Deep Q-Networks based Auto-scaling for Service Function Chaining

Doyoung Lee, Jae-Hyoung Yoo, and James Won-Ki Hong

Department of Computer Science and Engineering, POSTECH, Pohang, Korea
{dylee90, jhyoo78, jwkhong}@postech.ac.kr

***Abstract*—Network function virtualization (NFV) is a key technology of the 5G network era. NFV decouples a network function from proprietary hardware so that the network function can operate on commercial off-the-shelf (COTS) servers as a form of virtual network functions (VNFs). Owing to the advantage of NFV, network functions can be applied dynamically to the networks. However, NFV complicates network management because this technology creates numerous virtual resources that should be managed. To solve the problem of complicated network management, studies on applying artificial intelligence (AI) to the NFV-enabled networks, i.e., VNF life cycle management, have attracted attention. In particular, auto-scaling, which is one of the essential functions of VNF life cycle management, adds or removes VNF instances to meet service requirements. It is a challenging task to determine the optimal number of VNF instances in dynamic networks, satisfying service requirements. In this paper, we propose a novel auto-scaling method using reinforcement learning (RL) for scale-in/out of multi-tier VNF instances, i.e., service function chaining (SFC) in NFV environments. The proposed approach defines RL's states using a status of SFC composed of multi-tier VNF instances and uses service level objectives (SLO) to make a reward model. We validate the proposed approach in an OpenStack environment, and it shows that our proposed auto-scaling method provides the optimal number of VNF instances in each tier while minimizing SLO violation.**

***Index Terms*—Network Function Virtualization (NFV), Auto-Scaling, Reinforcement Learning (RL), Scale in/out, Artificial Intelligence (AI)**

I. INTRODUCTION

With the advent of the 5G network era, it is required to flexibly build and manage networks to meet rapidly changing service requirements. To this end, network function virtualization (NFV) and software-defined networking (SDN) have emerged as key technologies to enable service providers to provision new services dynamically. NFV decouples a network function from proprietary hardware so that the technology provides network functions as software, which are virtual network functions (VNFs). When networks require network functions, VNFs can be instantiated on commercial off-the-shelf (COTS) servers because VNFs operate on virtual machines (VMs) or containers hosted on the servers. Therefore, NFV can reduce OPEX and CAPEX for network management. However, despite the advantage of the NFV, there are still challenges to apply NFV to the networks. For instance, NFV complicates network management because it creates numerous virtual networks and functions, which should be managed by operators. It is hard for human operators to manage the networks manually because the on-demand virtual resources can be created dynamically. Accordingly, studies on applying artificial intelligence (AI) to complicated network management have attracted attention.

Due to the development of AI technology, various AI algorithms have been applied to network management. For example, machine learning algorithms are used for traffic classification, anomaly detection, intrusion detection, and so on [1]. Although the studies propose a direction on how to manage complicated networks using AI, there still have been issues. One of the critical issues is that machine learning approaches must have big data for training. In other words, it should store huge data volume for the training, and the data should be pre-processed, i.e., labeling, used for machine learning. Besides, it is a challenge to prepare network data representing dynamic conditions. Reinforcement learning (RL) can be used for network management to overcome the problem.

RL is well-suited to the dynamic networks because it does not require a priori knowledge or data. In particular, RL can be applied to the VNF life cycle management required in NFV-enabled networks. The VNF life cycle management provides functions such as VNF deployment, service function chaining (SFC), anomaly detection, and auto-scaling to operate VNF instances well. Among those functions, auto-scaling is an essential function to provide the ability and knowledge to add/remove the required amount of resources in response to changes in demand. In other words, determining the optimal number of VNF instances, which can run in VMs or containers, is defined as scale-in/out of auto-scaling.

Traditional methods use threshold-based auto-scaling, which adds or removes VNF instances in case of service level objectives (SLO) violation. However, it is difficult to determine the threshold value by a human because this value can be significantly affected by various conditions such as available resources in the networks and network status. Besides, many services in NFV-enabled networks are provided through SFC, in which VNF instances are distributed to multi-tier. Hence, when auto-scaling is required, it is required to consider not

978-3-903176-31-7 ©2020 IFIP

only the total number of VNF instances in SFC but also the optimal number of VNF instances in each tier. Therefore, RL can be used for the auto-scaling to provide the optimal number of VNF instances while minimizing the SLO violation.

In this paper, we propose an RL-based, deep Q-networks (DQN), auto-scaling method to determine the optimal number of VNF instances, and the approach is validated in an OpenStack environment. The proposed way defines the auto-scaling model, which includes states, actions, and rewards to set an optimal auto-scaling policy. According to our evaluation, the proposed auto-scaling method not only provides the optimal number of VNF instances but also minimizes SLO violation. Our contributions can be described as follows.

- **Deep Q-networks (DQN) model for auto-scaling**: Defining each state, action, and reward for auto-scaling. We propose a DQN-based auto-scaling method, considering which tier is scaled and which node is used for scaling.
- **Reward model for auto-scaling**: Reward definition to deal with the number of running VNF instances in SFC and SLO violation. Our proposed reward model considers not only the number of VNF instances but also the SLO in terms of response time.
- **Development of an auto-scaling module as open-source**: open-source module interacting with an OpenStack environment. We develop an auto-scaling module by considering ETSI standard MANO architecture [2] and publish the source on a GitHub page [3].

