Abstract—Since Network Function Virtualization (NFV) decouples network functions (NFs) from the underlying dedicated hardware and realizes them in the form of software called Virtual Network Functions (VNFs), they are enabled to run in any resource-sufficient virtual machines (VMs) and offer diverse network services by service function chains (SFCs). Given the complexity and unpredictability of the network state, we propose a deep reinforcement learning (DRL) based online SFC placement method named DDQP (Double Deep Q-networks Placement). Meanwhile, VNFs are vulnerable to various faults such as software failures. Thus, we backup standby instances to enhance the fault tolerance of our model, and DDQP automatically deploys both active and standby instances in real-time. Specifically, we use DNN (Deep Neural Networks) to deal with large continuous network state space. In the case of stateful VNFs, we offer constant generated state updates from active instances to standby instances to guarantee seamless redirection after failures. With the goal of balancing the waste of resources and ensuring service reliability, we introduce five progressive schemes of resource reservations to meet different customer needs. Our experimental results demonstrate that DDQP responds rapidly to arriving requests and reaches near-optimal performance. Specifically, DDQP outweighs the state-of-the-art method by 16.30% higher acceptance ratio with 82x speedup on average.

Index Terms—Deep Reinforcement Learning, Service Function Chain, Network Function Virtualization, Fault Tolerance

I. INTRODUCTION

Currently, given that networks are filled with a massive and ever-growing variety of NFs which coupled with proprietary devices, the difficulty of network management and service provision raises rapidly. NFV changes such a situation by decoupling NFs from the underlying dedicated hardware and realizing them in the form of VNFs [1]–[3].

Conventional hardware NFs have fixed physical locations; on the contrary, VNFs can be placed in any resource-sufficient virtual machines (VMs) [1]. And usually, a sequence of NFs has to be processed in a pre-defined order which is known as service function chain (SFC) [4], [5]. Thus, a critical problem appears, that is, how to determine the positions for deploying SFCs so that the service requirement can be satisfied. Such a problem is proved to be NP-hard in general [6].

When deploying SFCs, the reliability of the model is essential. Since VNFs are software running in data centers (DCs), they are vulnerable to various problems such as connectivity errors and software faults [7]. Adopting an additional instance of each SFC instance as a backup helps to achieve the reliability of the model [8], [9]. However, reserving no resources for the standby instance in advance may fail in starting it; in the opposition, DC resources will be wasted for not all SFCs always encounter problems simultaneously. Therefore, balanced fault-tolerant deployment schemes are required, which would relax the rigid requirement of an entire dedicated copy per instance and work well in most cases.

However, the network state space shows its complexity for covering a great deal of information including the DCs, link edges and incoming requests. Specifically, DCs and edges are always together with their corresponding information such as computing resources. The incoming requests also host a body of characteristics. Furthermore, given the fault-tolerant placement, deploying standby instances makes this continuous decision problem even more complex. Nevertheless, network state typically exhibits real-time unpredictable changes due to stochastically arriving requests [10], [11], thus an appropriate online model is needed to acquire the dynamic network state transitions. However, general methods own great complexity and cost. Thus, we are requested to offer an online method to handle the huge and dynamic network state space.

To conclude, we want to reach three major targets, which are: (1) showing high performance brought by placement, (2) handling complex and large network state space in real-time, and (3) offering reliable deployment schemes flexibly.

Differing from existing works, we reach these three goals at once. We propose a deep reinforcement learning (DRL) based online method to flexibly get the dynamic network state transitions and jointly optimize SFC active and backup placement, which solves the first and second problems. For the third task, we propose five progressive schemes of resource reservation to meet different customer needs and balance the tremendous waste of duplicating all the entire SFC instances and the unreliability of reserving no resources. Also, we show our deployment design example in Fig. 1. In particular, in the case of malfunctions, we transfer the states generated by stateful VNFs during traffic processing to standby instances to guarantee seamless request redirection.

In summary, we study the online fault-tolerant SFC place-
The contributions of this paper are as follows:

- We apply the model of DDQN to deploy both active and standby SFC instances concurrently and in real-time. Our approach named DDQP provides a pretty solution to handle dynamic network variations and various service requests. To our best knowledge, we are the first to study the online fault-tolerant SFC placement problem.

- To provide a fault-tolerant deployment solution, we propose five progressive schemes of resource reservations to satisfy customer needs and then compare their performance in multiple dimensions.

