EdgeDQN: Multiple SFC Placement in Edge Computing Environment

Suman Pandey, Tu Van Nguyen, Jae-Hyoung Yoo, James Won-Ki Hong
Dept. of Computer Science and Engineering, POSTECH, Pohang, South Korea {suman, tunguyen ,jhyoo78, jwkhong}@postech.ac.kr

***Abstract*— Network Function Virtualization (NFV) and service orchestration has simplified the Service Function Chain management (SFC) tasks, while the edge cloud infrastructure has reduced the latency. Due to these existing technological advantages, there is an urgent need for a dynamic and flexible service chain placement model that performs resource allocation of substrate network in a delay-sensitive and resource-efficient manner. We propose an off-policy Deep Reinforcement Learning algorithm EdgeDQN for efficient SFC placement in the edge cloud environment. The problem of edge resource scarcity is handled by designing a network model that allows worst-case resource renting from neighbors and data centers. This network model is integrated with EdgeDQN using several constraints. This paper aims to find the optimal placement by minimizing the underlying resource utilization and SFC end-to-end delay for multiple SFCs at the same time. To achieve that, an intuitive reward model is proposed. We compare the proposed EdgeDQN algorithm with DQN, Q-learning, and EdgeQL algorithms in terms of performance parameters such as cumulative reward, cumulative standard deviation, latency, and learning convergence time for 420 different test cases. Extensive test results on a simulated and physical (OpenStack) testbed demonstrate the effectiveness of the proposed EdgeDQN algorithm.**

***Keywords*—AI-based Network Management, Deep Q-Network, Reinforcement Learning, OpenStack, Edge Computing**

I. INTRODUCTION

Managing and orchestrating VNFs has become easier than ever with the advent of advanced cloud management systems such as OpenStack [1]. It has also enabled operators to take advantage of Machine Learning (ML) algorithms to solve various VNF lifecycle management tasks [2]. SFC is one of the crucial VNF lifecycle management tasks. SFC provides a wide range of preprocessing for a request before it reaches the server through the chained VNFs, such as firewall, IDS, DPI, video optimizer, proxy, load balancer, etc. Service providers also use the multi-access edge computing (MEC) infrastructure [3] to support these SFC services with the desired short latency. Due to these technological advantages, network manager can create a dynamic and flexible service chaining deployment model that performs resource allocation of substrate network in a delay-sensitive and resource-efficient manner. In the edge cloud environment, the model should take care of the scenario when there are not enough resources available at the edge, in such a scenario it should be able to rent resources from neighbors or data centers. The model should also be designed to accommodate multiple SFC placement scenarios. This research focuses on efficient resource allocation for multiple SFCs on the edge cloud environment using Deep Reinforcement Learning (DRL).

The current OpenStack implementation provides a very basic approach to this problem [5], where incoming VNFs are placed on the least loaded server. The other option is to stack as many VNFs as possible on the same server. Also, the server resource constraints such as CPU, memory, storage and bandwidth etc. need to be configured separately. These approaches are very primitive, which gives us the scope and motivation to improve the SFC embedding. Embedding and placement is used interchangeably through this paper.

In academia, sufficient attention has been paid to the placement of a single VNF [6], but much less attention is paid to SFC placement. Moreover, there is no research that finds optimal placement for multiple SFCs using machine learning. The Most common approach to this problem is to first place the VNFs and then find the appropriate VNFs to create the chain [7, 8]. These approaches can be categorized as flow scheduling and routing problem rather than embedding problem, and they lead to unnecessary creation of VNFs and infrastructure overload without considering how they should be chained. Therefore, before embedding VNFs of SFCs, we need to consider the end-to-end delay of each SFC, the resource utilization (RU) of the edge infrastructure, and the overall requirements of the SFCs in advance.

The other common approach is to formulate the SFC embedding as a Virtual Network Embedding (VNE) problem [4]. The VNE problem is usually formulated as an optimization problem using graph and Integer Linear Programming (ILP) [9,10], where an SFC is represented as a set of VNFs and virtual links connecting these VNFs. VNFs and links require a certain amount of resources, such as processing power, memory, storage, and bandwidth, depending on the needs of clients and incoming requests. Based on these resource requirements of VNFs, they are mapped and embedded in the substrate network and link. These problems are combinatorial in nature, hence the ILP solution becomes NP-hard as the number of network parameters and SFC configurations increases. For the placement of multiple SFCs, the combinations are even higher, which makes ILP solutions impractical. Therefore, we considered ML-based approaches to solve this problem.

Among the ML approaches, RL (Reinforcement Learning) is a promising alternative to solve the combinatorial problems. In RL, the agent interacts with the environment by performing actions and receiving a reward. The agent can decide which action is best based on the long and short-term

978-3-903176-36-2 ©2021 IFIP

rewards after several iterations. We solved the SFC placement and server assignment problem for the edge cloud environment using an RL approach.

All RL problems are defined in terms of state, action, reward, and transition probability. It is also important to design the environment well, as the agent would interact with the environment and learn the rewards for its actions. A very basic RL approach to solve this problem is based on Q-learning [11, 12]. This related work is limited to a single SFC placement. They also chose to use a simplified state model. Their tests were limited to only 8 VNFs. We model the state with CPU, memory, storage and bandwidth. Such a complex state model leads to a high state-action combination, which makes the Q-learning model very impractical. In such a situation, we need a proper function approximation technique which is provided by DRL. Therefore, we have chosen the Deep Q-Network (DQN) algorithm introduced by DeepMind [13] to solve the problem of placing multiple SFCs. Learning from experience through neural networks and function approximation is integrated in this selection process. The DQN approach can efficiently determine the near optimal solution. We call this algorithm EdgeDQN.

The two main challenges in implementing the EdgeDQN algorithm were the size of the action space and modeling a reward function that supports multiple SFC placements. It is not useful to consider only the edge resource, as the edge infrastructure can quickly reach its limits. Therefore, a mechanism is needed that can rent the resources of the neighbors or the data center in the worst case. However, considering the whole network could increase the action space tremendously. The action space in RL grows as the target topology grows. To address these issues, we opted for a hierarchical model of the network consisting of local, edge, and data center servers. Constraints are introduced, so that the EdgeDQN will look for neighbor and DC resource only if the edge is not able to fulfill the requests due to lack of resources. The introduction of multiple edge-specific action selection constraints helps the algorithm to converge faster with less exploration.

