Deep Reinforcement Learning based VNF Management in Geo-distributed Edge Computing

Lin Gu*, Deze Zeng†, Wei Li‡, Song Guo§, Albert Y. Zomaya‡, Hai Jin*

*National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
†School of Computer Science, China University of Geosciences, Wuhan, Hubei, China
‡Centre for Distributed and High Performance Computing, School of Computer Science, The University of Sydney, Australia
§Department of Computing, The Hong Kong Polytechnic University, Hong Kong

Abstract—**Edge computing is an effective approach for resource provisioning at the network edge to host *virtualized network functions* (VNF). Considering the cost diversity in edge computing, from the perspective of service providers, it is significant to orchestrate the VNFs and schedule the traffic flows for *network utility maximization* (NUM) as it implies maximal revenue. However, traditional model-based optimization methods usually follow some assumptions and impose certain limitations. In this paper, inspired by the success of deep reinforcement learning in solving complicated control problems, we propose a *deep deterministic policy gradients* (DDPG) based algorithm. We first formulate the NUM problem with the consideration of end-to-end delays and various operation costs into a non-convex optimization problem and prove it to be NP-hard. We then redesign the exploration method and invent a dual replay buffer structure to customize the DDPG. Meanwhile, we also apply our formulation to guide our replay buffer update. Through extensive trace-driven experiments, we show the high efficiency of our customized DDPG based algorithm as it significantly outperforms both model-based methods and traditional non-customized DDPG based algorithm.**

Index Terms—**Deep Reinforcement Learning; VNF Orchestration; Flow Scheduling**

I. Introduction

It is a common pain point for network service providers to provision services in an efficient manner as it is time, cost, and resource consuming to deploy services at the hardware level. The heavy reliance on customized hardware severely hinders the development of computer networks. Modern networking industry highly demands lightweight service provision methods to foster network innovation and to drive long-term expenditure reduction. *Network function virtualization* (NFV) is thus proposed to take network functions (e.g., firewall, DPI, router) off the hardware, and to "softwarize" them such that they can run on common servers (e.g., x86 servers) as on-demand *virtualized network functions* (VNF) in cloud computing or edge computing infrastructure. Edge computing, which utilizes resources in the network edge, is considered as an ideal platform for VNF deployment. By deploying VNFs and processing network flows in geo-distributed edge servers close to the end users, the transmission delay and the traffic congestion on the Internet can be effectively reduced, thereby improving the user experience.

Along with NFV, *software defined networking* (SDN), by decoupling the control and data planes, allows network service providers to schedule the network traffic in a centralized manner on flow-level granularity. NFV and SDN together enable centralized network control and show great potential in promoting openness, innovation, flexibility, and scalability of networks. Both technologies therefore have attracted lots of interests in the networking communities. Many pioneering researchers have proposed different methods to better utilize the two technologies with different goals such as performance efficiency improvement [1], [2], cost reduction [3], [4], and *network utility maximization* (NUM) [5], [6]. From the network service provider perspective, as they usually rent resources from the infrastructure providers, one of the most important issues is the profit relating to both cost and revenue. The latter is further related to the *quality-of-service* (QoS), e.g., end-to-end delay, as declared in the *service level agreement* (SLA) between network service providers and consumers. Similar to cloud computing, edge computing also exhibits cost diversity. To pursue the goal of better user experience and higher profit, network service providers eagerly call for better intelligent control agents that consider both edge computing cost diversity and QoS.

When surveying the literature, we notice that existing optimization methods are mostly model-based, assuming that the network environment can be well modeled, or can be accurately predicted. However, real-world networks have become more complicated and highly dynamic, making it hard to model, predict, and control. Moreover, these model-based algorithms heavily rely on prior knowledge and are usually designed in an offline manner, sacrificing the flexibility feature of NFV and SDN. Even worse, some studies intentionally neglect some issues in order to simplify problem formulation or make the model easier to solve. In particular, we observe that many studies overlook the end-to-end delay, especially the intermediate processing delay, e.g., [2], [4]. While, the end-to-end delay is one of the most important metrics in the

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00097

SLA. But it is hard to accurately model, especially in the case of multi-hop transmission in "*service function chain*" (SFC) consisting of a set of ordered VNFs [1].

To make the algorithm more practical, we try to seek a model-free approach that can exploit the advantages of NFV and SDN, and can be applied online to adapt to the time-varying traffic demands. Recently, the success of AlphaGo has drawn considerable interest on artificial intelligence. The essence of AlphaGo is an improved version of reinforcement learning, i.e. *deep reinforcement learning* (DRL), which integrates deep learning into reinforcement learning to address more complicated control problems [7]. Thereafter, DRL has been widely applied in a variety of domains, with no exception to computer networking. Reinforcement learning, including DRL, has been successfully applied as an alternative promising model-free means to address many issues in computer networks, e.g., TCP congestion control [8], workload balancing [9], SDN routing [10], resource allocation [11], traffic optimization [12]. Similarly, we believe that DRL is also applicable for building intelligent control agents for VNF orchestration and flow scheduling.

Although a DRL based algorithm could be model-free, this does not mean that the model is totally useless. We note that *deep deterministic policy gradient* (DDPG), as a representative advanced DRL technique capable of dealing with continuous control problems (e.g., load balancing), requires an experience replay buffer to store the explored samples during the training process. Other than letting the DDPG agent naively update the buffer, we can use our model-based solution, although still based on some assumptions, to customize the experience replay buffer. Therefore, we are motivated to adopt such approach to design a customized DDPG based intelligent control agent. The main contributions of this paper are summarized as follows:

- We formally describe the online VNF orchestration and flow scheduling for NUM as a non-convex optimization problem and prove its NP-hardness by reducing from the multi-level uncapacitated facility location problem.
- Based on our formulation, we redesign the experience replay buffer structure in the DDPG algorithm. We also invent a bias exploration method and accordingly propose our customized DDPG-based VNF orchestration and flow scheduling algorithm. To the best of our knowledge, this is the first work to apply DRL for joint optimization of VNF orchestration and flow scheduling.
- By conducting extensive trace-driven experiments, the high efficiency of our customized DDPG based algorithm is verified by the fact that it significantly outperforms both existing model-based algorithms and traditional non-customized DDPG algorithm.

The remainder of the paper is organized as follows. Section II describes the system model, based on which the problem is formally formulated in Section III. Then, we propose our customized DDPG-based algorithm in Section IV. The trace-driven performance evaluation results are reported in Section V. Section VI discusses some related work. Finally, Section VII concludes our work.