- Experiments show that our proposal not only surpasses the state-of-the-art greedy method by 16.30% higher acceptance ratio but also gains 82x speedup on average.

The rest of this paper is organized as follows. Section II discusses the related work. Section III presents the problem formulation and MDP model. In Section IV, we formally propose our algorithm design. Section V presents the evaluations and at last, Section VI concludes the paper.

II. RELATED WORK

Online SFC Placement. There has been a large body of work that studies the VNF placement problem tending to reach different objectives in varying scenarios [13]. Usually, mathematical programming methods such as Integer Linear Programming (ILP) [14] and Mixed ILP (MILP) [15] are used to tackle it. For instance, Bari et al. [6] formulated the problem as an ILP model, in which the goals of minimizing the OPeration EXpenses (OPEX) and maximizing the utilization are considered. However, in the large scale networks, the execution time of mathematical solutions grows exponentially with the network size, resulting in poor performances [1]. Besides, they do not consider that VNFs are always chained as SFCs, or even that requests come in real-time, which causes network traffic fluctuations [11]. Xiao et al. [11] tackled it by introducing a Markov decision process (MDP) model to acquire the dynamic network state transitions. Although this approach solved the problem of unpredictable network state space and offered an online approach, it takes no care of the traffic routing and service reliability.

Fault-tolerant SFC Placement. Nevertheless, the reliability of the service is also crucial. NFs can suffer from temporary unavailability due to misconfiguration or software and hardware failure [16]. Through redundancy, it is possible to offer schemes guaranteeing high levels of reliability. In the case of malfunctions, states generated by stateful VNFs during traffic processing need to be transferred to standby instances for guaranteeing seamless request redirection. However, Carpio et al. [17] studied the joint active and standby stateless VNF placement problem without considering VNF state transfers. Yang et al. [18] jointly optimized the objectives, but several challenges remain. Their work could not ensure completely reliable service and would waste abundant resources. Besides, its incapacity to deal with the real-time requests makes its performance poor. Thus, an effective solution to schedule both active and standby instances concurrently is urgently needed.

DRL for NFV. Previous solutions of optimizing SFC placement such as [6] have the problem of overspending in dealing with the complex network states. Another important paradigm is to leverage machine learning mechanisms. For example, the authors in [11], [19] have proposed DRL-based approaches to handle the large network state space in SFC placement problem. However, existing efforts either neglected fault tolerance or addressed the fault-tolerance problem in an offline fashion. Our model solves the online fault-tolerant SFC placement for the first time, which automatically learns and acts to arriving requests in real-time. In particular, we decide the placements of both active and standby instances with different levels of resource guarantees, which can discriminate customized SFC availability.

III. MODEL AND FORMULATION

In this section, we begin with the NFV structure description and fault-tolerant SFC placement problem formulation with the objective and constraints. Then we elaborate on how to use the MDP model. The key notations are listed in Table I.

A. Problem Formulation

In an NFV network structure, not the entire DC is available for us and only part of the permissions are opened leading to finite available resources [3]. Thus we represent the network as a connected graph $G = (V, E)$, where $V$ is the set of DCs, and $E$ is the collection of edges (or links) connecting each of two DCs, $\forall e = (v_1, v_2) \in E, v_1, v_2 \in V$. Every DC has a computing resource capacity, which is denoted as $C_v$. And $W_e$ and $L_e$ represent the bandwidth and communication latency of edge $e \in E$ respectively.

