traffic since D-ITG supports four heavy-tailed distribution, include: Pareto, Weibull, Cauchy and Gamma.

Network Topology: We will build up the Netrail topology presented in seconds. In the three load balancing algorithms (except Shortest Path), we choose *k* = 3, which mean load balancing will be carried out in 3 paths for each source – destination pair.

Traffic patterns: Research in [17] [18] proposed using heavy-tailed distribution which could mimic the patterns of traffic in the Internet. In our scenario, we used Pareto and Weibull distributions with different shapes and scales (2 parameters of them) to generate different levels of load ranging from 10% to 100%.

B. Network peformance

We first show performance in average delay of the four algorithms. As can be seen in Figure 8, in the range of 10% to 20% of load, Shortest Path, ANNLB and RLLR have nearly the same delay and are smaller than of Round Robin. It can be explained that these three algorithms almost choose a shortest path for routing, which is usually the best choice when load is small. In the range of 30% to 70%, two dynamic algorithms show more and more efficiency in minimizing delay meanwhile

Shortest Path becomes the worst, since all the traffic concentrates upon the shortest path. When load is higher than 80%, our RLLR makes a breakthrough and outperform the ANNLB, with the largest improvement of 75ms compared to ANNLB.

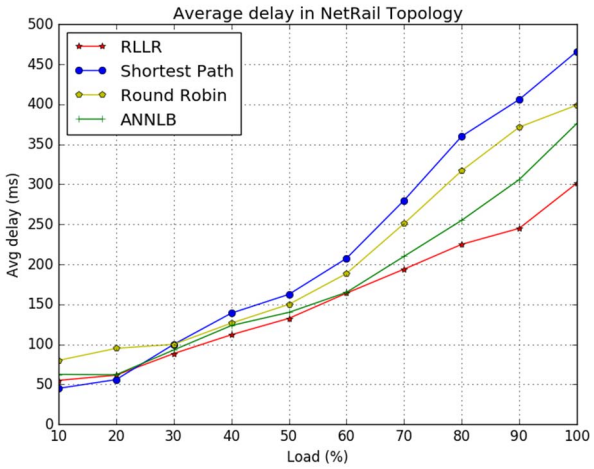

Fig. 8. Average delay comparison

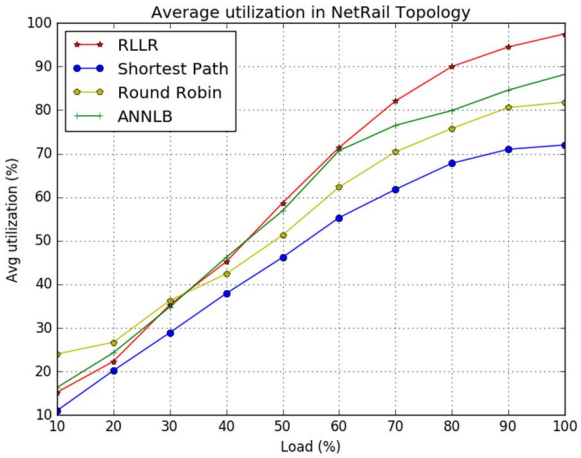

Fig. 9. Average utilization comparison

Figure 9 shows the consequence of minimizing the standard deviation of utilization which is different from minimizing average utilization. Since the aim of load balancing is often applied in high load condition; we focus on the condition where load is higher than 60%. As the Shortest Path algorithm only considers fixed shortest paths, the average utilization is the smallest which results in big queueing delays. Two dynamic load balancing algorithms utilized resource better than two others, especially our RLLR, which explain the cause of small delay

VII. Conclusion and Future Work

In our paper, we have presented our RLLR algorithm based on Reinforcement Learning and Deep Neural Network. Our algorithm is proven to take advantages of global view of SDN controller to make a global scheme in routing and load balancing better in terms of delay and utilization. In the future, we will consider improving more QoS parameters like jitter, packet loss rate, etc. and conduct experiments in more different network contexts.

Acknowledgment

This research is supported by the Office of Naval Research Global (grant number: N62909-17-1-2003)

References

[1] W. Tanebaum, Book: Computer networks. ISBN-13: 978-0132126953. Publisher: Pearson; 5 edition (October 7, 2010)

[2] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," in *RFC 2992, IETF*, 2000.

[3] W. J. D. a. B. Towles, "Principles and Practices of Interconnection," Morgan Kaufmann Publisher, 2004.

[4] D. M. E. Mustafa, "Load balancing algorithms Round Robin (RR), Least-Connection, and least loaded efficency," in *ISSN 1512-1232*, 2017.

[5] W. Xia, Y. Wen, C. H. Foh, D. Niyato and H. Xie, "A Survey on Software-Defined Networking," *IEEE Communications Surveys & Tutorials,* vol. 17, no. 1, pp. 27 - 51, 2015.

[6] X. Y.-b. Cui Chen-xiao, "Research on Load Balance Method in SDN," *International Journal of Grid and Distributed Computing,* Vol. 9, No. 1, pp. 25-36, 2016.

[7] C. Yu, Z. Zhao, Y. Zhou and H. Zhang, "Intelligent Optimizing Scheme for Load Balancing in Software Defined Networks," in *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, Sydney, NSW, Australia, 2017.

[8] D.-C. Li and F. M. Chang, "An In–Out Combined Dynamic Weighted Round-Robin Method for Network Load Balancing," *The Computer Journal,* vol. 50, 2007

[9] H. Y, W. W, G. X, Q. X and C. S, ""BalanceFlow: Controller load balancing for OpenFlow networks," *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on IEEE,* pp. 780-758, 2012

[10] D. B. a. J. R. R. Wang, "OpenFlow based server load balancing gone wild," Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services USENIX Association, pp. 12-12, 2011

[11] Y. L. D. Pan, "OpenFlow based load balancing for Fat Tree networks with multipath support," in *Proc. 12th IEEE International Conference on Communications (ICC'13)*, Budapest, Hungary, 2013

[12] S. Patil, "Load Balancing Approach for Finding Best Path in SDN," in *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, Coimbatore, India, 2018

[13] N. Karumanchi, Data structure and algorithms made easy

[14] "Ryu," [Online]. Available: https://ryu.readthedocs.io/en/latest/index.html.

[15] "Mininet," [Online]. Available: http://mininet.org

[16] W. d. D. A. D. S. A. a. A. P. Alessio Botta, "D-ITG 2.8.1 Manual," 28 10 2013. [Online]. Available: http://www.grid.unina.it/software/ITG/manual/http://www.grid.unina.it/software/ITG/manual/

[17] L. L. Mark E. Crovella, "Long-Lasting Transient Conditions in Simulations with Heavy-TailedWorkloads," in *IEEE Xplore*, 1998

[18] M. Z. T. D. N. Ronald G. ADdle, "Broadband Traffic Modeling: Simple Solutions to Hard Problems1998," *IEEE Communication Magazine,* pp. 88-95, 1998