The remainder of this paper is structured as follows. In Section II, we describe previous studies that have been conducted to apply machine learning and RL to auto-scaling. We give a detailed description of the proposed method in Section III, covering deep Q-network (DQN), and reward models for finding an optimal auto-scaling policy. Based on this, we highlight the characteristics of the auto-scaling module implementation in Section IV. In Section V, we evaluate our proposed method in terms of SLO violation. Finally, we summarize the paper and discuss possible future work in Section VI.

II. Related Work

In dynamic networks, it is an essential requirement to provide various services while meeting SLO. Many studies have been conducted to deploy VNF instances for meeting the service requirement. In particular, deciding the optimal number of VNF instances is related to both VNF deployment and auto-scaling. To make clear the difference between those functions, we define that VNF deployment determines the initial number of VNF instances or selects optimal nodes to deploy VNF instances.

To address the VNF deployment problem, the authors of [4], [5] use integer linear programming (ILP). The ILP-based solutions generate ground-truth labels for temporally dynamic request traces and use them to train supervised learners that can predict VNF deployment decisions. The decisions can be increasing, decreasing, or maintaining VNF instances. However, the studies have not dealt with VNF placement. To handle the placement of VNF deployment, the authors of [6] use machine learning with graph neural network (GNN) [7]. The study conducts training labeling data generated by ILP and predicts the processing time of different VNF types. According to the study, machine learning approaches can be used to decide the optimal number of VNF instances and VNF placement. Besides, various studies have been conducted to deploy VNF instances by considering the resource capacities of physical servers and traffic rates [8]–[10].

Auto-scaling is a function to re-allocate resources such as CPU, memory, and disk to VNF instances (scale-up/down) or resize the number of VNF instances (scale-in/out) while VNFs are running. It is hard to determine how much resources or instances are needed to meet each service requirement. Therefore, the authors of [11] propose an application-agnostic auto-scaler that requires minimal application knowledge and manual tuning. They use neural networks to build a performance model of application and leverage a linear regression algorithm to predict a post-scaling state. Further, there have been studies that propose a performance model to predict resource usages for auto-scaling [12]–[14]. Besides, it is also hard to decide manually when auto-scaling should trigger. To solve the problem, the authors of [15] propose a fully automated workflow for resource allocation of the VNF instance. The study provides a Q-learning based auto-scaling method. Q-learning is one of the RL algorithms, and many recent studies have used various RL algorithms for auto-scaling in NFV environments.

In NFV environments, services are provided through SFC composed of multi-tier VNF instances. Because SFC consists of various VNF types an auto-scaling method applied to NFV environments should consider the number of VNF instances in each tier. To deal with the problem, the authors of [16] propose a way of predictive auto-scaling of multi-tier applications. The proposed approach uses supervised learning to identify the appropriate resource provisioning for multi-tier applications based on the prediction of the application response time and the request arrival rate. Further, the authors of [17] propose a vertical auto-scaling method, i.e., scale-up/down to meet the service requirement for multi-tier web applications. The approach uses Q-learning for adaptive resource allocation such as CPU and memory. Q-learning is a simple and effective algorithm to solve a Markov decision process (MDP) problem. However, Q-learning defines states and actions by tabular representation, so the table size increases extremely when the MDP problem, auto-scaling in this paper, is complex. To solve this problem, the authors of [18] propose an auto-scaling method using deep RL. Because deep RL leverages neural networks to take any number of possible states instead of tables, it can be used to address a complex auto-scaling problem.

With the attraction of NFV, many NFV frameworks have been proposed [19]–[24]. However, those studies lack pro-

viding an auto-scaling function using AI. Besides, most commercial cloud providers offer only reactive auto-scaling methods based on a threshold, which are required manual tuning by operators [25]. This reactive auto-scaling checks SLA violation by comparing a pre-defined threshold value to measured performance metrics and triggers scaling in case of SLA violation. Owing to the simplicity of threshold-based auto-scaling, this approach has become popular. However, it is difficult to define an appropriate threshold for auto-scaling in complicated NFV environments. Therefore, it is required to not only develop an AI-based auto-scaling method but also implement an NFV framework regarding the European telecommunications standards institute (ETSI) NFV architecture to support the method.

While the above studies have effectively suggested a direction on how to apply AI algorithms to the resource allocation to VNF instances in NFV environments, most of the studies validate their approaches by simulations. Besides, they have not deeply dealt with how to define the reward of RL. Because a reward model of RL significantly affects the performance, it should be considered in more detail. Further, there is still a lack of AI-enabled framework compliant with ETSI standards. Therefore, future auto-scaling studies with RL should consider not only the issue of allocating resources to VNF instances but also the implementation in a form referring to the ETSI NFV reference architectural framework.

III. Proposed approach

In NFV environments, various services are provided through service function chaining (SFC) composed of multiple VNFs. SFC consists of multi tiers, and different type's VNF instances can be placed in each tier. For example, SFC composed of 2-tier can have a firewall in the first tier, and an IDS in the second tier. Besides, each tier can have multiple VNF instances. When traffic passes an SFC path, a load-balancer in each tier distributes the traffic to VNF instances as shown in Fig 1. In this multi-tier scenario, it is required to apply auto-scaling to an SFC while considering each tier's status because if one of the tiers occurs a bottleneck, SLO such as throughput and response time can be violated. Therefore, we propose a deep Q-networks (DQN)-based auto-scaling approach and a method selecting a tier to be scaled.