II. Background and System Model

A. Background

In this paper, we consider the scenario that a service provider provides a set $\mathbb{S}$ of different network services by renting resources from the edge computing infrastructure provider. The edge computing infrastructure consists of a set $\mathbb{N}$ of networked edge servers and is represented as an indirected graph $G_n = (\mathbb{N}, \mathbb{E}_n)$, where $\mathbb{E}_n$ is a set of links between the edge servers. A network service $s \in \mathbb{S}$ is provided in the form of SFC as a set of ordered network functions. Therefore, a service s can be represented as a directed graph $G_s = (\mathbb{F}_s, \mathbb{E}_s)$, where $\mathbb{F}_s$ denotes the set of functions required by service s and $e_{ff'} \in \mathbb{E}_s$ enforces the flow traverse sequence between the functions. Specifically, we assume that there is a proxy server as the origin function $o_s \in \mathbb{F}_s$ for service s. Any network flow requesting service s shall start from o_s, sequentially traverse over all the functions defined in $\mathbb{F}_s$ and eventually reach the destination function to complete the service.

A network function f could have one or multiple VNF instances on different edge servers in the infrastructure. Different network functions are with different preferred software and hardware specifications. For example, to setup a Bro IDS usually requires a $c4.2large$ IDS VNF [13]. Launching a VNF instance of type f commonly involves transferring a VM image containing the VNF instance to the hosting server, and then booting the VM image with a setup cost Φ_f^{SU} [14]. Moreover, once a VNF f on edge server n is set up, the service provider needs to pay for the VNF operation at a unit price of Φ_{fn}^{OP}, determined by VNF type, edge server location, and operation time. Geo-distributed edge computing shall have similar charging policy as cloud computing and therefore also exhibits cost diversity. For example, the recommended running environment for a Bro IDS is an 8-core processor pricing at $\$0.398$ per hour in Ohio while $\$0.504$ per hour in Tokyo in Amazon EC2 [15]. Due to the cost geo-diversity, the hosting server location will have deep influence on the operation cost.

To reduce the operation cost, it seems that we may aggressively choose the edge servers with the lowest unit price to host the VNFs. However, another non-negligible issue to the overall cost is the communication cost as the flows will traverse the VNFs hosted on different edge servers. The hosting server location affects the flow scheduling and hence the communication cost. For example, the communication price of Amazon EC2 is $\$0.01/GB$ and $\$0.120 - 0.200/GB$ for the same geographic region and different regions, respectively [16]. Geo-distributed edge computing shall also have the same communication cost diversity. To describe such phenomenon, Φ_{mn}^{TR} is used to denote the unit communication cost on link $e_{mn} \in \mathbb{E}_n$ between the edge servers m and n. Specifically, we define $\Phi_{nn}^{TR} = 0, \forall n \in \mathbb{N}$, indicating that no cost is incurred if the transmission is within the same edge server. Note that, a network function may have multiple VNFs on different edge servers and the network flows shall be carefully balanced. Also, one VNF may be shared by different SFCs with light traffic load and the flows for these SFCs shall be merged at the VNF. Both the above two issues refer to flow scheduling.

From the perspective of network service providers, it is of course important to lower the overall cost. However, low cost does not always mean high profit as the profit also relates to the revenue, which further relates to the QoS declared in the SLA. It is widely agreed that the end-to-end service delay is one of the most important QoS metrics. A network flow must traverse and be processed by all ordered VNFs in the SFC. Therefore, the end-to-end delay of a flow is the sum of the processing delay on all required VNFs. Without loss of generality, we use μ_{fn} to denote the average processing rate for VNF of type f on edge server n. As widely known, queuing theory, e.g., M/M/1, M/D/1, G/D/1, can be applied to estimate the processing delay on one VNF, but it is not accurate enough. Even worse, it is hard to solve an optimization model including the end-to-end delay as the sum of processing delay on sequential queues [1].

B. Runtime VNF Orchestration and Flow Scheduling

We consider a discrete time period $\mathbb{T} = \{1, 2, 3, .., T\}$. In practice, the user requests for a specific service vary over the time. Let $R^s(t)$ be the flow rate requesting service s at time slot t, i.e. arriving rate at o_s. Therefore, both VNF orchestration and flow scheduling shall be operated in an online manner at run-time. The run-time operation in the control agent is conducted at the beginning of each time slot. VNF orchestration refers to the activation and deactivation of the deployed VNFs. Flow scheduling is not only about the routing path planning but also about the load balancing when one network function has multiple instances in the infrastructure.

To clearly tract the flow relationship between the edge servers for any SFC, we build a directed VNF graph $G_v = (\mathbb{V}, \mathbb{E}_v)$ based on the SFC graph G_s and infrastructure graph G_n. The VNF graph is built according to the following rules. We define $v_{fn} \in \mathbb{V}$ to present a VNF of type f on edge server n. That is, for a network function f, whenever there is an instance on edge server n, a vertex v_{fn} is introduced in G_v. Note that, there is only one origin function o_s for each service s. Therefore, it is directly introduced into G_v. For each SFC s, strictly following the VNF sequence, a network flow processed by $v_{f'n'}$ shall be passed to the next VNFs of type f in order as defined in the service graph G_s. Therefore, a directed link $e_{(v_{f'n'}, v_{fn})}$ is added in set $\mathbb{E}_v$ for each $v_{fn} \in \mathbb{V}$ whenever $e_{f'f} \in \mathbb{E}_s, \forall s \in \mathbb{S}$.

The essence of VNF orchestration is to decide whether the VNF of type f on edge server n shall be activated or not at time slot t. Supported by the VNF graph, we can represent such activation status by introducing binary variables as

$$x_{fn}(t) = \begin{cases} 1, & \text{if VNF of type } f \text{ on edge server } n \\ & \text{is activated at time slot } t, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Accordingly, the essence of flow scheduling is to allocate the traffic load on the edges $\mathbb{E}_v$ as the VNF graph already implies the processing sequence.

At run-time, we shall decide the values of $x_{fn}(t)$ as well as the flow value on each edge in $\mathbb{E}_v$ according to the flow rates requesting for different SFCs so as to achieve the goal of NUM.

III. Problem Formulation

In this section, we formulate the NUM problem into a non-convex optimization form for a formal understanding of the problem and then prove it as NP-hard even in the case with known service request rates and without the consideration of end-to-end delay.

A. Flow Scheduling

For any flow requesting a service, its flow rate must be always reserved across any edge server visited, i.e. network flow conservation. Intuitively, we can express the flow conservation using linear programming. However, without known VNF activation status, it is hard to describe the flow relationship between the edge servers in G_n directly. Fortunately, with the introduction of VNF graph, we can instead express the flow relationship between the vertex in G_v, regardless of actual VNF activation status.

For any service s, a constituent network function f may have multiple VNFs and the flow requesting s may be distributed to these VNFs. As we do not know the activation status of the VNFs yet, any link in $\mathbb{E}_v$ could be allocated with certain traffic requesting the service. We define variable $r_{e_{(v_{f'n'}, v_{fn})}}(t), \forall e_{(v_{f'n'}, v_{fn})} \in \mathbb{E}_v$ as the flow rate on link $e_{(v_{f'n'}, v_{fn})}$. Then, we can express the network flow conservation on a service basis, as follows.