Next, we use $R$ and $\hat{R}$ to denote the sets of real-time arriving requests and their standby instances respectively. Each request $r \in R$ needs to be steered through its service-related SFC and
TABLE I
Key Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G )</td>
<td>The graph ( G = (V, E) ) representing the underlying NFV network</td>
</tr>
<tr>
<td>( V )</td>
<td>The set of DCs within the network</td>
</tr>
<tr>
<td>( E )</td>
<td>The set of edges (or links) between the DCs, ( \forall e = (v_1, v_2) \in E, v_1, v_2 \in V )</td>
</tr>
<tr>
<td>( R )</td>
<td>The set of service requests</td>
</tr>
<tr>
<td>( \tilde{R} )</td>
<td>The set of standby instances</td>
</tr>
<tr>
<td>( L_e )</td>
<td>The communication latency of edge ( e \in E )</td>
</tr>
<tr>
<td>( W_e )</td>
<td>The bandwidth of edge ( e \in E )</td>
</tr>
<tr>
<td>( C_v )</td>
<td>The computing resource of DC ( v \in V )</td>
</tr>
<tr>
<td>( D_r )</td>
<td>The resource demand ( D_r = (D_{r,1}^e, D_{r,2}^e, D_{r,3}^e) ) of service request ( r \in R ) in terms of computing resource, throughput, state updating throughput and latency</td>
</tr>
<tr>
<td>( s_r )</td>
<td>Source of request ( r \in R )</td>
</tr>
<tr>
<td>( d_r )</td>
<td>Destination of request ( r \in R )</td>
</tr>
<tr>
<td>( h_r )</td>
<td>The path occupied by service request ( r \in R ) from its source ( s_r ) to destination ( d_r ) via the DC on which it is deployed</td>
</tr>
<tr>
<td>( h_r^b )</td>
<td>The state updating path occupied by service request ( r \in R ) from the DC on which it is deployed to the DC on which its backup is deployed</td>
</tr>
<tr>
<td>( x_{e,h} )</td>
<td>1 if edge ( e \in h ), 0 otherwise</td>
</tr>
<tr>
<td>( t_r )</td>
<td>The actual processing latency of request ( r \in R )</td>
</tr>
<tr>
<td>( p_r )</td>
<td>1 if request ( r \in R ) is accepted, 0 otherwise</td>
</tr>
<tr>
<td>( a_{v,r}^r )</td>
<td>1 if request ( r \in R ) is in service at time slot ( \tau ) on DC ( v ), 0 otherwise</td>
</tr>
<tr>
<td>( b_r )</td>
<td>1 if the standby instance of request ( r \in R ) starts successfully when its active instance is broken, 0 otherwise</td>
</tr>
</tbody>
</table>

has its computing resource, transition throughput, and state updating throughput demand and latency limitation, denoted by \( D_r = (D_{r,1}^e, D_{r,2}^e, D_{r,3}^e) \). And we denote the path occupied by request \( r \in R \) from its source \( s_r \) to destination \( d_r \) via the DC on which it is deployed as \( h_r \). Meanwhile, \( h_r^b \) is used to indicate the state updating path. And we use \( x_{e,h} \) to indicate whether edge \( e \in h \) is deployed. Besides, for each request \( r \in R \), we use a binary variable \( p_r \) to indicate the deployment decision of each SFC. Specifically, if request \( r \in R \) can be accepted, \( p_r = 1 \); otherwise, \( p_r = 0 \).

To handle the real-time network variations caused by stochastic arrival and departure of requests, we use the time slot \( \tau \) to denote the integral multiple of a constant period time \( \Delta \) (i.e., \( \tau = n \times \Delta, n \in \mathbb{N} \), \( \Delta \) is configurable, e.g. \( \Delta = 1 \mu s \), 1 ms or 1 s). In particular, we use binary \( a_{v,r}^r \) to indicate whether request \( r \in R \) is in service on DC \( v \) at time slot \( \tau \). Thus we use binary \( a_{v,r}^r \) to denote whether it is still in service:

\[
\forall r \in R, a_{v,r}^r = \sum_{v \in V} a_{v,r}^r \quad (1)
\]

Meanwhile, we introduce binary \( b_r \) to indicate whether the standby instance of request \( r \in R \) starts successfully when its active instance is broken. Specifically, if backup of request \( r \) can be activated, \( b_r = 1 \); otherwise, \( b_r = 0 \).

Now we present the mathematical formulation of the fault-tolerant SFC placement problem. We begin with the constraints and then the objective together with some insight.

First, NFV providers can place multiple SFCs at the same DC if it has sufficient resources [12]. Since we consider backup resource reservations, the total resources occupied by active instances and reserved for standby instances should not exceed the capacity of the current DC. As we propose five schemes for backup resource reservations, we denote it as \( C^{res} \) here and will discuss it in the following sections. Thus, we state the resource constraint on DCs in inequality (2).

\[
\forall v \in V : \sum_{r \in R \cup \tilde{R}} a_r^e D_r^e + C^{res} \leq C_v \quad (2)
\]