The second major challenge was to design a reward model suitable for multiple SFC placement scenarios. Mostly, researchers have designed a reward function based on placement success and failure [14] or throughput [15]. These reward models have two main drawbacks. First, these reward models completely ignore the resource utilization of the underlying infrastructure. Resource utilization is an important parameter to reduce energy consumption by keeping servers and switches in energy saving mode. Second, they cannot be used for the multiple SFC placement scenario. Therefore, we have developed a reward model that considers the underlying resource utilization and optimizes the placement for multiple SFC placement. Our reward model awards higher rewards for more complex SFCs, e.g., SFCs that require more resources and are longer in length. Successful placement of such SFCs should be rewarded with higher rewards. Moreover, our reward model maximizes the resource utilization and minimizes the total delay of each SFC.

Another distinguish contribution of this work is the evaluation methodology. Most related works evaluate their algorithm based on one test case over multiple episodes [13, 14]. This approach cannot guarantee the legitimacy of the DRL algorithm, since a large part of the DRL algorithm depends on the randomness posed by the *e-greedy*[16]. It is possible that algorithm behaves well in one test run and doesn't behave well in other. Therefore, adopting a commutative approach for evaluating DRL algorithm is essential. We evaluated our model by comparing the cumulative reward of 420 different test cases and the cumulative standard deviation by running each test case 10 times. Our model performs significantly better compared to our preliminary studies based on Q-learning and DQN. We also evaluated this algorithm on the OpenStack physical testbed with 42 different test cases. We found that the cumulative latency of test cases is much lower for the proposed EdgeDQN algorithm.

Our main contribution can be summarized as follows:

1) First attempt to use DRL for multiple SFC placement at the same time. This is achieved by designing a reward model based on SFC complexity, SFC end-to-end latency and underlying network resource utilization.
2) A hierarchical network constraint model for reducing the action space for DRL.
3) Cumulative evaluation methodology instead of evaluating individual test cases.
4) Evaluation of the proposed model on the OpenStack physical testbed.

The rest of the paper is organized as follows. Section II elaborates some of the related work. Section III defines the service topology and environment. Section IV explains the proposed algorithm with discussion on state, action, reward and constraints model. Section V explains the evaluation methodology, results and discussion. Finally, section VI concludes our work with possible future work directions.

II. RELATED WORK

Our related work is focused on elaborating the use of DRL in the field of VNF and SFC embedding. In the past DRL has been used to solve this problem in two steps, where VNFs are placed first and then linked [6, 7]. Recently DRL was also used for flow scheduling [17]. Such approaches can be categorized as routing and scheduling problem rather than a placement problem.

DRL has been also used extensively for scaling and load balancing of VNFs [18]. Some researchers used scaling and embedding as a combined approach as well [19]. However these research are limited to one SFC. The DRL module would scale in and out VNFs for a single SFC. Our research focused on embedding multiple SFC at the same time finding the optimal placement reducing the resource utilization and end-to-end delay. Given the scale out information, our module can do the VNF embedding for existing SFC as well utilizing same reward model.

Recently a few researchers have opted DDPG and A3C algorithm to solve VNF placement [14, 15]. Two main problems of their approach in solving an SFC placement are

1) high dimensionality of the state [14], as they considered entire network in their state model. 2) large size of the discrete action space. The high dimensionality of the state would require a large amount of training and the large action space would require a high exploration in the DRL. Reference [15] proposes a Heuristic Fitting Algorithm (HFA) to reduce the action space based on the capacity of the substrate network to serve the query. However, this approach reduces the action space only when the network is heavily loaded.

The reward model of related work is also limited to throughput [14] and successful placement of VNFs [15] only, which completely ignored the energy consumption of the placement and the complexity of the placement. Hence, we designed a reward model that considers the complexity of the SFC and the resource utilization (RU) of the underlying network in addition to the end-to-end delay.

The basis of a DRL algorithm is the design of its environment, state, action, and reward. Currently, we have focused on these aspects and tested the DQN and Q-learning algorithm on our model. In the future, we could easily incorporate the DDPG and A3C algorithm into the current design of environment, state, action and reward function. A realistic network scenario representing an edge cloud with an intuitive reward function and the evaluation methodology are the main contribution of our work. The basic DQN algorithm is also significantly modified to incorporate these aspects of network design. To the best of our knowledge, none of the related work has focused on the placement of multiple SFCs with DRL.

III. Service Topology and Environmnt Variables

In this research, we represent an edge cloud environment as a hierarchical network model. The hierarchy of nodes from top to bottom consists of center, core, edge, spine, leaf, top of the rack (TOR) switch, and servers as shown in Fig.1. In the hierarchical network model, an edge router represents a Central Office (CO). Edge routers connected to the same core router are neighbors to each other. In SFC placement in the edge environment, we only care about the local, neighbor, and DC network.

The information about the edge router R_e and the core C_{dc} is presumed to be given in advance. Given the edge router R_e, we extract the substrate topology attached to it. That gives us the set of local servers S_l. The substrate topology of C_e gives us a set of servers attached to it represented as S_e where $S_l \subset S_e$. Similarly the set of servers connected to C_{dc} is represented as S_{dc}. This way S_e, S_l, and S_{dc} represents the set of edge, local and DC servers. It is important to note that the network topology is converted to the liner sets of S_e, S_l, and S_{dc}. For obtaining this set from Openstack test bed, we just need edge router R_e and the core C_{dc} information as inputs.

Each server in the set S_e, S_l, and S_{dc} is represented in terms of resource availability. Resources can be CPU, memory, storage, network bandwidth, etc. H denotes the set of servers, server IDs from 1 to h. H' denotes the set of servers in a power-saving mode, where $H' \subset H$. These are the servers without VNF placement. K denotes the types of resources, resource type ID from 1 to k. CPU, memory, storage and bandwidth are the main resources, and so k is 4. a_h denotes a resource vector for each server $[a_{h1}, a_{h2}.. a_{hk}]$. a^a_{hk} denotes the available resource, and a^o_{hk} represents the occupied resource.