According to the network flow conservation theory, the total egress flow amount shall be equal to its ingress flow. As a result, for any intermediate network function f hosted on edge server n, we shall have

$$\begin{aligned} \sum_{\forall e_{(v_{f'n'}, v_{fn})} \in \mathbb{E}_v} & r_{e_{(v_{f'n'}, v_{fn})}}(t) = \\ \sum_{\forall e_{(v_{fn}, v_{f''n''})} \in \mathbb{E}_v} & r_{e_{(v_{fn}, v_{f''n''})}}(t), \forall v_{fn} \in \mathbb{V} \end{aligned} \tag{2}$$

where $v_{f'n'}$ and $v_{f''n''}$ are a parent vertex and a child vertex of v_{fn} in the VNF graph, respectively. Note that, we do not distinguish the service in the intermediate VNFs. This is because one VNF could be shared among different SFCs. Once the flow conservation is guaranteed, we can ensure that all flows get completely served. For the origin function, we shall distinguish the service as each service has one unique origin function. The total egress flow amount from the origin function of each service must be equal to the request rate, i.e.

$$\sum_{e_{(o_s, v_{(f,n)})} \in \mathbb{E}_v} r_{e_{(o_s, v_{(f,n)})}}(t) = R^s(t), \forall s \in \mathbb{S} \tag{3}$$

B. VNF Orchestration

The VNF orchestration is mainly about the activation and deactivation of the VNFs of each SFC. To pursue the goal of NUM, only a portion of VNFs, not all of them, shall be activated according to the real-time traffic flow rate, provided that the SLA is obeyed. Anyhow, if a flow passes through a VNF f hosted on edge server n, the VNF must be activated to process the flow, i.e. $x_{fn}(t) = 1$. On the other hand, it would be better to make $x_{fn}(t) = 0$ for operation cost reduction when there is no flow to be processed. Let $r_{fn}(t)$ denote the total flow rate to VNF f on edge server n, i.e.

$$r_{fn}(t) = \sum_{f', e_{(v_{f'n'}, v_{fn})} \in \mathbb{E}_v} r_{e_{(v_{f'n'}, v_{fn})}}(t), \forall v_{(f,n)} \in \mathbb{V} \quad (4)$$

Thus, we can easily express the relationship between $x_{fn}(t)$ and $r_{fn}(t)$ in a linear form as

$$\frac{r_{fn}(t)}{L} \le x_{fn}(t) \le r_{fn}(t) \cdot L, \forall v_{fn} \in \mathbb{V} \quad (5)$$

where L is an arbitrary large number. The setup cost is incurred only when a VNF needs to be activated but is inactive in the previous time slot. As a result, we can calculate the total setup cost as

$$C^{SU}(t) = \sum_{v_{fn} \in \mathbb{V}} \max\{x_{fn}(t) - x_{fn}(t-1), 0\} \cdot \Phi_f^{SU} \quad (6)$$

from which we shall see that the VNF activation decision not only depends on the previous decisions, but also affects the subsequent decisions.

If a VNF is activated on an edge server during a time slot, certain operation cost will be charged by the infrastructure provider. The total operation cost therefore can be written as

$$C^{OP}(t) = \sum_{v_{fn} \in \mathbb{V}} x_{fn}(t) \cdot \mu_{fn} \cdot \Phi_{fn}^{OP} \quad (7)$$

Meanwhile, the communication cost for transferring the traffic flow from the servers hosting its parent function will be also counted in as

$$C^{TR}(t) = \sum_{e_{(v_{f^*n^*}, v_{fn})} \in \mathbb{E}_v} r_{e_{(v_{f^*n^*}, v_{fn})}}(t) \cdot \Phi_{n^*n}^{TR} \quad (8)$$

C. End-to-End Delay and Utility

As mentioned above, the end-to-end delay of a flow requesting service s is defined as the sum of the processing delay on each constituent VNF of the SFC. Thus, the average end-to-end delay of a flow requesting service s can be computed as

$$d_s(t) = \sum_{f \in \mathbb{F}_s} d_{fn}, \forall s \in \mathbb{S} \quad (9)$$

where d_{fn} is the average processing delay for VNF f on edge server n. Although it can be estimated via applying various queuing models, none of them gives an accurate value in practice. Even based on queuing theory, we can estimate it in closed-form as a function of the processing rate μ_{fn} and request rate r_{fn}, e.g., $d_{fn}(t) = \frac{1}{\mu_{fn} - r_{fn}(t)}$ with M/M/1 model, it is still non-convex, no matter which model is applied. This is the reason that end-to-end delay is rarely included in the optimization models in the existing work.

Nevertheless, to build a formal description on the problem studied in this paper, we still take into account of the end-to-end delay as it is closely relevant to the network utility, even without closed-form expression. We follow the classical model defined in [17] and define the revenue function as:

$$U_s^{SLA}(t) = P_s \cdot R^s(t) - \log d_s(t), \forall s \in \mathbb{S} \quad (10)$$

where P_s is the expected service payment from consumers according to the SLA. The service provider receives full payment when the delay is small. With the increase of service delay, less payment can be obtained. Now, we can calculate the network utility by subtracting the total cost from the revenue, i.e.

$$U(t) = \sum_{s \in \mathbb{S}} U_s^{SLA}(t) - C^{SU}(t) - C^{OP}(t) - C^{TR}(t) \quad (11)$$

D. NUM Problem Formulation

Summing up all the issues, we can finally formulate the VNF orchestration and flow scheduling for the NUM problem as

Overall-NUM:

$$\max : \sum_{t \in \mathbb{T}} U(t),$$
$$\text{s.t.} : (1), (2), (3), (5).$$

Due to the involvement of non-convex expression $U(t)$, it is obvious that the problem described above is a non-convex optimization problem, which is hard to solve, i.e. at least NP-hard. Next, we will formally show that it is NP-hard.

E. Hardness Proof

Intuitively, we can greedily maximize the NUM of each time slot, i.e. Instant-NUM with known flow rates and VNF activation statuses, at run-time to approach the Overall-NUM. While, we will see that even such Instant-NUM is NP-hard.

Theorem 1: The VNF orchestration and flow scheduling for the NUM problem in each time slot is NP-hard.

Proof: The k-level (or multilevel) uncapacitated facility location problem is one of the classical generalizations of the uncapacitated facility location problem, and usually occurred in modeling supply chains, e.g., how to locate warehouses and distribution centers in a hierarchical distribution network [18]. The definition of the k-level uncapacitated facility location problem is that: given K collections S_k of facilities for $|K|$ levels, and a set of clients J. Each client $j \in J$ has to be served by a sequence of $|K|$ open facilities in order, one from each level $(k, k-1, ..., 1)$. That is, for each level, at least one facility must be selected from each facility collection. The goal is to open a subset of the facilities that minimize the sum of the total opening cost of the facilities and the total communication cost of the paths between facilities.