A. Deep Q-networks (DQN) Model for auto-scaling

A goal of our proposed approach, an RL-based auto-scaling method, is to automate making a scaling decision. Thus, an auto-scaling problem should be an RL model that consists of state, action, and reward to be solved by the proposed method. However, it is a challenge defining the RL model because numerous conditions that complicate the model can be candidates. For instance, dynamic network metrics and low-level data about VNF instances such as CPU usage and memory usage can be used. Besides, if various conditions are used, the number of states extremely increases, and it complicates an RL problem. To define and solve the RL-based auto-scaling problem effectively, we use deep Q-networks (DQN) with a target network and replay memory proposed in [26].

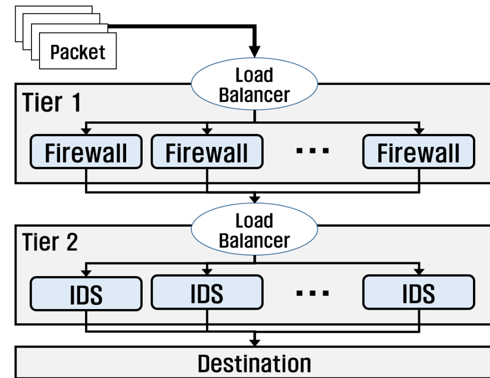

Fig. 1. Multi-tier architecture VNFs in SFC

DQN used in this paper is one of the popular deep RL algorithms, which consists of RL agent, replay memory, and an environment as shown in Fig 2. An RL agent is responsible for finding an optimal policy to solve an RL problem. A policy of an RL problem determines an action to be performed in each state. In other words, an RL agent of auto-scaling should determine an optimal scaling action from a given state. An agent interacts with NFV environments, in which physical nodes and VNF instances are running. When an RL agent chooses an auto-scaling action, it adds, removes, or maintains VNF instances in the environments and gets a new state and reward.

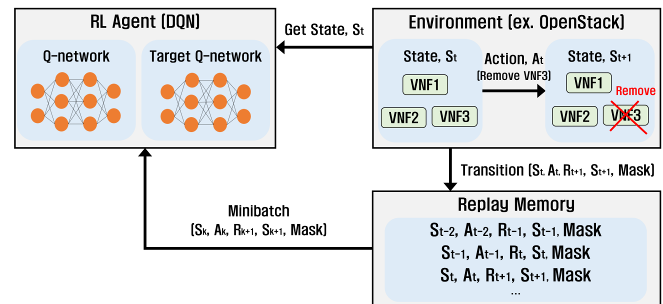

Fig. 2. Overall Deep Q-networks architecture for auto-scaling

Whenever an agent carries out a scaling action, this agent stores a *transition* that is a tuple composed of a *current state*, an *action* performed in the current state, *reward* of the action, a *new state*, and a *mask* into a replay memory. The *mask* is a value to check whether scaling successes or not. In general, training data for RL is collected sequentially over time from an environment, so this sequential data has a high correlation. If an agent learns this data sequentially, training will be unstable due to the high correlation of input data. To solve this problem, an agent uses transitions stored in the memory by mini-batch sampling. This approach prevents the correlation of data sets and enables the DQN model to learn well.

DQN uses Q-values to present an optimal policy for given states. Each action has a Q-value, and an agent chooses an action, which has the maximum Q-value in a given state. Thus, an agent should store Q-values and update them to be used as an optimal policy. In general, Q-values can be stored in a Q-table as a tabular representation. However, this table is inadequate for auto-scaling applying to an SFC. In particular, a tabular representation cannot be used to define many states because the table size infinitely increases in a complex auto-scaling problem.

$$L_i(\theta_i) = E_{(s,a,r,s')}[(r + \gamma max Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)^2] \quad (1)$$

Accordingly, DQN uses neural networks as a function applicator instead of the tabular representation. Besides, we use a target Q-network proposed in [26] for stable DQN training. DQN learns input data, $E(s, a, r, s')$ from a replay memory and sets network parameters of neural networks using a back-propagation algorithm to minimize a loss function shown in an equation 1. Q-value at iteration *i* is updated by the formula, in which γ is a discount factor, θ_i are the parameters of the Q-network at iteration *i* and θ_i^- are network parameters used to compute the target value.

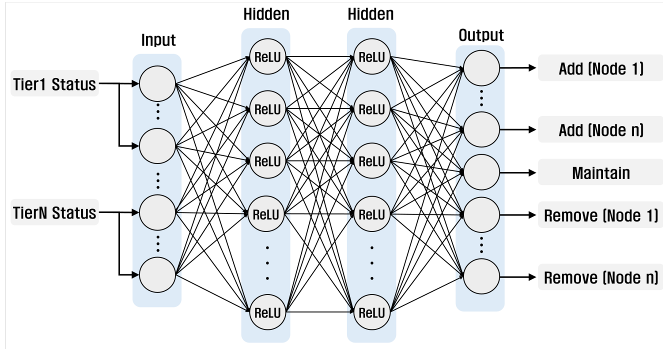

Fig. 3. Deep Q-networks model for auto-scaling

In our proposed approach, we define a state of an auto-scaling problem using each tier's status and input the state to DQN as shown in Fig. 3. The proposed DQN model has two hidden layers and uses ReLU (Rectified Linear Unit) as an activation function. A tier's status consists of 5 elements, which are the utilization of CPU and memory, the number of disk operations, size, and distribution value. Among them, the size is the number of VNF instances placed in the tier, and the distribution value presents how many nodes host the VNFs in the tier out of the total nodes. CPU utilization affects the response time of SFC because a VNF instance consumes CPU resources to process packets. Although memory utilization does not smoothly effect on packet processing, high memory utilization increases packet processing time as well because it occurs swapping activity [17]. During the activity, OS tries to free memory by saving pages to a disk. Since the speed of the disk operation is slower than memory it negatively affects the response time of SFC. Therefore, the number of disk operations is also used to define a state of an auto-scaling problem. When DQN gets the state, this network outputs one of the auto-scaling actions, i.e., adding, removing, or maintaining. Besides, the proposed approach chooses one of the tiers to apply the selected action.