An SFC consists of multiple VNFs connected by virtual links (VLs). Each VNF requires a certain amount of resources according to the number of supported users and type of services. N is the number of VNFs in SFC with sequence ID from 0 to n. v_n denotes a VNF flavor, and resource vector of a particular v_n is denoted as $[v_{n1}, v_{n2} \ldots v_{nk}]$. SFC is denoted as a list of v_n $[v_1, v_2 \ldots v_n]$.

VL is defined as the number of hops in the substrate link multiplied by the link delay. The number of hops between two VNFs ID i and $i+1$ is denoted as vl_i^{i+1}, and the link delay is denoted as vl_d. The value of vl_d is assumed to be the same unit value (e.g., 1 ms).To simplify the calculation of the delay, we keep an array that defines the number of hops from server to server. The number of hops for the same server is 0, same TOR is 1, same edge is 3, neighbor is 7, and DC is 9 for this work. We simply check which particular sets the two nodes i and $i+1$ belong to in order to assign the number of hops. The total delay of SFC can be expressed as the sum of the delays of these virtual links between VNFs. Our delay model can be improved in the future.

All the environment variables including S_e, S_l, and S_{dc} , vl_i^{i+1}, vl_d , a^a_{hk} and a^o_{hk} are used in imposing constraints and calculating rewards. Our objective is to minimize the overall delay of SFCs while embedding the VNFs. The other goal is to maximizing resource utilization (RU). RU is required to reduce the energy consumption by keeping the servers and switches in energy-saving mode. These two objectives are taken care while designing the reward functions of the DRL model.

In the best case, all VNFs in an SFC should be placed to the local CO (central Office) assuming there are sufficient resources on the local CO. One of the best edge placements of a first few requests of the SFC with given constraints on edge resources with minimum delay and maximum RU is shown in Fig. 1.

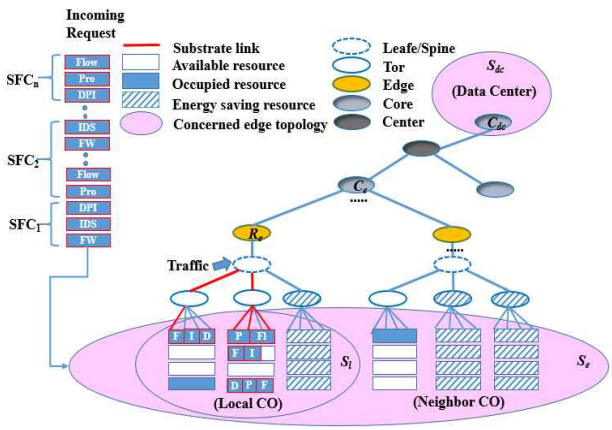

Fig 1: Possible placement operation of SFCs with minimum substrate link delay and maximum resource utilization on an Edge cloud environment.

IV. Proposed Approach

We designed an EdgeDQN algorithm by adding edge-specific constraints to the DQN algorithm. The number of ways SFCs can be placed in a physical network is combinatorial. The DRL algorithm would have to try multiple combinations to obtain the best policy. Therefore, a simple DRL approach potentially requires a large number of explorations to find an optimal solution. We address this challenge by building a hierarchical model of the network based on local, neighboring and DC. This will give the first priority to the local servers, however in the case of resource scarcity only the algorithm would reach to the neighbor and DC servers. Such constraints reduces the action space while also taking care of edge resource scarcity issues. The reward model is designed to incorporate the energy consumption and complexity of SFC with end-to-end latency.

We first explain the state of the art of RL algorithms, including Q-learning and DQN, followed by a detailed discussion and benefits of our state, action, and reward model. We then explain the edge-specific environment variables and their role in improving the action selection policy, and finally we explain our overall EdgeDQN agent. We compared the EdgeDQN algorithm with QL, DQN and EdgeQL algorithms with common modeling of state, action, and reward function. Our first version of QL and EdgeQL [11] algorithm is extended to support the new reward model, states and multiple SFC placement scenarios, same as explained in this paper.

A. DRL Approach

Usually, an RL problem is modeled by 4-tuple states (s), actions (a), state transition probabilities (pi), and rewards (r). In RL there is a learner (also called an agent), that interacts with the environment to select an action for moving to the next state based on the rewards. The RL algorithm is based on reward and value function. Reward indicates what is good and bad in an immediate sense, and value function indicates what is good and bad in the long term. Learning happens in several *episodes*. In each *episode i*, the agent in state s_i performs an action a_i, receives a reward r_i, and moves to the next state s_{i+1}. In Q-learning the action value is updated in the Q matrix of size [s × a] with the formula given in (1). α represents learning rate (0.05). γ represents discount rate (0.5). We use argmax to get the action that returns the highest reward. In Q-learning the agent's brain is Q matrix, on the other hand in DQN the agent's brain is a deep neural network (DNN).

$$Q(s_i, a_i) = (1-\alpha)Q(s_i, a_i) + \alpha(r_i + \gamma \max Q(s_{i+1}, a_i)) \quad (1)$$

In DQN we create DNN and train it to resemble $Q(s, a)$. The input in the neural network is a tuple $E_{(s_i, a_i, r_i, s_{i+1})}$ for the current environment. This information is stored in replay memory. During the interactions between the agent and environment, it periodically samples data from the replay memory for training the DNN, and updating the weight parameters ρ in the DNN to minimize the loss function, as in equation (2).

$$L_i(\rho) = E_{(s_i, a_i, r_i, s_{i+1})}(r_i + \gamma max Q(s_{i+1}, a_i) - Q(s_i, a_i)^2) \quad (2)$$

B. EdgeDQN States

The success of the RL algorithm depends on modeling state, actions and rewards. In our implementation states represent the set of VNFs. Several researchers [17] used the entire network and VNF-FG (Forwarding Graph) request as a state. However such an input for learning model will require huge number of samples eventually increasing the learning time. To avoid that we modeled our state with VNF requests *[v_1, v_2, v_3, ø, v_4, v_5, v_6, ø.... , v_{n-2} , v_{n-1} ,v_n]*. In this VNF requests ø denotes the end of one SFC request. Each VNF request is mapped to a resource vector of that VNF, we also call it VNF flavor. The resource are CPU, memory, storage and bandwidth requirement of the VNF denoted as *[v_{n1}, v_{n2} ... v_{nk}]*. Hence each state in DRL is basically a set of 4 variables *[v_{n1}, v_{n2}, v_{n3} v_{n4}]*.