We assume that the VNFs can provide sufficient resources and the end-to-end delay is negligible, so we have $U_s^{SLA}(t) =$

$-C^{SU}(t) - C^{OP}(t) - C^{TR}(t)$. Then the NUM problem can be transformed into a cost minimization problem as

Instant-NUM:

$$\min : \sum_{t \in \mathbb{T}} C^{SU}(t) + C^{OP}(t) + C^{TR}(t),$$
$$\text{s.t.} : (1), (2), (3), (5).$$

In one time slot, the network flow of each service $s \in S$ should be served by an ordered set $\mathbb{F}_s$ of VNFs, i.e. $|\mathbb{F}_s|$-level. For each level, we must activate at least one instance as multiple VNF instances in the same level may coexist on different edge servers. We shall minimize the activation cost of the VNFs and the communication cost between the edge servers. This is exactly a multi-level uncapacitiated facility location problem, which has been proved as NP-hard [18]. ■

IV. Customized DDPG-based Algorithm Design

There are many available DRL algorithms (e.g., DQN, DDPG, A3C) with different features. As the NUM problem involves both discrete control (i.e. VNF activation and deactivation) and continuous control (i.e. flow scheduling), we choose DDPG capable of continuous control as the base algorithm. Although the Instant-NUM problem is NP-hard, fortunately we notice that it is in *integer linear programming* (ILP) form with binary variables, which has been extensively studied. Many heuristic algorithms (e.g., branch-and-bound) and solvers (e.g., Gurobi Optimizer) are available. Furthermore, we will see that the solution actually can be used to customize the DDPG training algorithm such that it can better fit our NUM problem.

The basic DDPG training procedure can be viewed as a combination process of policy-based and value-based methods. The DDPG agent learns the optimum policy and its value function through interactions with the network environment. The agent is composed of two parts: the actor network (i.e. policy) and the critic network (i.e. estimated value function). The role of the actor is to define parameterized policy and generate actions (e.g., VNF activation and flow scheduling) according to the observed network state (e.g., network topology, current VNF activation, flow rate), while the critic is in charge of evaluating current action considering the reward received from the network. In detail, the critic produces a *temporal difference error* (TD-error) which indicates whether current actions are getting better or worse than expected, and then adjusts both the actor and the critic accordingly to reduce the TD-error mostly. Therefore, actor-critic methods typically perform well in learning continuous-valued stochastic policies. Nevertheless, to utilize the DDPG algorithm, we shall first accurately define state, action, and reward.

A. Definition of State, Action, and Reward

The essential elements required by any DRL, including DDPG, can be described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, referring to the state space, action space and reward space, respectively. Upon the state $\mathcal{S}$, after taking action $\mathcal{A}$, the agent shall observe a new state $\mathcal{S}'$, and corresponding reward $\mathcal{R}$ can be calculated to judge the effectiveness of action. As a result, we define

Algorithm 1 Customized DDPG-based Agent Training

```
1: Randomly initialize critic network $Q(\mathcal{S},\mathcal{A}|\theta^Q)$ and actor
   $\pi(\mathcal{S}|\theta^\pi)$ with weights $\theta^Q$ and $\theta^\pi$
2: Initialize target network $Q'$ and $\pi'$ with weights $\theta^{Q'} \leftarrow$
   $\theta^Q$, $\theta^{\pi'} \leftarrow \theta^\pi$
3: Initialize global replay buffer $R_g$ and baseline buffer $R_b$
4: Receive the initial observed state $s(0)$
5: for $t = 1$ to $T_{episode}$ do
6:     Apply the extreme case aware exploration method to
       obtain action $\mathcal{A}(t)$
7:     Execute action $\mathcal{A}(t)$, observe reward $\mathcal{R}(t)$ and new state
       $s(t+1)$
8:     Solve Instant-NUM and get the utility as $\mathcal{R}'(t)$
9:     if $\mathcal{R}(t) \le \mathcal{R}'(t)$ then
10:        $(\mathcal{S}(t),\mathcal{A}(t),\mathcal{R}(t),\mathcal{S}(t+1)) \rightarrow$ global buffer $R_g$ and
           baseline buffer $R_b$
11:    else
12:        $(\mathcal{S}(t),\mathcal{A}(t),\mathcal{R}(t),\mathcal{S}(t+1)) \rightarrow$ global buffer$R_g$
13:    end if
14:    Sample a random mini-batch of $N$ transitions
       $(\mathcal{S}_i,\mathcal{A}_i,\mathcal{R}_i,\mathcal{S}_{i+1})$ from $R_g$ and $R_b$
15:    for $n = 1$ to $N$ do
16:        $y_n = \mathcal{R}_n + \gamma Q'(\mathcal{S}_{n+1}, \pi'(\mathcal{S}_{n+1}|\theta^{\pi'})|\theta^{Q'})$
17:    end for
18:    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i(y_n -$
       $Q(\mathcal{S}_n,\mathcal{A}_n|\theta^Q))^2$
19:    Update the actor policy using the sampled policy gra-
       dient:
$\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_{\mathcal{A}} Q(\mathcal{S},\mathcal{A}|\theta^Q)|_{\mathcal{S}=\mathcal{S}_n,\mathcal{A}=\pi(s_n)} \nabla_{\theta^\pi}\pi(\mathcal{S}|\theta^\pi)|_{\mathcal{S}_n}$
20:    Update the target networks:
           $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
           $\theta^{\pi'} \leftarrow \tau\theta^\pi + (1-\tau)\theta^{\pi'}$
21: end for
```

the state, action and reward associated with our problem as follows:

$\mathcal{S}$: The state $\mathcal{S}(t)$ at time t is represented by a vector consisting of the flow rate of each service and the current VNF activation statues, i.e. $\mathcal{S}(t) = [(R^s(t), x_{fn}(t)), \forall s \in \mathbb{S}, f \in \mathbb{F}, n \in \mathbb{N}]$.

$\mathcal{A}$: The action shall include both VNF activation and flow scheduling. According to (5), we see that the VNF activation can be derived from the flow scheduling. Therefore, we only explicitly define the action on flow scheduling as $\mathcal{A}(t) = [r_e^s(t), \forall e \in \mathbb{E}_v]$, where $\sum_{e \in \mathbb{E}_v} r_e^s(t) = R^s(t)$, leaving the VNF activation and deactivation as hidden actions.

$\mathcal{R}$: The reward we received at time t is set as the objective of our NUM problem defined in (11), i.e. $\mathcal{R}(t) = U(t)$.

B. Problems in General DDPG Training

Incorporating the above definitions, we start to design our DDPG based algorithm. The normal training process of DDPG algorithm is to first generate a bulk number of transition samples in the format of $(\mathcal{S}(t), \mathcal{A}(t), \mathcal{R}(t), \mathcal{S}(t+1))$ in the replay buffer via exploration, and then draw some samples as mini-batch from the reply buffer to train the actor and critic networks.

We got the following findings on our initial trial on directly applying the DDPG algorithm.

- DDPG uses a normal-distribution-based exploration method to explore an action $\mathcal{A}(t)$ as a real number within the range of $[0, R^s(t)]$. However, during the experiments, we find that most actions $\mathcal{A}(t)$ generated by normal-distribution-based exploration are in the range of $(0, R^s(t))$. That is, the case without load balancing is seldom explored and learned.
- DDPG randomly draws the samples from the relay buffer for the training of actor and critic networks. Our experiments show that it may be trapped into bad samples with low utility. Moreover, due to the relay buffer size limitation, the bad samples may even occasionally extrude the good samples from the buffer. Both may lead to slow convergence and bad performance.