Second, we introduce the latency constraint. We use \( t_r \) to indicate the processing latency of request \( r \in R \), while the total latency should be the sum of processing latency and the communication latency on links. If a request \( r \in R \) is accepted, its response latency can not exceed its limitation \( D_r^l \). Thus, we state the end-to-end latency constraint as follows:

\[
\forall r \in R \cup \tilde{R} : t_r + a_r^e \sum_{e \in h_r} L_e \leq D_r^l \quad (3)
\]

Third, we explore the bandwidth resource constraint. Since the throughput hosted by link contains the bandwidth occupied by SFC transition and used for the state updating, the total occupation demand of all requests passing through link edge \( e = (v_1, v_2) \in E \) cannot exceed its limits. Thus we have:

\[
\forall e \in E : \sum_{r \in R \cup \tilde{R}} a_r^e x_{e,h_r} D_r^e + \sum_{r \in R} a_r^e x_{e,h_r} D_r^b \leq W_e \quad (4)
\]

With all these constraints, we have the objective.

**Objective:** Maximize the number of accepted requests, which can be defined as maximizing the acceptance ratio and expressed as:

\[
\max_{s.t.(2), (3), (4)} \sum_{r \in R} p_r \quad (5)
\]

**Insight:** Under the premise of ensuring the service reliability, this objective intuitively demonstrates excellent performance of accepting a large number of requests. With a higher acceptance ratio, we gain larger throughput of requests and better service which shows great QoS.

B. MDP Model

Since the online fault-tolerant SFC placement is a continuous decision problem, in order to address it and handle the network state transitions, we formally present the MDP model, which is typically defined as \( < S, A, P, R, \gamma > \), where S and A are the sets of continuous states and discrete actions respectively. \( P : S \times A \times S \) is the transition probability distribution, \( R : S \times A \) is the reward function, and \( \gamma \in [0, 1] \) is a discount factor for future rewards.

**State Definition:** For each state \( s_t \in S \), we define it as a vector \( (\hat{V}_t, \hat{E}_t, F_t) \). We first label each DC in the network with an integral index \( i = 1, 2, \ldots, |V| \) and each edge with \( k = 1, 2, \ldots, |E| \). \( \hat{V}_t = (\hat{V}_t^1, \hat{V}_t^2, \ldots, \hat{V}_t^{|V|}) \) represents the current state of each DC. \( \forall j \in [1, |V|], \hat{V}_t^j = (\hat{V}_{c}^j, \hat{V}_{act}^j, \hat{V}_{res}^j) \) indicates the initial resources, resources occupied by active instances and resources reserved for standby instances of DC \( v_j \) respectively. \( \hat{E}_t = (\hat{E}_t^1, \hat{E}_t^2, \ldots, \hat{E}_t^{|E|}) \) represents the current
The current SFC being processed, where \( T \) resources demand, \( E \) reserved for standby instances of edge indicates the latency, remaining bandwidth and bandwidth state of each edge. \( \forall k \in [1, |E|], \hat{\mathcal{E}}_k = (\hat{E}^{i,k}_t, \hat{E}^{r,k}_t, \hat{E}^{i,k}_r) \) indicates the latency, remaining bandwidth and bandwidth reserved for standby instances of edge \( \epsilon_k \) severally. \( F_i = (C_{r_i}, T_{r_i}, L_{r_i}, T^*_r, L^*_p, S_{r_i}, D_{r_i}) \) reveals the characteristics of the current SFC being processed, where \( C_{r_i} \) is the computing resources demand, \( T_{r_i} \) is the throughput demand, \( L_{r_i} \) is the transition latency limitation, \( T^*_r \) is the state-updating throughput demand, \( L^*_p \) is the processing latency and \( S_{r_i} \) and \( D_{r_i} \) are the source and destination respectively. We design such state space to perceive the current state of the environment.

**Action Definition.** We denote action \( a \in \mathcal{A} \) as an ordered pair \((i_a, i_b), \) where \( i_a, i_b \in \{1, 2, ..., |V| \}. \) In particular, we place the active instance on the DC \( v_{i_a} \) while standby instance on the DC \( v_{i_b}. \)

**Reward Function.** Since we want to maximize the number of accepted requests, we define the reward function as whether the SFC is successfully deployed. We adopt a simple but effective way of using the active number 1 to reward triumphant actions while the negative number -1 to punish failing actions. The mathematical formulation of the reward function is given considering two different cases as the time slot moves on:

\[
\begin{align*}
    r(s, a) &= \begin{cases} 
        1, & \text{if request } r_i \text{ is accepted,} \\
        -1, & \text{if request } r_i \text{ is rejected.}
    \end{cases}
\end{align*}
\]

**State Transition.** As shown in Fig. 2, the MDP state transition is defined as \( (s_t, a_t, r_t, s_{t+1}) \), where \( s_t \) is the current network state and \( a_t \) is the action taken for dealing with the deployment of SFCs. And then reward \( r_t \) is fed back, and \( s_{t+1} \) is the new network state. The MDP state transition will also be expounded in IV-B.