B. Reward model for auto-scaling

In general, if many VNF instances are running and processing traffic by load-balancing, it helps to meet SLO. However, it can be an over-provisioning problem that consumes resources inefficiently in NFV environments. Therefore, the proposed auto-scaling method not only minimizes the SLO violation rate but also allocates the right amount of resources to operate an SFC. In this paper, we use mean response time as the SLO. Although the same number of VNF instances is in a tier, the SLO measured in the tier can be different because of VNF placement. For example, each tier's VNF instances distributed in many nodes can make the response time measured through the SFC is highly variable due to different packet forwarding paths. Therefore, it is necessary to consider placing the VNF instances of the same tier in SFC on nodes, which are close to each other. In other words, if VNF instances in SFC are placed regarding VNF placement, it can guarantee SLO effectively than that many instances are highly distributed in an environment. Accordingly, we use node utilization and density of VNF instances to define a reward model as shown in equation 2. The proposed reward model is helpful to not only maintain the appropriate number of VNF instances in SFC but also use a small number of nodes hosting those instances.

$$Node_{Util.} = \frac{Node_{used}}{Node_{total}}, \; Dens._{VNF} = \prod_{i=1}^{n} \frac{VNF_{node_i}}{VNF_{total}} \quad (2)$$

(Note that $node_i$ is a node hosting at least one VNF instance)

Node utilization ($Node_{Util.}$) shows how many nodes host VNF instances of an SFC, and it is calculated as the number of nodes hosting the VNF instance ($Node_{used}$) out of the total number of available nodes ($Node_{total}$). The density value ($Dens._{VNF}$) shows how much VNF instances in SFC are dense in the environment. For example, this value increases when many VNF instances are running on a small number of nodes. To obtain the density of VNF instances, we first calculate values that the number of VNF instances running in each node (VNF_{node_i}) out of the total number of VNF instances in SFC (VNF_{total}). In this calculation, nodes hosting at least one VNF instance are used only. Finally, we multiply every value to obtain the density of VNF instances of SFC.

With node utilization and the density of VNF instances, we also use a response time measured through an SFC path (*resTime*) to calculate a reward value. However, measured response time can have a large deviation so negatively affect the convergence of the auto-scaling model. Therefore, we use a ratio of measured response time to pre-defined SLO. As a result, the reward model is defined as shown in equation 3.

$$Reward = -\frac{resTime}{SLO} + \alpha Dens._{VNF} \times e^{-\beta Node_{Util.}} \quad (3)$$

In the reward model, a reward is obtained by the sum of the pre-processed response time and the value of an exponential function, $e^{-x}(x > 0)$. α and β are weights to adjust the value. Owing to the exponential function tuned by node utilization and the density of VNF instances in SFC, a reward value increases when only a few nodes host VNF instances to operate an SFC, or those instances are placed concentrated on a small number of nodes. In other words, the proposed reward model considers not only how many nodes and VNF instances are used to operate SFC but also performance in terms of response time measured through SFC.

C. Auto-scaling for service function chaining

When DQN makes an auto-scaling decision, an agent should choose one of the tiers to apply the decision, adding or removing instances. A simple way to select a tier to be scaled is to regard the resource utilization of each tier. However, it is hard to choose which tier should be scaled in/out when the utilization of each tier is slightly different. Therefore, we define equation 4 selecting a tier to be scaled.

$$score = mask \times f(Tier_i) \quad (4)$$

According to the equation, an agent calculates a score per each tier and selects a tier with the highest score. This equation consists of a *mask* variable and a score function. The *mask* variable presents whether a scaling action can be applied to an environment. For example, the *mask* is 0 when there are no available resources for scale-out or not enough VNF instances running in the tier for scale-in, so it makes the score as 0. Otherwise, the value is 1 and enables the formula to calculate score per tier.

$$resUtil = \alpha CPU_{Util.} + \beta MEM_{Util.} \quad (5)$$

The score is calculated by each tier's status, which includes an average resource utilization ($resUtil$) and a distribution value of the tier. Equation 5 uses the utilization of both CPU and memory to calculate a score because they affect packet processing of VNF. In the formula, α and β are weights to adjust each resource utilization.

$$f(Tier_i) = \begin{cases} \frac{Node_{tier}}{Node_{used}} \times e^{-resUtil}, \text{ if scale-in} \\ \frac{Node_{used}}{Node_{tier}} \times e^{resUtil} \text{ , else if scale-out} \end{cases} \quad (6)$$

The distribution value of a tier is defined by how many nodes host the tier's VNF instances ($Node_{tier}$) among all nodes hosting the SFC's VNF instances ($Node_{used}$). A score function in equation 6 calculates each tier's score. For the calculation, this equation uses not only the distribution value of a tier but also an exponential function tuned by a resource utilization. In the case of scale-out, the score function produces a high score when a tier consumes the large amount of resources, or this tier's VNF instances are running on a few nodes, which are close to each other. On the other hand, in the case of scale-in, this function provides a high score if the resource utilization of a tier is small, or VNF instances of the tier are highly distributed.