To tackle the above problems, we customize the DDPG algorithm by redesigning the exploration method and the replay buffer structure.

C. Customized DDPG Algorithm Design

1) Extreme Case aware Exploration: The inexperienced agent shall see sufficient transition samples to become intelligent for good decision making. But there are an infinite number of actions in continuous control problems, it is impossible to see all possible actions. Some potentially good actions may be easily overlooked due to the normal-distribution based exploration, i.e. the probability of exploring an action as either 0 or $R^s(t)$ approaching 0. In practice, it is common to activate only one instance of each constituent network function of an SFC, i.e. without load balancing, when the flow rate is low or even totally deactivate the whole chain when there is no request. Motivated by such fact, we propose a new extreme case aware exploration to ensure the actions of 0 and $R^s(t)$ can also be fairly explored and learned. Firstly, we randomly generate an integer i in range $[1, I]$, where I is a parameter to be set according to the estimated workload. Then, we explore $\mathcal{A} = 0$ and $\mathcal{A} = R^s(t)$ if $i = 1$ and $i = I$, respectively. For the other cases, normal-distribution-based exploration is still applied. Thus, the probabilities of exploring actions $\mathcal{A} = 0$ and $\mathcal{A} = R^s(t)$ are both $1/I$.

2) Dual Replay Buffer: The second problem mentioned above is incurred by indistinguishably storing all the transition samples in the replay buffer. Therefore, we propose a dual replay buffer structure consisting of a global buffer and a baseline buffer. The global buffer is the same as the replay buffer in the general DDPG design. The baseline buffer is particularly used to store the good samples according to our understanding of the problem. Specially, we update the baseline buffer with the help of model-based solutions. Although the Instant-NUM overlooks the end-to-end delay, it still provides potential solutions with good utility and therefore can be used as baselines. If the utility of the explored action is even worse than baseline, we only update the global replay buffer as normal. Otherwise, we update both buffers such that the baseline buffer can store some good samples. Accordingly, we also redesign our mini-batch policy. Other than drawing samples only from the global replay buffer, we enforce it with some samples from the baseline buffer. To this end, we define a parameter β ranging in $(0, 1)$. Then, a mini-batch in size N is constructed by drawing $N\beta$ samples from the baseline buffer and $N(1-\beta)$ from the global buffer. By such means, we can make sure that some potentially good samples are included in the mini-batch.

3) Algorithm Design: Incorporating the above design, we summarize our customized DDPG-based agent training algorithm in Algorithm 1. We first randomly set the weights of actor network $\pi(s|\theta^\pi)$ and critic network $Q(s, a|\theta^Q)$ as θ^π, and θ^Q, respectively (line 1). As for the target networks π' and Q', they are cloned from the actor and critic networks (line 2). To support experience-based learning and avoid bad sample trapping, we construct our dual replay buffer as R_g and R_b in line 3.

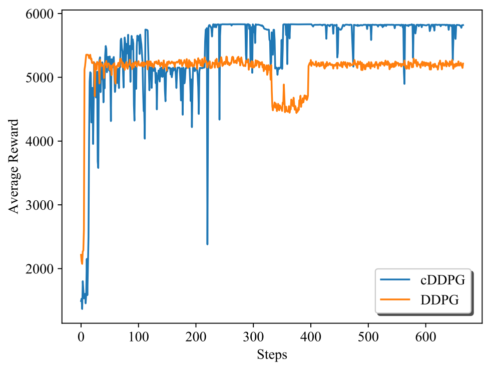
Then, we kick off the agent training with initial state $\mathcal{S}(0)$. At the beginning of each episode, we first apply our extreme case aware exploration method to obtain an action $\mathcal{A}(t)$, as shown in line 6. This action $\mathcal{A}(t)$ is executed to get a reward $\mathcal{R}(t)$ and new state $\mathcal{S}(t+1)$ in line 7. Next, we solve Instant-NUM to obtain a reference reward and apply our dual buffer update policy to update both global buffer R_g and baseline buffer R_b with transition sample $(\mathcal{S}(t), \mathcal{A}(t), \mathcal{R}(t), \mathcal{S}(t+1))$ in lines 8-13. After accumulating enough transition samples in R_g and R_b, we sample a mini-batch of N transition samples from both buffers to train the actor and critic networks in line 14. For the nth transition sample $(\mathcal{S}(n), \mathcal{A}(n), \mathcal{R}(n), \mathcal{S}(n+1))$, we calculate target value $y(n)$ from target actor network $\pi'(\mathcal{S}(n+1))$ and use the average value $\frac{1}{N}\sum_{n \in N} y(n)$ to update the critic networks (lines 16-18). Finally, we compute the policy gradient in line 19 and update the target networks with update rate τ in line 20. The above procedures iterate until convergence or reaching the predefined episode bound.

Once the control agent gets well trained after convergence, it shall already witness a sufficient number of samples and become experienced to make the right decision towards maximal network utility according to the real-time observations (i.e. state). We can embed the intelligent agent in the central controller to orchestrate the VNFs and schedule the flows at run-time, based on the information collected from the network.

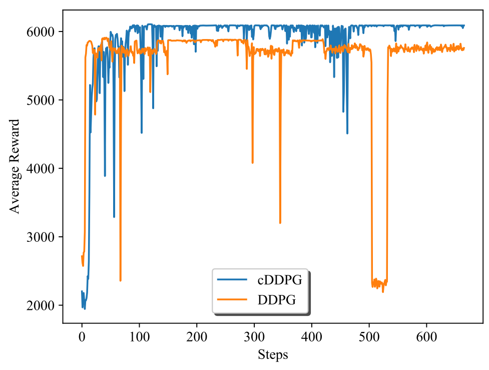
V. Performance Evaluation

To evaluate the performance of the proposed *customized DDPG* ("cDDPG") based VNF orchestration and flow scheduling algorithm, we conduct extensive trace-driven simulations and report the results in this section.

To verify the correctness of our newly designed exploration method and dual buffer structure, we compare cDDPG with non-customized DDPG based algorithm ("DDPG"). Besides,



(a) On Cogent's network



(b) On ARPA network

Fig. 1. Average reward during the training process

we also realize the algorithm ("NUM") by solving the Instant-NUM to maximize the utility in each time slot and the rule-based algorithm "POLAR" [19] which combines both online learning method for workload prediction and model-based optimization for scheduling decision making. The neural networks, employed in both cDDPG and DDPG, are implemented using the Tensorflow framework. For the bias exploration in cDDPG, we set $I = 10$ in our experiments. All the experiments are conducted on a server equipped with a 2.6GHz 8-Core Intel Xeon CPU E5-2670 processor. We consider two well-known real network topologies: Cogent's Network [20] and ARPA Network [21], consisting of 43 and 47 nodes, respectively, as the physical infrastructure. We view each node as an edge server in our experiments. We embed 10 different service chains consisting of 30 types of network functions. Each network function has 1-3 VNFs hosted on different edge servers. We adopt the dataset in [22] as the base trace to mimic the time-varying traffic demands.