**IV. Algorithm Design**

To optimize fault-tolerant SFC placement, we begin with the architecture introduction together with its neural network design. Then we have how this adaptive online DRL approach works while deploying. After that, we introduce the DDQN based training procedure of our design. Finally, we present five schemes to adapt to different scenarios and user needs.

**A. Architecture Design**

As shown in Fig. 2, there are two elements in the structure, agent and environment. The DDQP agent builds a replay buffer to remove data dependencies. During training, samples are randomly drawn from the experience pool to replace the current samples. Then, the learner trains on them and updates the sample network. And at every \( C \) step, the parameters of the target network are updated through the sample network. The NFV network environment includes the current information of topology and being processed SFC. Depending on the current state, the agent chooses an action. Then the environment feeds a reward back and moves to the next state.

With MDP, we can automatically and continually characterize the network traffic variations. Next, we need to find an effective and efficient policy that can automatically take appropriate actions in each state to achieve a positive reward. Thus, we propose our design to online deploy SFCs with service availability guarantee named DDQP.

Generally, DRL approach can be classified into two categories, one is the value-based approach (e.g., deep Q-networks (DQN) [20]) and the other is the policy-based approach (e.g., policy gradient [21]). In the SFC deployment problem, action space is discrete and we hope algorithm updates every action and learns rapidly. Under these circumstances, the value-based approach Double DQN (DDQN) [22] is adopted. The general goal is to learn a policy \( \pi(a|s) \) to maximize expected return. Q-function \( Q^\pi(s, a) \) is used to evaluate it. And calculating \( Q^\pi(s, a) \) is realized by value function approximation using Q-network \( Q_{\phi}(s, a) \), which is always a function with parameter
and the output layer is the neurons to process the hidden information. The input layer, an output layer and 3 hidden layers each with 512 neurons in the input layer should be \(3(|V| + |E|) + 7\). Activation function \textit{Leaky ReLU} [23] is also used between layers. Since we adopt a negative return in the reward, \textit{ReLU} [24] is suitable for it reaches the activation value just with a threshold other than complex calculations. However, it is fragile while training. Thus, \textit{Leaky ReLU} is proposed to solve this dying-ReLU problem and adopted in our model.

### B. Deployment Approach

We introduce how the DDQP works as the time slot \(\tau\) moves on in Fig. 4. As we can see, there are two cases as the time slot moves on: (1) **Inter time slot**, which indicates the moment between every two time slots. In this case, no action is taken and we just update the network state. Now we elaborate each case in turn.

**Intra time slot.** As shown in Fig. 4, two requests arrive (i.e., \(SFC_1\) and \(SFC_2\)) at time slot \(\tau\). The DDQP agent firstly rescans all the DCs, starts standby instances whose active instance has damaged and removes timeout requests, then refreshes the network state. Then it receives these two arriving requests successively based on the arriving time. At state \(s_1\), the DDQP agent observes the network state and calculates the Q-network \(Q_\phi(s, a)\) according to every \(a \in A\). Then we assume the action \((2, 7)\) owns the biggest value of \(Q_\phi(s, a)\) and is returned to the agent. Thus the action \((2, 7)\) is applied to place \(SFC_1\) on the selected candidate DCs, specifically the former indicates the active instance DC index while the latter shows the standby index. As a result, \(SFC_1\) is successfully deployed at state \(s_1\) with \(r_1 = 1\) as a reward for the action. At state \(s_2\), since there is no DC having sufficient resources to place \(SFC_2\), or the bandwidth or latency constraints can not be satisfied, the returned action \((3, 6)\) can not work and thus \(SFC_2\) is rejected. Then, reward -1 is fed back to the agent as a punishment and then the model moves to the new state \(s_3\).