IV. IMPLEMENTATION

In this paper, we implement an auto-scaling module to provide auto-scaling functions using DQN and threshold. The implemented module runs in an OpenStack environment and interacts with other components such as a monitoring module and NFV orchestrator (NFVO) to add or remove VNF instances as shown in Fig 4. Besides, we develop shell programs to test the proposed auto-scaling mechanism in the environment. We publish all of the implemented sources and detailed documents on a GitHub page [3].

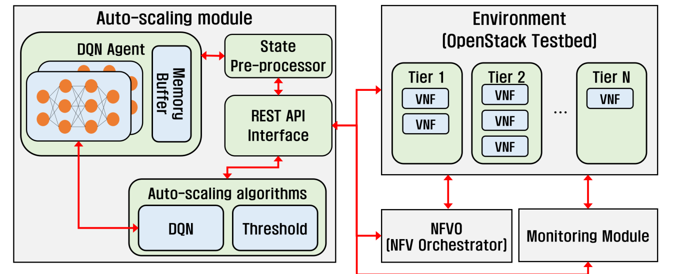

Fig. 4. Implementation of auto-scaling module

An auto-scaling module is responsible for adding or removing VNF instances in an SFC of the OpenStack environment. The proposed module needs to get information about resource utilization and VNF placement to define an auto-scaling problem. Thus, we implement a monitoring module that collects various data from the environment, physical nodes and VNF instances. In addition to the traditional purpose of monitoring systems, which is to monitor the status of the system and ensure that the system is working well, the purpose of a monitoring module is to provide state data which are used for running RL algorithms, DQN in this paper. A monitoring module continuously collects the data and stores it in a time-series database, InfluxDB.

To handle the VNF life cycle in an OpenStack environment, we implement an NFV Orchestrator (NFVO) module. An NFVO module provides REST APIs to create, update, and remove VNF instances. Besides, this module can set SFC. In this implementation, we use OpenStack Nova APIs[1] to deploy VNFs and OpenStack SFC APIs[2] to configure SFC paths. After an auto-scaling module has determined one of the SFCs created by NFVO to apply auto-scaling, the module continuously observes each tier's status in the SFC.

[1]OpenStack Nova, https://opendev.org/openstack/nova
[2]OpenStack SFC, https://opendev.org/openstack/networking-sfc

When an auto-scaling module runs, it interacts with a monitoring module to get information about nodes and VNF instances running in an OpenStack environment. An auto-scaling module applies a scaling policy to each SFC to be scaled, so the module gets available node information where VNF instances for the target SFC are running or can be deployed later. After obtaining the information, the module creates two Q-networks, one is for training, and the other is a target Q-network. Because a replay memory helps stable training of DQN, the auto-scaling module creates a memory buffer as a replay memory in this step. To implement the DQN algorithm for auto-scaling, we use PyTorch [27] and ADAM optimizer by default.

An auto-scaling module performs scaling actions as many times as episodes to find an optimal scaling policy. Although an optimal scaling action can improve performance in terms of load-balancing and SLO violation, it can also degrade performance if the scaling action is not optimal. Thus, a DQN agent should do many trial-and-errors for many episodes to find an optimal scaling policy. During those episodes, an auto-scaling module needs to consider a cooldown period to avoid oscillations regarding the number of VNF instances. The cooldown period is a time during which the module waits for scaling's effect. When an auto-scaling module performs a scaling action this module executes a new episode after a cooldown. A DQN agent in an auto-scaling module uses the current state as input data, so the module pre-processes information from a monitoring module to input the data to the agent. An auto-scaling module presents the input data as a tensor, and the size of a tensor depends on SFC length, i.e., the number of tiers in SFC. For example, the size of an input tensor is 10 if SFC length is 2 because each tier's status consists of 5 values such as CPU usage, memory usage, the number of disk operations, distribution, and size.

After creating and inputting a current state as a tensor data, an auto-scaling module chooses one of the actions such as *add*, *remove*, and *maintain* from the given state. The number of actions depends on the number of active nodes in an OpenStack environment because the proposed approach considers a node where scaling-in/out is carried out. If a selected action is *maintain*, the number of VNF instances does not change. DQN uses Q-value as a policy, so an agent chooses an action that has the maximum Q-value in the given state. However, the Q-values are not optimal in the initial stages, so the policy does not guarantee that a selected action is an optimal choice from the state. Therefore, it is necessary to consider the ratio at which to determine exploitation and exploration when choosing an action in each state.

Exploitation is a way of finding a policy with a preference in choosing an action by using a given value, and exploration chooses randomly one of the actions. Since the auto-scaling action is not optimal in the early episodes, the policy can converge into the wrong auto-scaling policy if an agent carries out using exploitation only. Therefore, an agent needs to consider carrying out random actions while DQN is running. Thus, an auto-scaling module uses an ϵ-greedy algorithm. The ϵ-greedy algorithm determines what to do between exploitation and exploration based on the ϵ value. An agent chooses an action more often randomly with a large ϵ value. Besides, an agent reduces an ϵ value by a certain percentage for each episode, gradually reducing the cases of choosing a random action.

Whenever an auto-scaling module chooses a scaling action, the module observes an OpenStack environment whether there are available resources to add or remove a VNF instance. If there are available resources for scale-out or enough VNF instances for scale-in, the module adds or removes a VNF instance in a target node and updates an SFC through an NFVO module. Otherwise, an auto-scaling module fails to perform a scaling action. After scaling, the module observes new states and calculates a reward. An auto-scaling module saves a transition into a memory buffer, which is an implementation of a replay memory. The mask included in a transition presents whether scaling action carries out or not. If transition data stored in a memory buffer is over a pre-determined number, which is 200 in this paper, an auto-scaling module uses the data for DQN learning. Besides, the module updates a target Q-network whenever an agent repeats a certain number of episodes.