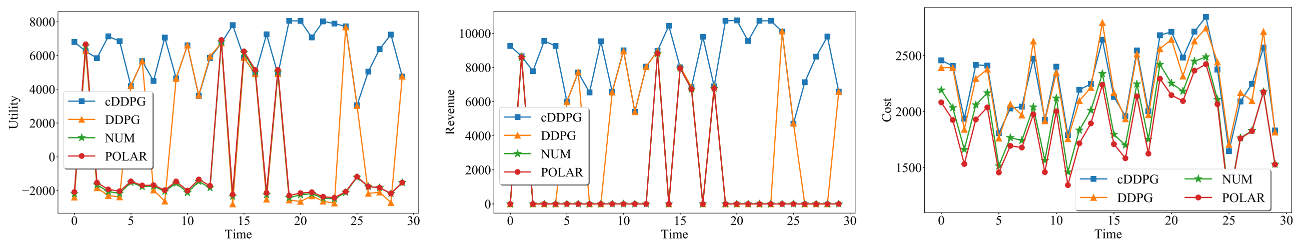
The first step of both cDDPG and DDPG is to train the control agent with respect to the network environment including the network topology, the SFC structure, and the VNF deployment. Based on the network environment described above, we train our agent for $600,000$ episodes. In order to make each agent observe a sufficient number of samples, the rates of the flows requesting the 10 services are randomly generated in range $[0, 11]$ during each episode. The main differences between cDDPG and DDPG is in the training process as we redesign the exploration method, replay buffer structure, and the mini-batch policy. To check whether our design does take effect, we check the average rewards during the training process on Cogents network and ARPA network in Fig. 1(a) and Fig. 1(b), respectively. Each point plotted in Fig. 1 is averaged over 900 episodes.

From Fig. 1, we can see that the reward of either cDDPG or DDPG gradually converges with the increasing number of training episodes on both network topologies. For example, the average rewards of DDPG and our cDDPG on Cogent's network are basically stable after about $200,000$ and $450,000$ training episodes, respectively, as shown in Fig. 1(a). One notable finding in Fig. 1(a) is that during the $340,000$ to $400,000$ episodes, the reward of DDPG suddenly drops. Similar observation can also be found in Fig. 1(b) during $500,000$-$510,000$ episodes with ARPA network topology. This is attributed to the trapping in a local optimal due to the normal-distribution based exploration and non-distinguishable replay buffer design. Such phenomenon is barely seen in the training process of cDDPG. This verifies that our bias exploration and dual buffer structure design effectively address the problems of traditional DDPG as mentioned in Section IV-B. The results from the training process imply that the cDDPG agent shall be more intelligent than the DDPG agent as it gets higher reward after $600,000$ episodes of training.

Next, we apply the well-trained agents to practically orchestrate the VNFs and schedule the flows to check how they perform. In addition, we include the performance evaluation results for comparison in this group of experiments. For each topology, we obtain the instant results during each time slot when the flow rate upper bound is set as 11 and calculate the average results under different flow rate upper bounds for 900 time slots. The results are reported in Figs. 2-5.

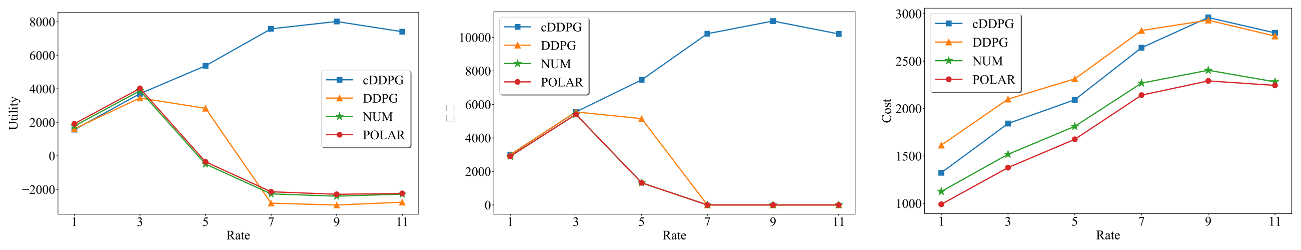
Let us first check the instant performance from Fig. 2 and Fig. 4. Figs. 2(a) and 4(a) give the instant utility during the first 30 time slots on Cogent's network and ARPA network, respectively. In most time slots, cDDPG achieves the maximal utility as desired, implying that a well trained cDDPG agent does indeed well control the network. NUM and POLAR almost always perform the worst, but occasionally with the best performance. DDPG looks better but never surpasses cDDPG.

To get more insightful understanding of the above phenomenon, we detail the instant revenue and cost in Figs. 2(b), 2(c), 4(b) and 4(c). It can be observed from Fig .2(b) and Fig. 4(b) that NUM and POLAR always give the lowest revenue because they fail to capture the end-to-end delay. DDPG performs better for its capability in incorporating the end-to-end delay. But it is not as good as cDDPG due to the not-so-good training, as reported in Fig. 1. The well-trained cDDPG agent always gets the highest revenue as it



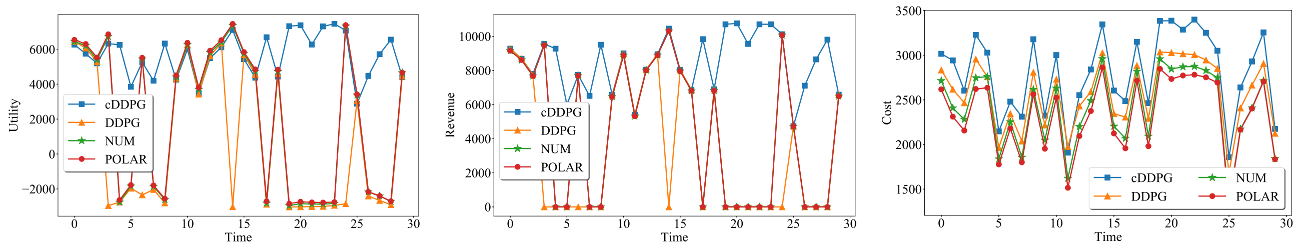
(a) Instant utility (b) Instant revenue (c) Instant cost

Fig. 2. Instant performance during the first 30 time slots on Cogent's network when the flow rate upper bound is 11



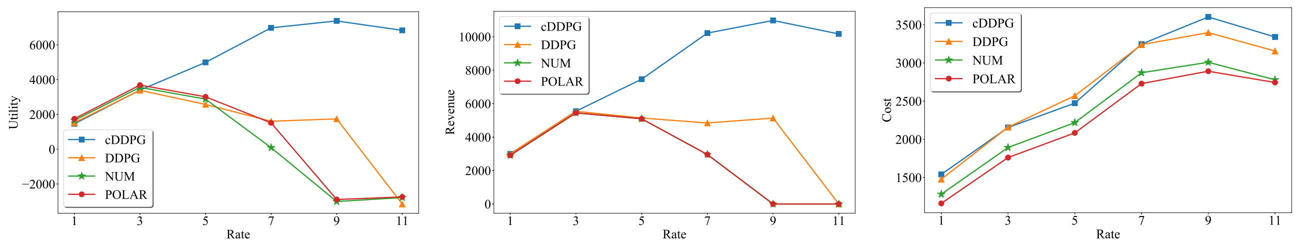
(a) Average utility (b) Average revenue (c) Average cost