The substrate network state is modeled separately as environment variables. The environment variables includes the set of edge *(S_e)*, local *(S_l)*, DC servers *(S_{dc})*, each server's available (a^a_{hk}) and occupied (a^o_{hk}) resources. These environment variables are used for imposing several constraints during action selection process. These also helps in calculating rewards.

C. EdgeDQN Action

Action represents the selection of the appropriate server for deployment. The set of actions are represented as *[a_1, a_2 .. a_h]*. The action space increases as the size of the network grow. Hence it is essential to devise a mechanism to reduce the action space. The action space is minimized by using a carefully crafted EdgeDQN action selection policy constraints.

D. Additional Constraints for Action Selection

Reducing the action space to suit the edge SFC placement is one of the important aspects of this research. To reduce the action space, we presented a hierarchical stochastic model for our environment. Moreover, we used a hierarchy of constraints to filter the best action that favors edge SFC placement, followed by neighboring Edge and DC placements.

The stochastic model of the environment is represented by variable θ and P. θ_{hn} stores 1 if the server has enough capacity to deploy the VNF, otherwise, it stores 0. h denotes the server id, k denotes the resource type and n denotes the VNF. a^a_{hk} represents the available resource k on the server h, and v_{nk} represents the resource demand of resource k for VNF n. Refer to (3) and (4). Later, this array is converted into the state transition probability and stored in P, see (5). P_{hn} gives us the probability of server h to place VNF n. Value of P_{hn} will be 0 if server h lacks to serve any of the resource demand of VNF n.

$$a_{\mathrm{hk}}^{a} = \begin{cases} 1, & a_{\mathrm{hk}}^{a} - v_{nk} > 0 \\ 0, & a_{\mathrm{hk}}^{a} - v_{nk} \leq 0 \end{cases} \quad (3)$$

$$\theta_{hn} = \begin{cases} 1, & \prod_{k=0}^{k} a_{\mathrm{hk}}^{a} > 0 \\ 0, & \prod_{k=0}^{k} a_{\mathrm{hk}}^{a} = 0 \end{cases} \quad (4)$$

$$P_{hn} = \frac{\theta_{hn}}{\sum_{h=0}^{h} \theta_{hn}} \quad (5)$$

The hierarchical edge network model is represented by the sets S_e, S_l, S_n, and S_{dc}. Starting from an edge router R_e and DC core router C_{dc}, we can traverse the topology to extract S_e, S_l, and S_{dc}. By traversing the tree associated with R_e, we obtain the set S_l. If any of the resource in the server h is not enough to serve VNF n then the value of P_{hn} will be 0, and such servers should be removed from the action space as in equations (6~8). S_d represents the servers that should be removed from the server at each hierarchy. We derive a set of local servers S_l containing only the servers with enough resources as described in equation (6). S_n represents the set of neighboring Edge servers. These are the servers that are connected to core edge router C_e but are not part of S_l and have enough resources, as shown in (7). S_{dc} represents the set of DC servers that are extracted given core router C_{dc} in the network topology. Then, a subset of S_{dc} with sufficient resources is derived by (8).

$$S_l = S_l - S_d \quad \text{where } S_d \in P_{hn} = 0 \quad (6)$$
$$S_n = S_e - S_d \quad \text{where } S_d = (S_l \cup S_o),\ S_o \in P_{hn} = 0 \quad (7)$$
$$S_{dc} = S_{dc} - S_d \quad \text{where } S_d \in P_{hn} = 0 \quad (8)$$

In standard DQN algorithm we predict the *q_values* for a particular states and then the *q_values* that gives the highest reward is selected as the action. In our approach while passing the *q_values* for the highest reward selection we consider the actions in the hierarchy we defined using S_l , S_n and S_{dc}. When selecting the action in the EdgeDQN algorithm, the conditions in (9) are checked in a hierarchical way. If set S_l is not null action is selected from S_l. If set S_l is null then action is selected from S_n. If S_l and S_n are all null then action is selected from S_{dc}.

$$a = \begin{cases} argmax\,(S_l), & S_l \neq \emptyset, \text{deploy at local} \\ argmax\,(S_n), & S_l = \emptyset \wedge S_n \neq \emptyset, \text{deploy at neighboring} \\ argmax\,(S_{dc}), & S_l = \emptyset \wedge S_n = \emptyset, \text{deploy at DC} \end{cases} \quad (9)$$

Even though we have incorporated the graph embedding approach by computing the shortest path in our reward function, this is not sufficient to guarantee edge placement in the DQN unless we train the algorithm with a significantly large number of episodes. The action selection policy proposed in EdgeDQL not only reduces the number of episodes and the amount of exploration but also ensures that local servers are given the highest priority, followed by the neighbor and DC servers.

E. Reward

Rewards are feedback to the agent about how good its actions are. This reward function represents the throughput and cost of the placement combined. The reward is calculated cumulatively after all VNF placements in a chain are completed. In addition to successful SFC placement, we also aim to minimize the overall latency and maximize the utilization of physical resources. Therefore, we modeled our reward function based on the complexity of an SFC, the E2E delay of the SFC, and the energy consumption of the placement.

It is important to consider SFC complexity in case of multiple SFC placement. A successful placement of an SFC with higher complexity should receive a higher reward. An SFC which requires higher amount of resources such as CPU, memory, disk or bandwidth is of higher complexity. It is harder to place such SFCs. As the length of the SFC increases the number of VNFs increases and it can also increase the resource demand. Hence we defined the complexity of the SFC in terms of its resource requirements as in (10). *k* here is the type of resource CPU, mem, disk and bandwidth and *n* is length of SFC. v_{nk} represents each resource requirement of a VNF in the chain of SFC. Higher value of ϖm should receive higher reward.

$$\varpi m = \sum_{n=0}^{n} \sum_{k=0}^{k} \frac{v_{nk}}{k} \quad (10)$$

The latency of an SFC S_{vl} is modeled by (11). Therefore, the number of hops for each VNF placement in a sequence is stored in a virtual link array vl_i^{i+1}, and at the end of the chain, the total delay is calculated cumulatively. We took a ratio of S_{vl} and n to normalize the latency parameter, where n is the length of SFC. The reward should be low for high value of S_{vl}/n. Therefore, we chose an exponentially decaying profile to include it in the reward function. It is possible to improve this delay equation bye considering several other factors such as propagation delay, processing delay of VNF etc. in future.