V. Evaluation

To evaluate the proposed approach, we build an OpenStack-based testbed as shown in Fig 5. The testbed is built on Dell servers operating Ubuntu version 18.04. Those servers consist of compute nodes, a controller node, AI nodes, and a monitoring node.

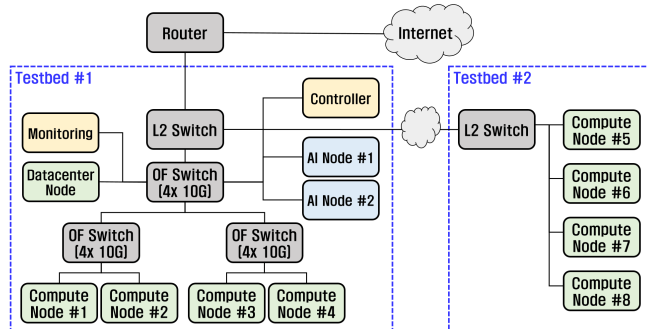

Fig. 5. OpenStack-based testbed

The controller node and compute nodes are components of OpenStack. When installing a new VNF, the controller node creates a VNF instance, i.e., VM in an OpenStack environment, on one of the compute nodes. The AI nodes run the DQN algorithm to apply an auto-scaling policy to running VNF instances. The monitoring node running a monitoring module collects the data used by DQN at every 1 second and stores it in a time-series database, InfluxDB[3].

[3]InfluxDB v1.7.9, https://github.com/influxdata/influxdb

Collectd[4] is used as the collector since it is fast and does not consume much resources of compute nodes in the OpenStack environment. In this evaluation, we assume that each VNF instance executes one VNF process. Those VNF instances run based on Ubuntu version 16.04, and specification is vCPU 1 core, RAM 1024MB, disk 10GB.

An auto-scaling module carries out scale-in/out for an SFC. Therefore, we create an SFC and have the module handle the SFC. A DQN agent in the module iteratively trains data and updates Q-networks to set an optimal scaling policy. We tune hyper-parameters for the iterations as shown in Table I. Besides, we limit the longest SFC length to 5, so SFC can have up to five VNF types. They are firewall, flow monitor, deep packet inspection (DPI), intrusion detection system (IDS), and proxy. To operate those VNFs, we use open-source software such as iptables [28], ntopng [29], nDPI [30], Suricata [31], and HAProxy [32] respectively.

TABLE I
DQN PARAMETERS

Parameter	Value(s)
η (learning rate)	0.01
γ (discount factor)	0.98
ϵ (probability of exploration)	0.08
Replay memory size	5000
Batch size	32

After creating an SFC, we need to generate traffic passing through the SFC's path. For traffic generation, we use RUBiS [33], which is a web application and an open-source benchmark tool widely used for experimental evaluation, as a destination of the SFC. RUBiS is an auction web application that users can access the web to search, bid, buy, and sell products. This tool provides an emulator to generate traffic which is client requests. We use this emulator to create dynamic traffic through an SFC and run a benchmark tool, stress-ng [34] optionally, to give VNF instances random stress. Moreover, an auto-scaling module uses a response time measured through an SFC to calculate rewards or check SLO violation. For response time measurement, this module leverages an Apache-Bench (AB) tool to generate probe packets, i.e., HTTP messages, and measure the mean response time of those messages. In our evaluation, an auto-scaling module measures the mean response time at every 10 seconds.

To evaluate the proposed approach, we create two SFCs, in which SFC lengths are 2 and 5 respectively. We apply three scaling approaches to each SFC; 1) **Threshold (Random)**: threshold-based scaling, randomly selecting a tier to be scaled, 2) **Threshold (Target)**: threshold-based scaling, selecting a tier to be scaled by proposed score function, 3) **Proposed DQN**: proposed DQN-based scaling. The evaluation environment is shown in Fig. 6.

In this evaluation, an auto-scaling module applies scaling-actions to two SFCs while each RUBiS client sends traffic

[4]Collectd v5.8.1, https://github.com/collectd/collectd

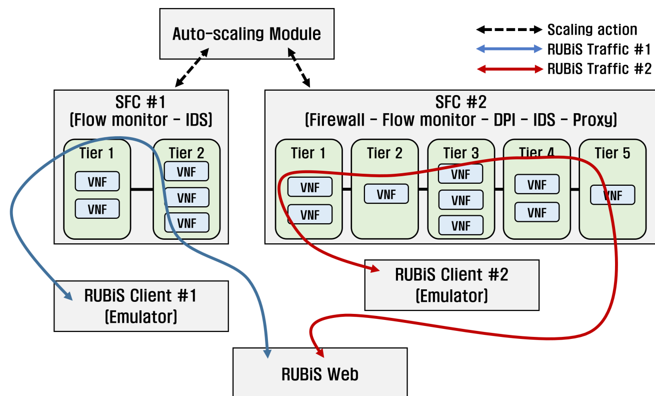

Fig. 6. Evaluation environment

through the SFCs. The Threshold (Random) and Threshold (Target) approaches carry out scale-in/out when measured response time is under or over the threshold. The threshold values triggering scale-in/out are 25ms, 35ms for 2-tier SFC, and 30ms, 40ms for 5-tier SFC. We determine those threshold values by referring to measured mean response time through each SFC while no client traffic.