Fig. 3. Average performance on Cogent's network when the flow rate upper bound varies from 1 to 11



(a) Instant utility (b) Instant revenue (c) Instant cost

Fig. 4. Instant performance during the first 30 time slots on ARPA network when the flow rate upper bound is 11

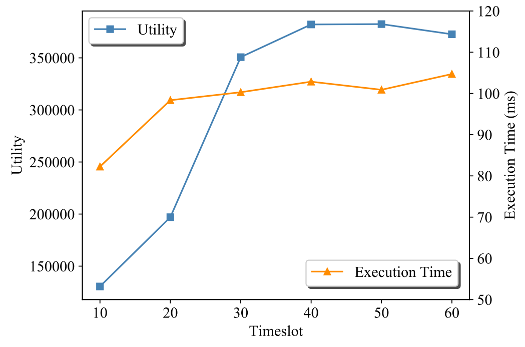


(a) Instant utility (b) Instant revenue (c) Instant cost

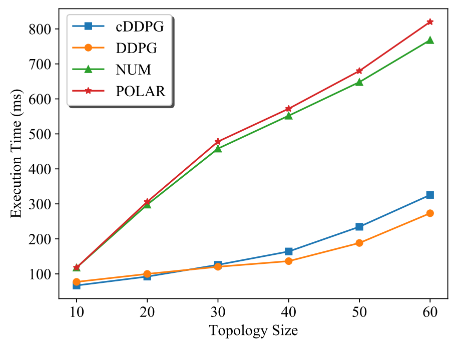
Fig. 5. Average performance on ARPA network when the flow rate upper bound varies from 1 to 11

effectively schedules the flows to experience the lowest end-to-end delay. However, when it comes to the instant cost, we can see from both Figs. 2(c) and 4(c) that NUM and POLAR always achieve the lowest cost. When the traffic demand is low, even the lowest end-to-end delay does not take much revenue, the cost is more dominant. This explains why NUM and POLAR occasionally achieve the maximal utility. Nevertheless, cDDPG can always well balance the tradeoff between the revenue and the cost to get the maximal utility in most cases.

Then, we investigate the long-term performance of all four algorithms when the flow rate upper bound increases from 1 to 11. The average results on Cogent's and ARPA networks are reported in Fig. 3 and Fig. 5, respectively. From Fig. 3(a) and Fig. 5(a), we can see that the average utility obtained by either algorithm first increases and then decreases with the increasing of flow rate. Initially, although more workload leads to higher cost, as shown in Fig. 3(c) and Fig. 5(c), more revenue can also be obtained if the flows get scheduled effectively as shown in Fig. 3(b) and Fig. 5(b). Under any flow rate, cDDPG always



(a) Utility and execution time under different number of time slots



(b) Execution time under different network size

Fig. 6. Utility and execution time under different number of time slots and network sizes

performs better than DDPG because it is trained better thanks to the bias exploration and dual buffer design. An interesting phenomenon is that when the flow rate is very small NUM and POLAR can deliver the best performance, as shown in Figs. 3(a) and 5(a). The reason is that when the flow rate is very small, the cost dominates the utility and the model-based NUM and rule-based POLAR can accurately find the optimal solution with the lowest cost. However, when the flow rate becomes larger, both cDDPG and DDPG outperform NUM and POLAR as the revenue influences the utility more than the cost. With further increasing of the flow rate, the processing delay on a VNF is substantially increased, leading to severe degradation of the revenue according to the definition in (10). As a result, the utility of either algorithm on both topologies starts to decrease.

Finally, we discuss how our cDDPG algorithm performs with different numbers of time slots and network sizes as shown in Fig. 6. First, we averagely divide the time period T into multiple time slots varying from 10 to 60. The corresponding long-term utility and total execution time of cDDPG are reported in Fig. 6(a). It can be observed that the execution time slightly increases with the number of time slots since the agent needs to make decisions at the beginning of each time slot. The long-term utility first shows an increase trend and then converges. The reason is that, with less time slots, the time slot interval is relatively long and the dynamics of network flow cannot be accurately captured. The flow scheduling decisions made at the beginning of each time slot cannot deal with the rate fluctuation thereafter. As the number of time slot increases, the interval of each time slot decreases, so does the flow rate fluctuation level. After the number of time slot researches 30, the agent can accurately capture the flow rate fluctuation and make proper actions, hence the utility of cDDPG gradually converges. Then, we check the scalability of cDDPG by evaluating the execution time of NUM, POLAR, DDPG and cDDPG on different network sizes from 10 to 60. We can see from Fig. 6(b) that model-free algorithms (i.e. cDDPG and DDPG) always require less execution time than the model-based algorithms (i.e. NUM and POLAR). cDDPG only slightly requires a little more execution time than DDPG.

VI. Related Work

A. NFV Optimization

Since the proposal of NFV, the networking researchers have conducted various studies to optimize the network with different goals such as cost minimization [3], performance improvement [1], [2], utility maximization [6]. Different aspects related with NFV, e.g., VNF deployment [2], VNF resource scaling [5], flow scheduling [23], are investigated independently or jointly. For example, Sang et al. [3] study how to deploy and allocate the VNFs to minimize the total number of VNF instances. Luizelli et al. [4] present an SFC deployment algorithm that minimizes the virtual switching cost. For performance improvement, Li et al. [1] propose NFV-RT to maximize the achievable service capacity of each SFC, under of constraints of estimated end-to-end delay. Guo et al. [2] discuss the joint placement and routing of network function chains in data centers for performance optimization. For utility maximization, Kuo et al. [5] investigate the VNF placement and path selection such that the network resources can be better utilized. Later, Kuo et al. [6] further take the link capacity into the SFC embedding problem to saturate the usage of both computation and communication resources.

The above studies are all based on one-shot offline optimization. Recently, some studies have focused on how to optimize the NFV networks at run-time to adapt to network dynamics. For example, Zhang et al. [15] adopt online learning to predict the traffic demands and proactively adjust the VNF deployment to minimize the cost. Xu et al. [24] design an online admission control algorithm to maximize the network throughput for NFV-enabled multicasting. These online algorithms are still model-based with some assumptions and none of them takes the end-to-end delay into consideration.