$$S_{vl} = \sum_{i=0}^{n} vl_i^{i+1}\, vl_d \quad (11)$$

Additionally maximizing resource utilization is required to reduce the energy consumption by keeping the servers and switches in energy-saving mode. RU is modeled as (12). Here h is number of servers, that is length of set H, and h' represents number of servers in saving mode, that is length of H'. k represents type of resource. a_{hk}^{a} represents the available resource and a_{hk}^{o} represents the occupied resource on a server h. At the end of the sequence, the resource utilization RU of the placement is calculated using (12). A higher value of RU represents better resource utilization and hence lower energy consumption.

$$RU = \left(\sum_{k=0}^{k} \sum_{h=0}^{h} \frac{a_{hk}^{o}}{a_{hk}^{a}}\right) / (h - h') \quad (12)$$

Equation (13) represents our overall reward function calculated after each SFC. This reward function gives high rewards for lower E2E latency. The exponential delay equation $e^{-\frac{\beta S_{vl}}{n}}$ gives higher reward for lower value of $\frac{\beta S_{vl}}{n}$ and $e^{\lambda RU}$ will give higher value for higher resource utilization. Finally multiplying it with SFC complexity will give higher rewards for complex SFCs.

$$r = \xi \times \omega m \times e^{-\frac{\beta S_{vl}}{n} + \lambda RU} \quad (13)$$

ξ and β and λ are the constants that can be adjusted to give higher importance to one or the other parameter. In this simulation, ξ represents a higher value, e.g., 50, and β and λ represent a lower value, e.g., 2.

F. Agent

The agent takes the set of the SFC requests and substrate topology as input. The substrate topology is converted in the environment variables explained in Section II. The VNF in the SFC requests is modeled as states of RL algorithm and represented as 4 tuple tensor $[v_{n1}, v_{n2}, v_{n2}\ v_{n4}]$ indicating CPU, memory, storage and bandwidth requirement of each VNF. The output of the algorithm is a list of actions. Fig 2. shows the overall functioning of Agent.

The action represents the servers to embed a particular VNF. The important point to note here is that, reward is calculated for each SFC in a commutative manner, not just for one VNF. Also the reward for each state is updated only after the sequence is complete, as we can see in line 18, 19 in EdgeDQN agent algorithm.

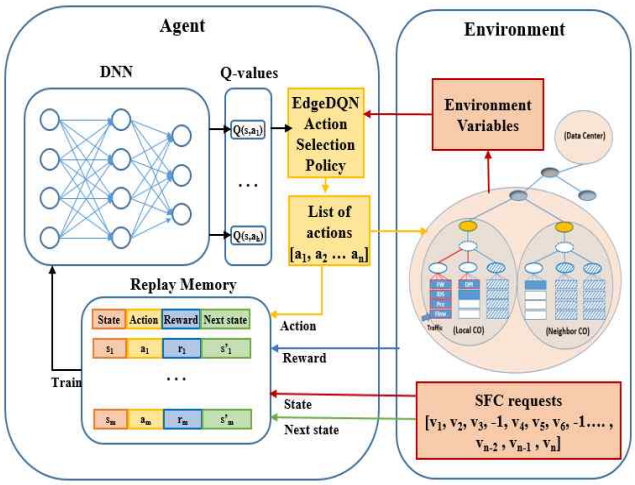

Figure 2: EdgeDQN Model

In DQN agent improves action selection strategy by switching between exploration and exploitation of the solution space for each hierarchy as shown in line 9, 11 and 13 . *ε-greedy* selection helps the learner uses a small amount of randomness to explore new solutions. *ε-greedy* method indicates that the agent has a probability of $1-\varepsilon$ to choose the action a_n that maximizes $Q(s,a)$, and has the probability of ε to randomly choose the action a_n. Periodic decay of Q-values is taken care of using gamma (discount factor *epsilon* = *epsilon*/1.5).

The agent predict the *q_values* for a particular states and then the *q_values* that gives the highest reward is selected as the action. However, while passing the *q_values* for the highest reward selection we consider the actions in the hierarchy S_l , S_n and S_{dc}. This way if set S_l is not null action is selected from S_l. If set S_l is null then action is selected from S_n. If S_l and S_n are all null then action is selected from S_{dc}.

After each action selection, we also update the environment variables again (line 15). The selected action is then appended in the list of action. This action list is passed to the reward function after each SFC. The action list helps in calculating the overall latency. Based on the complexity of the SFC, overall latency and resource utilization of the SFC the reward is calculated and added to the commutative reward. All the past experience is then stored in the memory (line 23). The neural network used by the DQN agent updates its gradient using backpropagation to finally converge (line 25). DQN was modeled with two hidden ReLU layers. Mean square error (MSE) based loss function and ADAM optimizer are used.

Algorithm : EdgeDQN Agent

1. Input : Network topology, SFC requests $[v_1, v_2, v_3, \emptyset, v_4, v_5, v_6, \emptyset\ldots., v_{n-2}, v_{n-1}, v_n]$.
2.Output: list of actions $[a_1, a_2 .. a_n]$
3.env.**initialize**()
4.For all episodes
5. Initialize reward=0, action = []
6. **For** all states v_n
7. **If** v_n != $\emptyset$:
8. **If** env.$S_l \neq \emptyset$
9. $a_n = \begin{cases} \text{random.choice(env.}S_l\text{, env.P[}S_l\text{])} \\ q_{values} = \text{model.predict(}v_n\text{), argmax(}q_{values}\ [S_l]\text{)} \end{cases}$
10. **Else If** $S_l = \emptyset \wedge S_n \neq \emptyset$
11. $a_n = \begin{cases} \text{random.choice(env.}S_n\text{, env.P[}S_n\text{])} \\ q_{values} = \text{model.predict(}v_n\text{), argmax(}q_{values}\ [S_n]\text{)} \end{cases}$
12. **Else If** $S_l = \emptyset \wedge S_n = \emptyset$
13. $a_n = \begin{cases} \text{random.choice(env.}S_{dc}\text{, env.P[}S_{dc}\text{])} \\ q_{values} = \text{model.predict(}v_n\text{), argmax(}q_{values}\ [S_{dc}]\text{)} \end{cases}$
14. **End If**
15. env.**update**(v_n, a_n) based on (3) ~ (8)
16. action.**append** (a_n)
17. **Else**
18. Reward is calculated after the end of each SFC
19. $r = r + \xi \times \omega m \times e^{-\frac{\beta S_{vl}}{n} + \lambda RU}$ based on (13)
20. **End If**
21. **End For**
22. **For** all states v_n
23. **remember**(**Transition** (v_n, a_n, r, v_{n+1}))
24. **End for**
25. **replay**(batch_size) based on (2)
26.End For