**Inter time slot.** As shown in Fig. 4, at the time slot \(\tau\), the DDQP agent has rejected \(SFC_2\). Between the time slot \(\tau\) and \(\tau + 1\) is an inter time slot. The DDQP agent only executes the following procedures: rescanning all the DCs, starting standby instances whose active instance has damaged and removing timeout requests, then updating the network state.

### C. DDQN Based Training Procedure

We adopt DDQN to directly optimize the quality of SFC deployment by calculating the Q-network \(Q_\phi(s, a)\) of the candidate actions. With DDQN, our target is to learn the parameter \(\phi\) to let Q-network \(Q_\phi(s, a)\) approach Q-function \(Q^\pi(s, a)\). An episode of training is defined as a time slot and thus consists of a sequence of MDP state transitions. During each episode, all the state transitions are successively stored in a buffer and used for training until this episode ends. The target function is defined as:

\[
Y = r + \gamma Q(s', \arg \max_a Q(s', a; \theta); \theta^-)
\]

where \(r\) and \(s'\) are the reward fed back and the state of the next time, \(\theta\) and \(\theta^-\) are weights of Q-function \(Q\) and target Q-function \(\hat{Q}\) respectively.

The DDQN based training procedure is listed in Algorithm 1. In each episode, we initialize the NFV environment. And in each MDP state transition the agent processes one SFC and gains a reward \(r_t\). Specifically, in order to balance the exploitation of policy and the exploration of the environment, we adopt the \(\epsilon\)-greedy method when choosing actions. Also, we benefit from the Double method to train function \(Q\) and \(\hat{Q}\) with different parameters for it greatly alleviates the problem of overestimation and shows great performance.
Our Proposed Backup Schemes

In order to meet the different needs of customers and adapt to different scenarios, we design five progressive schemes range from Level 0 to Level 4. We picture them in Fig. 5 and now introduce these schemes in detail.

**Algorithm 1 DDQN Based Training Procedure**

1. **Begin:** Initialize replay memory $D$ to capacity $N$
2. Initialize Q-function $Q$ with random weights $\theta$
3. Initialize target Q-function $\hat{Q}$ with weights $\theta^- = \theta$
4. for episode $\leftarrow 1, M$ do
5. Initialize the NFV environment and let state $s \leftarrow s_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6. for $t \leftarrow 1, T$ do
7. Select an action $a_t$ randomly with the probability $\epsilon$, otherwise select action $a_t = \max_a Q(\phi(s_t), a; \theta)$
8. Execute action $a_t$ and observe reward $r_t$
9. Transfer the state to $s_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
10. Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
11. Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
12. if episode terminates at step $j + 1$ then
13. Set $y_j = r_j$
14. else
15. Set $a' = \arg\max_a Q(\phi_{j+1}, a, \theta)$
16. Set $y_j = r_j + \gamma Q(\phi_{j+1}, a'; \theta^-)$
17. end if
18. Perform a gradient descent step on $(y_j - Q(\phi_j, a; \theta))^2$ with respect to the network parameters $\theta$
19. Every $C$ steps reset $\hat{Q} = Q$
20. end for
21. end for

**D. Our Proposed Backup Schemes**

Let's start with our definition of SFC states together with its state transition. As shown in Fig. 6, we divide the states into six categories, namely, **undeployed**, **active**, **failed**, **backup**, **broken** and **expired**. Specifically, once a request comes, we initialize the SFC as **undeployed**. Through our decision, we use **active** and **failed** to express whether the deployment succeeds and the active instance starts. When the active instance fails, we have **backup** if the standby instance starts up successfully and runs well. However, if both active and standby instances cannot run when the deployment is successful, that is to say, there is no standby instance or even the standby instance is damaged, we set this state as **broken**. If it meets the condition of time-expired at any time, we use **expired** to indicate it.

Now we go through each scheme in detail. We refer to the original non-backup scheme as **Level 0** and the backup scheme without considering resource reservation as **Level 1**. As mentioned earlier, the system is fragile and prone to malfunction in the **Level 0** situation. The **Level 1** method is also prone to failure when there are many active instances deployed on the backup DC and a standby instance wants...
to start up. The Level 2 method is designed by considering whether DC has sufficient resources at present. The difference in effect between the Level 1 and Level 2 methods is minor because the Level 2 method will also fail in cases of a new active instance is deployed on the backup DC and one standby instance wants to start up. Hence, we have the idea of Level 3. As for DCs, we reserve the maximum demand of resources among all the standby instances deployed on it. And we do the same reservation towards every link. However, it may still fail if there are more than one backups that need to be activated. Thus, Level 4 is proposed and it has the highest level of fault tolerance since it reserves all the needed resources, including DCs and link bandwidth in advance. Nevertheless, its reliability is conditional on the sacrifice of half of the resources. Thus we can say, the progressive schemes are a tradeoff between the level of fault tolerance and the resource efficiency. An appropriate scheme can be chosen according to customer needs (e.g. SLA). We will give the evaluation results in the next section.