B. Reinforcement Learning in Network Control

To many network control problems, besides model-based optimization, an alternative promising solution is to apply reinforcement learning. Actually, reinforcement learning has been successfully applied in a variety of complex control problems, in both wired networks and wireless networks. For example, in wired networks, Xu et al. [9] propose a DRL-based control framework for network traffic flow balancing among a set of pre-known paths. Stampa et al. [10] apply DRL to adapt to the traffic demands and optimize routing for network delay minimization in SDN networks. Chen et al. [12] develop a two-level DRL to deal distinguishably with the short flows and long flows for traffic optimization in datacenters. Mao et al. [11] build a resource management and job scheduling system based on reinforcement learning to minimize the job completion time. In wireless networks, Ortiz et al. [25] find that it is possible to apply reinforcement learning to tune the transmission power to maximize the two-hop communication throughput on nodes with energy-harvesting capability. Yan et al. [26] design a multi-agent reinforcement learning based radio access technology selection method to maximize the network throughput.

The above successful examples show that reinforcement learning is promising in addressing complex network control problems. Hence, we apply reinforcement learning for VNF orchestration and flow scheduling. Meanwhile, the reinforcement learning algorithms shall be customized according to the problem characteristics, as noted in [9], [12] and this work.

VII. Conclusion

In this paper, we investigated how to orchestrate the VNFs (i.e. VNF activation and deactivation) and schedule the flows to maximize the overall network utility with the consideration of end-to-end delay and various cost. It is hard to use traditional model-based algorithm to address such NUM problem formulated as a non-convex optimization form and proved as NP-hard. Instead, we resort to the newly proposed DRL technique, i.e. DDPG, to design a model-free solution. Although DDPG is applicable, we notice that it is too general to fit our problem and therefore we further customize it by redesigning the exploration method and proposing dual replay buffer for distinguishable sample reservation. In particular, we exclude the end-to-end delay to simplify our problem formulation, whose solution is then applied for the dual buffer update. The well trained control agent then can intelligently orchestrate the VNFs and schedule the flows for maximal network utility at run-time, with respect to the real-time traffic demands. Extensive experiments verify the correctness of our design and the efficiency of our algorithm by the fact that it outperforms both model-based method and non-customized DDPG based algorithm.

Acknowledgment

This work is supported by National Key Research and Development Program of China under grant No. 2018YFB1003500 and National Natural Science Foundation of China under grant No. 61602199, 61772480, and 61872310. Deze Zeng is the corresponding author (deze@cug.edu.cn).

References

[1] Y. Li, L. T. X. Phan, and B. T. Loo, "Network Functions Virtualization with Soft Real-Time Guarantees," in *Proc. of IEEE INFOCOM*, 2016, pp. 1–9.

[2] L. Guo, J. Z. F. Pang, and A. Walid, "Joint Placement and Routing of Network Function Chains in Data Centers," in *Proc. of IEEE INFOCOM*, 2018, pp. 1–9.

[3] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye, "Provably Efficient Algorithms for Joint Placement and Allocation of Virtual Network Functions," in *Proc. of IEEE INFOCOM*, 2017, pp. 1–9.

[4] M. Luizelli, D. Raz, and Y. Saar, "Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching," in *Proc. of IEEE INFOCOM*, 2018, pp. 1–9.

[5] T. W. Kuo, B. H. Liou, K. C. J. Lin, and M. J. Tsai, "Deploying Chains of Virtual Network Functions: On the Relation Between Link and Server Usage," in *Proc. of IEEE INFOCOM*, 2016, pp. 1–9.

[6] J. J. Kuo, S. H. Shen, H. Y. Kang, D. N. Yang, M. J. Tsai, and W. T. Chen, "Service Chain Embedding with Maximum Flow in Software Defined Network and Application to the Next-Generation Cellular Network Architecture," in *Proc. of IEEE INFOCOM*, 2017, pp. 1–9.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[8] W. Li, F. Zhou, K. R. Chowdhury, and W. M. Meleis, "QTCP: Adaptive Congestion Control with Reinforcement Learning," *IEEE Transactions on Network Science and Engineering*, vol. pp, no. 99, pp. 1–1, 2018.

[9] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-Driven Networking: A Deep Reinforcement Learning based Approach," in *Proc. of IEEE INFOCOM*, 2018, pp. 1–9.

[10] G. Stampa, M. Arias, D. Snchez-Charles, V. Munts-Mulero, and A. Cabellos, "A Deep-Reinforcement Learning Approach for Software-Defined Networking Routing Optimization," in *Proc. of ACM CoNEXT*, 2017, pp. 1–3.

[11] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource Management with Deep Reinforcement Learning," in *Proc. of ACM HotNets*, 2016, pp. 50–56.

[12] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: Scaling Deep Reinforcement Learning to Enable Datacenter-Scale Automatic Traffic Optimization," in *Proc. of ACM SIGCOMM*, 2018, pp. 1–15.

[13] "Bro IDS Configuration." [Online]. Available: https://www.bro.org/sphinx-git/cluster/index.html

[14] Y. Jia, C. Wu, Z. Li, F. Le, and A. Liu, "Online Scaling of NFV Service Chains Across Geo-Distributed Datacenters," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 699–710, 2018.

[15] Z. Zhang, Z. Li, C. Wu, and C. Huang, "A Scalable and Distributed Approach for NFV Service Chain Cost Minimization," in *Proc. of IEEE ICDCS*, 2017, pp. 2151–2156.

[16] "Amazon EC2 Pricing." [Online]. Available: http://aws.amazon.com/ec2/pricing/

[17] S. Shenker, "Fundamental Design Issues for the Future Internet," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 7, pp. 1176–1188, 1995.

[18] R. Krishnaswamy and M. Sviridenko, "Inapproximability of the Multi-level Uncapacitated Facility Location Problem," in *Proc. of SODA*, 2012, pp. 718–734.

[19] X. Zhang, C. Wu, Z. Li, and F. C. M. Lau, "Proactive VNF Provisioning with Multi-Timescale Cloud Resources: Fusing Online Learning and Online Optimization," in *Proc. of IEEE INFOCOM*, 2017, pp. 1–9.

[20] "Cogent's Network." [Online]. Available: http://cogentco.com/en/network/network-map

[21] "ARPA Network." [Online]. Available: https://en.wikipedia.org/wiki/ARPANET

[22] "Brain Flow Trace." [Online]. Available: http://www.zib.de/itds/brain

[23] H. Feng, J. Llorca, A. M. Tulino, D. Raz, and A. F. Molisch, "Approximation Algorithms for the NFV Service Distribution Problem," in *Proc. of IEEE INFOCOM*, 2017, pp. 1–9.

[24] Z. Xu, W. Liang, M. Huang, M. Jia, S. Guo, and A. Galis, "Approximation and Online Algorithms for NFV-Enabled Multicasting in SDNs," in *Proc. of IEEE ICDCS*, 2017, pp. 625–634.

[25] A. Ortiz, H. Al-Shatri, X. Li, T. Weber, and A. Klein, "Reinforcement Learning for Energy Harvesting Decode-and-Forward Two-Hop Communications," *IEEE Transactions on Green Communications and Networking*, vol. 1, no. 3, pp. 309–319, 2017.

[26] M. Yan, G. Feng, J. Zhou, and S. Qin, "Smart Multi-RAT Access Based on Multiagent Reinforcement Learning," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 5, pp. 4539–4551, 2018.