Our EdgeDQN implementation is not a simple DQN implementation. We introduced several environment variables and constraints, divided the network in a hierarchy, after each action selection we also updated the environment to reflect the action. Apart from that we introduced a communicative reward function which gives higher rewards to complex SFC with lower delay and higher resource utilization. All these properties make this algorithm superior to the current state of art DQN algorithms for SFC embedding.

V. Results and Evalutation

A. Simulation Results

Our simulation and physical testbed topology are similar to MEC scenario, as shown in Fig. 1. Our simulation network consists of 6 edge routers, 3 Tor switches attached to each router and 9 servers attached to each Tor switch. Which makes it to be 162 servers all together. The Leaf and Spine switches are merged with the edge routers for simplicity. Each edge server has compute, memory, storage and bandwidth capacity of 6 CPU core, 8 Gb, 90 Gb and 50Gbps units respectively. DC servers have a higher amount of compute, memory, and storage resources of 32 CPU core, 48 Gb, 120 Gb and 100Gbps units respectively. VNFs are taken with varying capacity of 1 to 8 CPU core and 1 to 8 Gb memory and 1 to 20 Gb storage. SFC bandwidth demand varies from 1Gbps to 5Gbps. Delay between two hops is considered as 1 ms. We tested our algorithm up to 64 VNFs divided into 10 SFCs of various resource demands.

We designed 420 different test cases with different SFC lengths, VNF flavors, and random server loads. We used tensor-flow and Keras to implement Q-learning and DQN algorithms. We ran our AI module on a Linux machine with 2GB ram and 4 VCPUs.

We compared our EdgeDQN algorithm with three base case scenarios QL, DQN, and EdgeQL. QL and DQN are basic Q-learning and DQN algorithms, respectively. All of our algorithms have the same state, action, and reward models, but EdgeQL and EdgeDQN include edge-specific action selection constraints and hierarchical network modeling. The QL and DQN algorithms, despite having the shortest path reward function, require more training to converge to the best results. We compared these algorithms with four different evaluation matrics: 1) cumulative reward 2) cumulative SD 3) cumulative latency 4) learning time.

A legitimate comparison of the algorithms requires that we compare multiple test cases, and thus it is best to use a cumulative value of the finally learned reward of all 420 test cases, as shown in Fig. 3. It can be seen that EdgeDQN learns the best reward compared to all other algorithms.

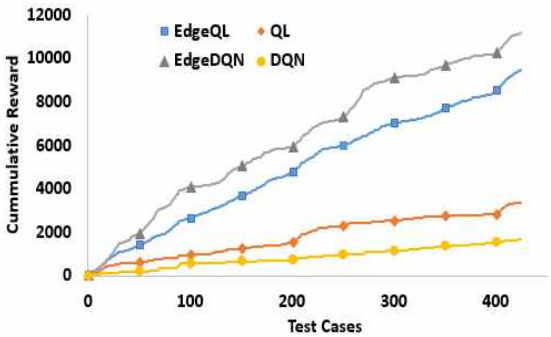

Figure 3: Cumulative reward of 420 different test cases

The standard deviation defines the difference in learning between multiple runs of the same test case. We ran the same algorithm T=10 times with the same hyperparameters and averaged the final learned reward. Lower SD represents the stability of the algorithm, and EdgeDQN outperforms all other algorithms (Fig. 4).

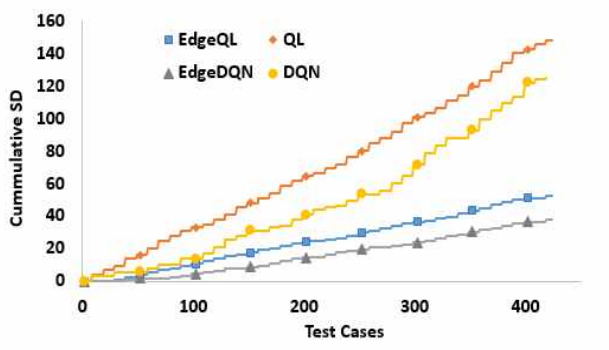

Figure 4: Cumulative SD of the finally learn rewards, N = 10

Fig. 5. Shows the cumulative latency of multiple SFCs. 64(10) in x axis of Fig 5, represents 64 VNFs divided into 10 different SFCs. These 10 SFCs are of size 3, 5 and 8 length. As we increase the size of the VNFs for placement, we observe that performance of DQN is much better. DQN performs better even for small number of states. Even for the SFC of length 5 the performance of EdgeDQN is better as compare to EdgeQL.