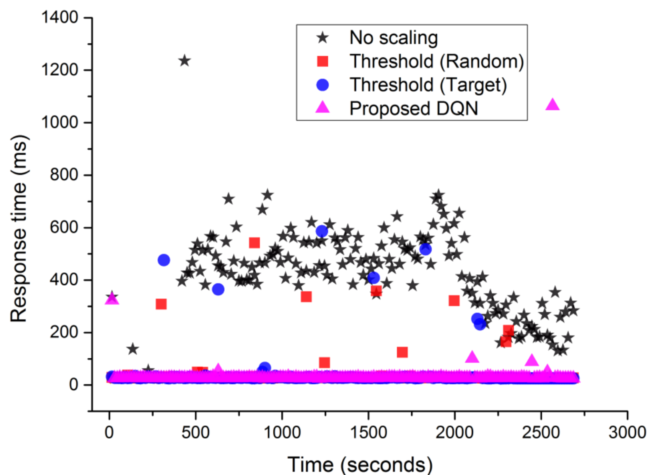

Fig. 7. Measured response time during auto-scaling (5-tier SFC)

RUBiS emulator generates traffic for about 45 minutes, i.e., 2700 seconds, so each auto-scaling approach also runs during that period. Fig. 7 presents response time measured by each approach. We also measure the response time while no auto-scaling to validate whether our scaling methods are effective. According to the result, Threshold (Random) and Threshold (Target) reduce response time compared to no auto-scaling case. Besides, the Threshold (Target) approach provides a smaller mean response time than Threshold (Random). This is because it chooses a tier to be scaled by using the proposed score function, which enables an auto-scaling module to select an adequate tier for scaling. Moreover, the result shows the DQN-based auto-scaling effectively adds or removes VNF in-

stances. The experiment presents that the method we proposed carries out appropriate scaling actions to handle dynamic traffic stably.

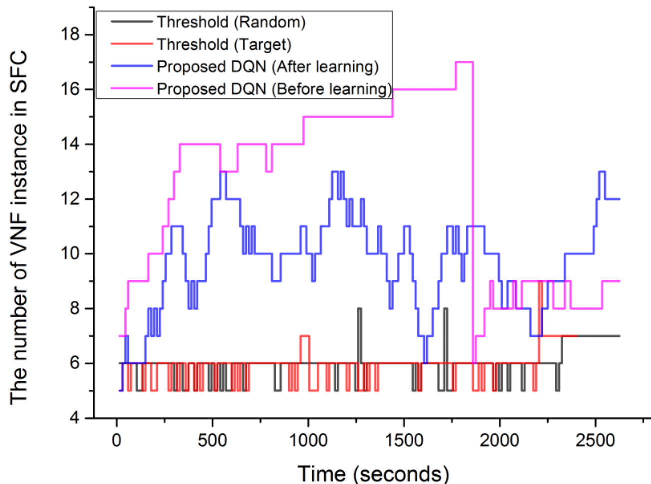

Fig. 8. The number of VNF instances during auto-scaling (5-tier SFC)

Our proposed auto-scaling method using DQN operates the appropriate number of VNF instances while reducing SLO violation. DQN takes states as input and determines a scaling action. Therefore, it can make a scaling decision before a mean response time exceeds a pre-defined threshold. Because the threshold-based method carries out scaling after the measured time exceeds the threshold value the SLO can be already violated at that time. However, our proposed method can make a scaling decision from a given state proactively before the SLO violation. Fig. 8 presents the number of VNF instances while auto-scaling is applied to an SFC, and traffic is generated through the SFC. Threshold-based approaches add or remove VNF instances when SLO is violated so it cannot effectively resize the number of instances. However, the proposed DQN method can effectively resize VNF instances. To show the effectiveness of the proposed method, we compare a trained DQN model using our reward model with an initial DQN model not trained yet in Fig. 8. According to the result, our proposed DQN model can operate the appropriate number of VNF instances while dynamic traffic passes an SFC. However, the initial DQN model before learning resizes the number of VNF instances inefficiently.

TABLE II
EVALUATION RESULTS

		Mean RT	SD	SLO violation
2-tier	Threshold (Random)	38.92ms	74.36	5.2%
	Threshold (Target)	35.26ms	55.52	4.2%
	Proposed DQN	26.38ms	10.71	5.3%
5-tier	Threshold (Random)	42.57ms	65.53	7.1%
	Threshold (Target)	45.36ms	82.04	5.7%
	Proposed DQN	35.91ms	81.38	4.6%

Table II summarizes the evaluation results in terms of mean RT (response time), SD (standard deviation), and SLO violation rate by each approach in two scenarios. An auto-scaling module implemented measures mean response time at every 10 seconds and if the mean response time exceeds a pre-defined SLO, it is an SLO violation.

According to the results, our DQN-based auto-scaling reduces mean RT. Therefore, the results present that the proposed method of updating an optimal policy using our reward model effectively makes scaling-actions. In this evaluation, we assume that the SLO is 45ms because it is enough time to process traffic through an SFC in our testbed. SLO violation rate of the proposed DQN is 5.3% in 2-tier SFC and 4.6% in 5-tier SFC respectively. The proposed auto-scaling approach uses DQN and provides scaling actions that minimize both mean RT and SLO violation effectively. Although a standard deviation of measured response time increases when SFC length is long, our approach ensures stable response time by allocating the appropriate number of VNF instances.