V. EVALUATION AND ANALYSIS

A. Simulation Setup

Network topology: Our evaluation experiments are based on five DC topologies, which are Abilene, ANS, AboveNet, Integra and BICS [25]. These topologies are located in the United States, Europe and Japan, and several of them also span these regions. The number of DCs in these topologies ranges from 9 to 33. Each DC is connected to one or several other DCs. We assume that in each of the above topologies, the computing resources of DCs used to place VNFs in each one range from 60,000 unit to 80,000 unit, the bandwidth of each link ranges from 400Mbps to 800Mbps, and the delay of each link ranges from 2ms to 5ms.

SFC of requests: According to [13], different VNFs are simulated for composing SFCs, including 5 typical VNFs (i.e., Firewall, proxy, NAT, DPI, and Load Balancer [26]). Further, the computing resource demand of a SFC is the sum of the computing demands of its contained VNFs (the number of contained VNFs is randomly selected between 1 and 7) and ranges from 3750 unit to 7500 unit. The processing delay of a packet for each SFC is randomly drawn from 0.863ms to 1.725ms [26]. Each request r is generated by randomly selecting its source sr and destination dr from G. Each request has a delay requirement ranging from 10ms to 30ms [27]. And we suppose the TTL of each request ranges from 5s to 10s. Finally, we simulate 3000 requests in 300 time slots.

Baseline and schemes compared: We compared DDQP with Random Greedy (RG) algorithm, Best-fit Greedy (BFG) algorithm [18] and Near Optimal Sorting Greedy (NOSD) algorithm. As we are the first to study the online fault-tolerant placement, we changed these three offline algorithms to online algorithms for evaluation. The RG algorithm is a time-efficient algorithm, but its performance is poor. It randomly chooses resource-sufficient DC to deploy active instances. The BFG algorithm is a fault-tolerant placement algorithm, which is a best-fit algorithm. The intuition behind best-fit is simple: by returning a DC whose remaining resources are closest to what the user asks, it tries to reduce wasted resources. BFG searches globally for a DC with least remaining resources that can meet the needs to place the active instance, and by choosing a shortest updating link to a closer DC to place the standby instance. This design reduces the waste of resources and the overhead of updating, and thus has better performance than the RG algorithm. NOSD is an improved algorithm based on the BFG algorithm. In the selection of standby instances, it takes the guarantee of the post-failure routing into consideration, which makes the selection of standby instances more reasonable. Since there is no optimal solution to this online decision problem, we hold that NOSD has a near-optimal performance for it always chooses the guaranteed best-fit instances, but its time overhead is also the largest. We compared the DDQP algorithm with these three algorithms, and the results of the comparison will be analyzed later.

Simulation platform: We use a Python-based simulation framework, which mainly contains the Pytorch library to build neural networks. All experiments are based on a workstation with 32GB RAM and an Intel (R) Core (TM) i7-5930K CPU, which has 6 cores and 12 threads.

B. Performance Evaluation

Acceptance ratio: We compared the acceptance ratios of the DDQP, RG, BFG and NOSD algorithms in five different topologies and adopted the RG algorithm as the baseline. From the comparison results in Fig. 7, we can see that DDQP has improved the acceptance ratio by 62.14% on average compared to RG and 16.30% to BFG, and finally, from the comparison between the DDQP and the NOSD algorithm, it can be seen that DDQP has a near-optimal performance.

Throughput: We also compared the throughput of them. From the comparison results in Fig. 8, we can find that compared with the RG algorithm, DDQP improves the throughput by 61.35% on average, and 16.04% compared with the BFG algorithm. Finally, it can be seen from the comparison between DDQP and NOSD that DDQP has a near-optimal performance.

