Towards Chain-Aware Scaling Detection in NFV with Reinforcement Learning

Lin He, Lishan Li, Ying Liu,
Institute for Network Sciences and Cyberspace, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS) | 978-1-6654-1494-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/IWQOS52092.2021.9521362

***Abstract*—Elastic scaling enables dynamic and efficient resource provisioning in Network Function Virtualization (NFV) to serve fluctuating network traffic. Scaling detection determines the appropriate time when a virtual network function (VNF) needs to be scaled, and its precision and agility profoundly affect system performance. Previous heuristics define fixed control rules based on a simplified or inaccurate understanding of deployment environments and workloads. Therefore, they fail to achieve optimal performance across a broad set of network conditions.**

In this paper, we propose a chain-aware scaling detection mechanism, namely CASD, which learns policies directly from experience using reinforcement learning (RL) techniques. Furthermore, CASD incorporates chain information into control policies to efficiently plan the scaling sequence of VNFs within a service function chain. This paper makes the following two key technical contributions. Firstly, we develop chain-aware representations, which embed global chains of arbitrary sizes and shapes into a set of embedding vectors based on graph embedding techniques. Secondly, we design an RL-based neural network model to make scaling decisions based on chain-aware representations. We implement a prototype of CASD, and its evaluation results demonstrate that CASD reduces the overall system cost and improves system performance over other baseline algorithms across different workloads and chains.

***Index Terms*—scaling detection, service function chain, reinforcement learning, NFV**

I. Introduction

Network Function Virtualization (NFV) was recently introduced to replace proprietary network function hardware devices with software-based virtual network function (VNF) applications [1], [2], which offers the potential to enhance service delivery flexibility and reduce overall operational expenses. NFV enables elastic scaling by creating and destroying VNF instances [3], [4], which can well adapt to frequent and substantial dynamics of network traffic volumes [5].

The primary goals of elastic scaling are to 1) satisfy service level agreements (SLAs) and 2) minimize VNF operating cost. Even a small increase in response latency leads to a breach of specific SLAs and a large amount of revenue loss. For example, Amazon reported that it lost 1% of sales for an increase of 100 ms in response latency [6]. However, simultaneously achieving both goals requires agile scaling detection, which determines the appropriate time when a VNF needs to be scaled. On the one hand, delayed scaling detection after overload causes performance degradation and the breach of SLAs. On the other hand, premature scaling detection before actual overload leads to the waste of allocated resources and higher operating costs.

To address the above issue, researchers have developed scaling detection algorithms based on estimated traffic rate ("rate-based") [7]–[10] or VNFs' runtime status ("status-based") [11], [12]. However, they are all designed based on a simplified or inaccurate understanding of deployment environments. Consequently, they inevitably fail to achieve optimal performance across a broad set of network environments and diverse traffic patterns. Moreover, they all focus on single VNF's features and ignore interconnection relationships among VNFs.

Recently, reinforcement learning (RL) techniques have been successfully applied to many network problems, such as adaptive bitrate algorithm in video streaming [13] and job scheduling [14]. Motivated by that, we propose a chain-aware scaling detection mechanism, namely CASD, which utilizes RL techniques to automatically learn scaling detection policies from experience without using any predefined control rules or explicit assumptions about the deployment environment. To further improve performance, we incorporate global chain information into control policies to efficiently plan the scaling sequence of VNFs. Specifically, we have the following two key innovations.

Firstly, it is common that input traffic needs to traverse multiple VNFs in a particular order, which is called a service function chain. It is observed that the arrival of huge traffic [5] often requires performing scaling out operations of multiple VNFs. However, upstream VNFs with lower throughputs may impede scaling detection of downstream VNFs with higher throughputs, as shown in Fig.1. Therefore, CASD extracts interconnection relationships among VNFs from the chain to efficiently plan the scaling sequence of them.

Secondly, CASD models scaling detection policies via neural networks, which maps VNF and global chain information to scaling actions. Compared with traditional statistical modeling methods, neural networks provide a scalable way to incorporate a wide variety of features into control policies. We utilize A3C algorithm [15], a state-of-the-art actor-critic RL algorithm, to train the model. During the training process, CASD learns policies directly from observations of the result-

The authors would like to thank all anonymous reviewers for their constructive comments. This work was supported by the National Key Research and Development Program of China (Grant No. 2018YFB1800405) and National Natural Science Foundation of China (Grant No. 61772307). Prof. Ying Liu is the corresponding author.

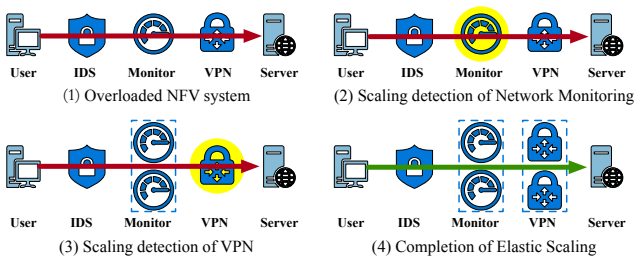


Fig. 1. A scenario requiring global chain information to make agile scaling detections. The considerable traffic demand requires scaling actions of both the network monitor and the VPN. The network monitor first becomes the system bottleneck and detects scaling requirements. During the setup phase, the output traffic rate of the network monitor is limited by its throughput. Therefore, the VPN cannot be aware of the considerable traffic demand. After the network monitor finishes its scaling events and is no longer the system bottleneck, the VPN can detect scaling events. Instance setup usually consumes several seconds/minutes. The performance of the whole system is profoundly affected by the delayed detection of downstream VNFs.

ing performance of past decisions. In this way, it can adapt to a variety of environments and workloads without human intervention.