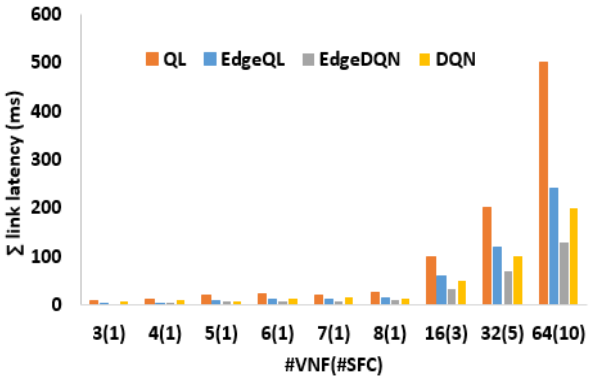

Figure 5: Cumulative link delay

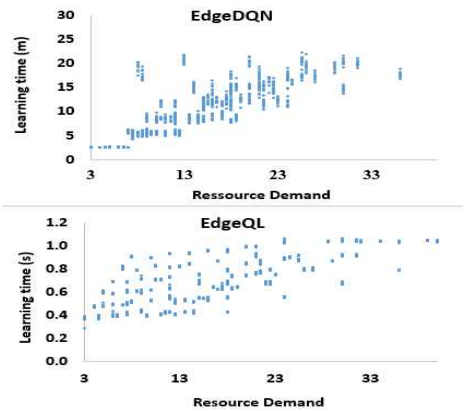

Figure 6: Learning time of Algorithm

We also observed that the learning time for EdgeDQN is higher as compare to EdgeQL. Neural networks and backpropagation mechanism used by EdgeDQN consumes time. The learning time is linearly proportional to the resource demand of the SFCs as shown in Fig 6. Higher resource demand such as CPU, memory etc. required higher amount of learning time in EdgeDQN algorithm. Here the resource demand in plotted only based on CPU demand. The

profile is similar for all other resource. One way to improve this learning time is by using Actor Critic method such as A3C. A3C uses multiprocessing and it can help reduce this learning time significantly. We will consider improving learning time in our future work.

B. *OpenStack Testbed Results*

We also verify our algorithm on the physical testbed with OpenStack environment. Fig.7 shows the overall integration of and testing approach of EdgeDQN with OpenStack. Our EdgeDQN modules interact with NFV monitoring modules to collect data, and NVF orchestrator[1] module to deploy SFC. The time-series database (InfluxDB[2]) is used to store the network state information. Collectd[3] is used as a collector of data from the testbed. This infrastructure consists of 8 edge servers and 1 DC server. Delay is emulated on this testbed using DEMU[4]. The delay between edge and core switch is 5ms and DC and core switch is 12 ms. 5 different types of VNFs were used which include iptables, ntopng, nDPI, Suricata, and HAProxy.

Our AI node runs the EdgeDQN algorithm and interacts with the monitoring and orchestrator module. It takes the list of SFC requests as well as edge_switch_id and DC_core_id as inputs from the json file. It extracts the topology information using the monitoring module and creates the EdgeDQN environment. It outputs the list of servers for provisioning each VNF. With the help of orchestrator VNFs are deployed and chained. Stress-ng is used to generate random loads on the servers. Traceroute is used to check SFC placement and chaining, and wrk is used to measure overall SFC latency from the client.

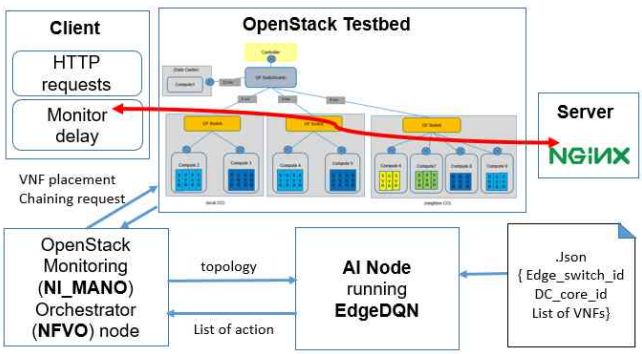

Figure 7: Integrating EdgeDQN with OpenStack

In Fig. 8, we compared our EdgeDQN algorithm with the OpenStack standard approach to VNF embedding. We placed several SFCs using the EdgeDQN and OpenStack standard approaches, and then tested two SFCs of length 5 and 3 by generating traffic for 3 hours. We measured the response time for simple web traffic going through these SFCs to the target Nginx server. In Fig. 8, we can see a significantly better performance of the EdgeDQN embedding compared to the standard OpenStack embedding.

OpenStack default VNF placement distributes instances evenly across all hosts (nova-scheduler)[5]. It assigns the weight for placing VNF on all available hosts. This weighting mainly depends on the available resources on the servers. The server with higher available resource is selected for placement. In this way, Openstack tries to maintain equal load on each server. We can manually configure the distribution option based on all types of resources, including CPU, ram and disk, using the *cpu_weight_multiplier*, *ram_weight_multiplier* and *disk_weight_multiplier* flags respectively. This approach has several drawbacks: First, each resource must be configured separately; second, this approach does not take into account the SFC and the end-to-end delays of the SFC. VNFs are provisioned first and then the network administrator selects the appropriate VNFs for chaining. This approach also does not consider resource utilization and energy saving. Our RL approach finds the appropriate embedding considering the SFC end-to-end delay and energy saving.

OpenStack also provides two other types of embedding, namely stacking and usage-aware embedding. In stacking, VNFs are stacked on the same server instead of spreading out. In utilization-aware embedding, we can configure the utilization ratio of CPU, Ram and Disk etc. Despite various configuration parameters, there is no comprehensive method for placement considering the entire SFC requirements, hence a Machine Learning based approach followed by EdgeDQN is justified. EdgeDQN-based embedding provides much better placement of VNFs compared to OpenStack default embedding.

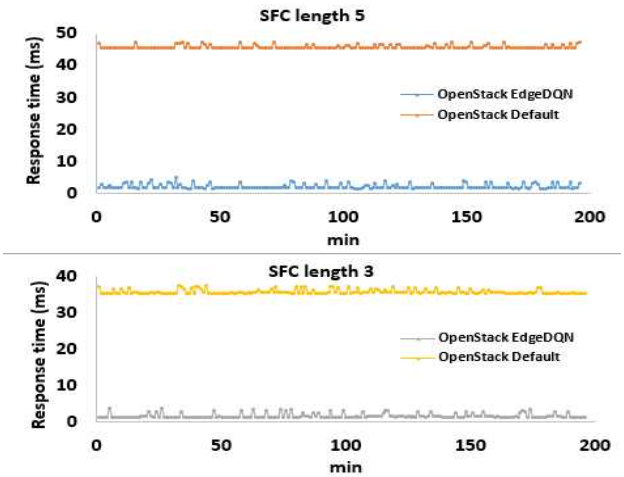

Figure 8: Response time, OpenStack default and EdgeDQN

C. *Discussion*

We see that EdgeDQN provides better sampling efficiency using the replay buffer, but longer convergence time. The learning time of EdgeDQN depends on the SFC length and the resource requirement of the SFC. When the SFC length was longer and the resource requirement was lower, the learning time was still lower, but when the resource requirement was high, the learning time was higher because

[1] OpenStack SFC, https://opendev.org/openstack/networking-sfc
[2] InfluxDB v1.7.9, https://github.com/influxdata/influxdb
[3] Collectd v5.8.1, https://github.com/collectd/collectd
[4] DEMU, https://github.com/ryousei/demu
[5] https://docs.openstack.org/nova/latest/admin/configuration/schedulers.html

it is difficult for the algorithm to place the SFC with very high resource requirement. There are several variations of RL such as A3C that can be adapted to further reduce the learning time. Tuning Hyperparameter should also be considered in the future. As the network size grows, we need to increase the number of episodes and decrease the gamma variable proportionally.