Service availability: We introduce the sum of the actual service lifetime of each SFC to indicate the service availability, because backup can increase the lifetime of the instance which is likely to fail. Specifically, both the number of accepted requests and the lifetime of every single request affect this parameter. And the comparison results of these four algorithms are shown in Fig. 9. Thus, we can conclude that DDQP has improved the service availability by 68.23% on average compared to the RG algorithm and 16.30% to the BFG algorithm, and finally, from the comparison between the DDQP and NOSD algorithms, DDQP shows a near-optimal service availability.

Training efficiency: To demonstrate our training efficiency, we compared the DDQP with the RG, BFG and NOSD algorithms on the AboveNet topology. All our comparison results were obtained under the Level 3 scheme. During training, we dynamically change the learning rate to speed up the learning process. It can be seen from the comparison results in Fig. 13 that when the DDQP algorithm is trained
to the 19th episode (each episode takes about one minute), its average accepted number exceeds the RG algorithm; when it is trained to the 121th episode, its performance exceeds the BFG algorithm; and when it is trained to the 676th episode, the average accepted number of it equals to the NOSD algorithm. In the end, DDQP started to converge when it was trained to about 1200th episode, and we must note that this training time cost is negligible compared to the time between every two changes in the network topology. Generally speaking, DDQP converges fast during training. As the number of DCs in the NFV network and the complexity of the network topology increase, it takes more time to converge.

Online runtime cost: We compared the online runtime costs of the DDQP, RG, BFG and NOSD algorithms in five different DC topologies. Although DDQP spends some time to train the neural networks, after the training is completed and deployed, it only needs to use the NN for reasoning. And without having to consider the complex calculations of DCs and links in NFV networks to make decisions, the DDQP algorithm has extremely low time overhead. From the experimental results in Fig. 14, we can see that DDQP saves about 88.78% of running time overhead compared to the RG algorithm, 98.81% compared to the BFG algorithm and 98.86% compared to the NOSD algorithm. That is to say, Our speed is 7.91 times faster than the RG algorithm, 82.71 times faster than the BFG algorithm, and 86.57 times faster than the NOSD algorithm. Thus, we have achieved a pretty performance in running speed.

Evaluation on backup schemes: We use the Level 0 scheme which has no backup as the baseline. From the comparison results of multiple schemes in Fig. 10, 11, it can be seen that, because the Level 0 scheme saves the reserved resources occupied by the standby instances, it has a higher acceptance ratio and throughput. As seen in Fig. 12, we introduce fault tolerance to represent the reliability of the service by calculating the startup success ratio of standby instances.
and these five different levels also progressively increase the requirements for fault tolerance. Taking these considerations into account, we assess the comprehensive indicator of service availability in Fig. 15. The service availability of Level 0 is relatively low because no backup leads to a shorter lifetime for each SFC instance. Similarly, Level 4 reserves more resources for backup and results in fewer requests being accepted, which also leads to a reduction in service availability. As shown in Fig. 15, we think service availability reflects the trade-off between performance (acceptance ratio, throughput) and reliability (fault tolerance) for different schemes, which reflects the quality of a scheme when the user does not explicitly specify the requirements.

Actually, we want users to choose the appropriate scheme based on their demands. For example, if the user has very strict requirements for fault-tolerance, we will recommend the Level 4 scheme. Otherwise, according to Fig. 10, 11 and 12, we will recommend using Level 1, Level 2 or Level 3 scheme to get a better balance between performance and fault tolerance. In Fig. 12, we can see that these three schemes have significantly improved the fault-tolerance compared to the Level 0 scheme. If users think their NFV placement environment is not necessary for backup, then they can choose the Level 0 scheme to acquire a higher acceptance ratio.

VI. CONCLUSIONS

In this paper, we present a deep reinforcement learning solution named DDQP for the optimization of the SFC deployment problem. Specifically, we use different weights to train action-value function and target action-value function with the purpose of solving the overestimation problem. Our design not only gets the actual real-time network traffic characteristics and responses fast in data centers, but also gains great improvement on acceptance ratio and service availability of requests. To minimize the resource wasted by reservation, we also propose five progressive schemes to match customer needs. Through our experiments, we show that our methods scale well in diverse scenarios and adapt well to large continuous state space and action space. To conclude, our proposal outweighs the state-of-the-art method by 16.30% higher acceptance ratio and 82x speedup on average.

ACKNOWLEDGMENTS

The work was supported by the National Natural Science Foundation of China under Grant No. 61972101. Jin Zhao is the corresponding author.

REFERENCES