VI. Conclusion

In this paper, we propose a novel auto-scaling approach applied to NFV environments, which can deal with multi-tier VNF instances, i.e., SFC. The proposed method uses a deep Q-networks (DQN) algorithm to define an auto-scaling model and determines an optimal action for scaling of VNF instances. We define that an optimal action is adding or removing VNF instances to provide the optimal number of VNF instances while minimizing SLO violation. We use resource utilization of VNF instances, the number of disk operations, size, and a distribution value to define each state. The proposed auto-scaling method is validated in an OpenStack-based testbed. We implement an auto-scaling module running with the OpenStack components and evaluate the module by comparing it to a threshold-based auto-scaling method. According to the evaluation, our auto-scaling approach minimizes SLO violation while the module adds or removes VNF instances.

In the future, we plan to use a graph neural network (GNN) to apply network topology information to auto-scaling policy. Because the GNN can present network topology as a feature applied to machine learning algorithms, this feature can be used for RL-based auto-scaling in the NFV environments. Further, we plan to improve the proposed method and evaluate it in various scenarios that include showing the learning time required to find the optimal number of VNF instances and comparison with other methods of auto-scaling methods. Finally, we will study to speed up the DQN learning time to converge an optimal policy fast.

ACKNOWLEDGMENT

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (2018-0-00749, Development of Virtual Network Management Technology based on Artificial Intelligence).

REFERENCES

[1] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, 2018.

[2] M. Ersue, "Etsi nfv management and orchestration-an overview," in *Proc. of 88th IETF meeting*, 2013.

[3] DPNM, *Network Intelligent Project*. [Online]. Available: https://github.com/dpnm-ni/ni-auto-scaling-module-public

[4] S. Lange, H.-G. Kim, S.-Y. Jeong, H. Choi, J.-H. Yoo, and J. W.-K. Hong, "Machine learning-based prediction of vnf deployment decisions in dynamic networks," in *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2019, pp. 1–6.

[5] S. Lange, H.-G. Kim, S.-Y. Jeong, H.-Y. Choi, J.-H. Yoo, and J. W.-K. Hong, "Predicting vnf deployment decisions under dynamically changing network conditions," in *2019 15th International Conference on Network and Service Management (CNSM)*. IEEE, 2019, pp. 1–9.

[6] S. Park, H. Kim, J. Hong, S. Lange, J.-H. Yoo, and J. W.-K. Hong, "Machine learning-based optimal vnf deployment," in *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2020 (Accepted to appear).

[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[8] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*. IEEE, 2015, pp. 255–260.

[9] X. Wang, C. Wu, F. Le, A. Liu, Z. Li, and F. Lau, "Online vnf scaling in datacenters," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 140–147.

[10] A. Xie, H. Huang, X. Wang, Z. Qian, and S. Lu, "Online vnf chain deployment on resource-limited edges by exploiting peer edge devices," *Computer Networks*, vol. 170, p. 107069, 2020.

[11] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut, "Using machine learning for black-box autoscaling," in *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2016, pp. 1–8.

[12] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011, pp. 500–507.

[13] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a iaas cloud," in *Proceedings of the 9th international conference on Autonomic computing*, 2012, pp. 173–178.

[14] A. Evangelidis, D. Parker, and R. Bahsoon, "Performance modelling and verification of cloud-based auto-scaling policies," *Future Generation Computer Systems*, vol. 87, pp. 629–638, 2018.

[15] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow," in *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, 2011, pp. 67–74.

[16] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, "Predictive auto-scaling of multi-tier applications using performance varying cloud resources," *IEEE Transactions on Cloud Computing*, 2019.

[17] L. Yazdanov and C. Fetzer, "Lightweight automatic resource scaling for multi-tier web applications," in *2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 2014, pp. 466–473.

[18] Z. Wang, C. Gwon, T. Oates, and A. Iezzi, "Automated cloud provisioning on aws using deep reinforcement learning," *arXiv preprint arXiv:1709.04305*, 2017.

[19] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "Nfv-vital: A framework for characterizing the performance of virtual network functions," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 2015, pp. 93–99.

[20] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 71–84.

[21] G. Gardikis, I. Koutras, G. Mavroudis, S. Costicoglou, G. Xilouris, C. Sakkas, and A. Kourtis, "An integrating framework for efficient nfv monitoring," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE, 2016, pp. 1–5.

[22] J. Deng, H. Hu, H. Li, Z. Pan, K.-C. Wang, G.-J. Ahn, J. Bi, and Y. Park, "Vnguard: An nfv/sdn combination framework for provisioning and managing virtual firewalls," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 2015, pp. 107–114.

[23] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache, "Nfv orchestration framework addressing sfc challenges," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 16–23, 2017.

[24] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 121–136.

[25] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[28] D. Coulson, "Network security iptables," 2003.

[29] L. Deri, M. Martinelli, and A. Cardigliano, "Realtime high-speed network traffic monitoring using ntopng," in *28th Large Installation System Administration Conference (LISA14)*, 2014, pp. 78–88.

[30] L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano, "ndpi: Open-source high-speed deep packet inspection," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2014, pp. 617–622.

[31] *Suricata: Open Source IDS/IPS/NSM engine*. [Online]. Available: https://suricata-ids.org/

[32] W. Tarreau *et al.*, "Haproxy-the reliable, high-performance tcp/http load balancer," 2012.

[33] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," in *5th Workshop on Workload Characterization*, no. CONF, 2002.

[34] C. I. King, "Stress-ng," *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, 2017.