VI. CONCLUSION

Inspired by the use of DRL to solve resource management and planning problems, in this paper we attempted to enable DQN to do the SFCs placement nearby edge in a way that minimizes the E2E delay of each SFC and maximizes the resource utilization of underlying infrastructure. EdgeDQN algorithm divides the network into a hierarchy of local, neighbor and DC networks and prioritizes the local edge for placement. This approach can successfully deal with the large size of discrete action space. This will also enable renting neighbor and data center resources in case of resource scarcity at local CO. An intuitive reward model reduces latency and increases resource utilization, saving energy. Our evaluation on the physical testbed as well as the simulation network model proves that EdgeDQN is an effective model. We also found that the DRL algorithm should always be compared based on the cumulative reward, delay, and standard deviation. Comparing individual test cases cannot justify the efficiency of the algorithm due to its e-greedy nature. Based on the cumulative reward, delay and standard deviation of 420 different tests, we conclude that EdgeDQN provides the best placement for SFCs compared to simple DQN. Our OpenStack tests also prove the efficiency of EdgeDQN over the standard OpenStack placement options. In the future, we will extend this work to DDPG and A3C. We will also propose tuning of hyperparameters, including episodes, gamma, learning rate, and epsilon for all different DRL approaches. We will fine tune these parameters based on the size of the network and the type of algorithm.

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) and the ITRC (Information Technology Research Center) support program (IITP-2021-2017-0-01633).

REFERENCES

[1] M. Ersue, “Etsi nfv management and orchestration-an overview,” in Proc. of 88th IETF meeting, 2013.

[2] S. Lange et al., "A Network Intelligence Architecture for Efficient VNF Lifecycle Management," in IEEE Transactions on Network and Service Management, 2020, doi: 10.1109/TNSM.2020.3015244.

[3] X. Wei, S. Wang, A. Zhou, “MVR: An Architecture for Computation Offloading in Mobile Edge Computing,” in Proc. IEEE Int. Conf. Edge Comput. (EDGE), vol. 27, pp. 89-95, Sep. 2017.

[4] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer and X. Hesselbach, "Virtual Network Embedding: A Survey," in *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888-1906, Fourth Quarter 2013, doi: 10.1109/SURV.2013.013013.00155.

[5] OpenStack compute schedulers, https://docs.openstack.org/nova/latest/admin/configuration/schedulers.html

[6] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, “Network slicing and softwarization: A survey on principles, enabling technologies, and solutions,” IEEE Communications Surveys, vol. 20, no. 3, pp. 2429–2453, 3rd Quart., 2018.

[7] D. Lee, J. -H. Yoo and J. W. -K. Hong, "Q-learning based Service Function Chaining using VNF Resource-aware Reward Model," 2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS), Daegu, Korea (South), 2020, pp. 279-282, doi: 10.23919/APNOMS50412.2020.9236975.

[8] J. Pei, P. Hong, K. Xue and D. Li, "Efficiently Embedding Service Function Chains with Dynamic Virtual Network Function Placement in Geo-Distributed Cloud System," in IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 10, pp. 2179-2192, 1 Oct. 2019, doi: 10.1109/TPDS.2018.2880992.

[9] N. Tastevin, M. Obadia, and M. Bouet, “A graph approach to placement of service functions chains,” in Proc. IFIP/IEEE IM, May 2017, pp. 134–141.

[10] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, “Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions,” in Proc. IFIP/IEEE IM, May 2015, pp. 98–106.

[11] S. Pandey, J. W. -K. Hong and J. -H. Yoo, "Q-Learning based SFC deployment on Edge Computing Environment," 2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS), Daegu, Korea (South), 2020, pp. 220-226, doi: 10.23919/APNOMS50412.2020.9236981.

[12] S. Pandey, J. W. Hong and J. -H. Yoo, "Environment Aware Adaptive Q-Learning to Deploy SFC on Edge Computing," 2020 16th International Conference on Network and Service Management (CNSM), Izmir, Turkey, 2020, pp. 1-5, doi: 10.23919/CNSM50824.2020.9269046.

[13] M. Volodymyr at el. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[14] M. Dolati, S. B. Hassanpour, M. Ghaderi and A. Khonsari, "DeepViNE: Virtual Network Embedding with Deep Reinforcement Learning," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Paris, France, 2019, pp. 879-885, doi: 10.1109/INFCOMW.2019.8845171.

[15] P. T. A. Quang, Y. Hadjadj-Aoul and A. Outtagarts, "A Deep Reinforcement Learning Approach for VNF Forwarding Graph Embedding," in IEEE Transactions on Network and Service Management, vol. 16, no. 4, pp. 1318-1331, Dec. 2019, doi: 10.1109/TNSM.2019.2947905.

[16] Sutton, R. S., and Barto, A. G. Reinforcement learning: An introduction. MIT press, 2018.

[17] Y. Liu, Y. Lu, X. Li, W. Qiao, Z. Li and D. Zhao, "SFC Embedding Meets Machine Learning: Deep Reinforcement Learning Approaches," in IEEE Communications Letters, doi: 10.1109/LCOMM.2021.3061991.

[18] L. Chen, J. Lingys, K. Chen, and F. Liu, “AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization,” in Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM), 2018, pp. 191–205.

[19] D. Lee, J. -H. Yoo and J. W. -K. Hong, "Deep Q-Networks based Auto-scaling for Service Function Chaining," *2020 16th International Conference on Network and Service Management (CNSM)*, 2020, pp. 1-9, doi: 10.23919/CNSM50824.2020.9269107.