We implement a prototype of CASD on top of OpenNetVM [16] which is a high-performance NFV platform. We compare it with multiple baseline algorithms spanning different traffic patterns and chains. The experiment results show that CASD outperforms the state of the arts in terms of overall cost and system performance.

Contributions. We make the following contributions in this paper:

- CASD is a RL-based scaling detection mechanism, which learns scaling detection policies from observations of the resulting performance of past decisions without any assumption of environments and workloads. (Section III and IV)
- CASD incorporates a global chain into control policies. We develop a scalable chain representation method, which embeds a chain of arbitrary size and shape into a set of embedding vectors based on graph embedding techniques. (Section V)
- A RL-based neural network model is proposed to encode scaling decisions as actions based on embedding vectors. (Section VI)
- We implement a prototype of CASD and compare it with multiple baseline algorithms. The evaluation results show that CASD can adapt to different traffic patterns and chains and outperforms baseline algorithms in all considered scenarios. (Section VII and Section VIII)

II. Motivation

Most existing scaling detection algorithms define fixed control rules based on estimated traffic rate or VNFs' runtime status. However, they cannot achieve accurate and agile scaling detection due to the following reasons:

- **Ignoration of Interconnection Relationships in Chains:** Commonly, input traffic is processed by multiple VNFs in a specific order based on a predefined chain. One potential scenario is that upstream VNFs with low throughput will impede scaling detection of downstream VNFs with high throughput. Current scaling detection mechanisms are designed based only on each VNF's attributes but ignore the information of the global service function chain. Interconnection relationships between VNFs can help improve the agility of scaling detection algorithms. Thus, it is necessary to incorporate chain information into control policies.

- **Inaccurate Understanding of Network Environments:** Network conditions vary significantly over time and across a variety of network environments. For example, in a time-varying network environment, traffic rate prediction is infeasible. In this situation, runtime status features, such as latency, should be prioritized. However, traditional methods utilize fixed control rules based on a simplified and inaccurate understanding of environments. Besides, they require sophisticated tuning and do not generalize to various network conditions.
- **Lack of Accurately Labeled Datasets:** Some existing studies use neural networks to solve scaling detection issue, which relies on accurately labeled datasets for model training [11], [12]. However, in an NFV system, network conditions fluctuate and can even vary significantly over time. Therefore, it is often tedious and infeasible for humans to accurately label data.

Recently, deep learning techniques have been proved useful in solving network problems. Some proposals are dedicated to solving online decision-making problems with neural networks and RL [13], [14], [17], [18]. In this paper, we try to apply neural networks and RL into scaling detection problem. The neural network provides an expressive and scalable way to incorporate both VNFs' status and global chain information into control policies. Moreover, RL can automatically learn scaling detection policies just through observation of an NFV system's performance of past decisions beyond a high-level goal. Its training process does not rely on either assumption about network environments or pre-labeled datasets. Its trained scaling detection mechanisms can automatically optimize for complicated and varying goals of different users.

III. Scaling Detection Task

In this section, we first formalize the scaling detection task in NFV. Then we explain the corresponding state space, action space, and reward function, which are further used in the RL-based neural network model. The notation we use throughout the CASD description is summarized in Table I.

A. Task Definition

Currently, most NFV architectures decouple the control plane from the data plane [19]. The controller uses a network-wide view to efficiently deploy and scale VNFs over instances to process fluctuating network traffic. In general, input traffic is processed by multiple VNFs which compose a service

function chain. Given a chain $\mathcal{C} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of VNFs and $\mathcal{E}$ is a set of directed edges between VNFs in $\mathcal{V}$.

Our designed scaling detection algorithm aims to compute the scaling action for each VNF in $\mathcal{V}$ to maximize system throughput with minimum instances, which may be one of the following three options:

1) **Scaling Out**: A new instance is launched to process the increasing traffic.
2) **No Scaling**: Its number remains unchanged.
3) **Scaling In**: An instance is removed to release vacant resources.

After a new instance is launched, flows are required to be redistributed across a collection of instances with efficient state control. Other problems, such as load balance and flow state migration, have been well addressed in other research work [3], [4], [20].

B. State Space

In previous research, either traffic rate or runtime status is solely used as the metric of scaling detection. To improve performance, we employ both of them as composite features. Also, feature sequence in time series is used as model input, instead of only using current time's feature information. The utilization of neural networks makes it possible to establish sophisticated relationships between high-dimensional feature sequences and scaling actions.

More specifically, the following five features of each VNF are passed to the RL agent for processing: input traffic rate, output traffic rate, CPU utilization, memory utilization, and latency that identifies per-packet process time.

C. Action Space

After receiving the above features, the RL agent computes a scaling action $a_t^i \in \mathbb{A} = \{1, 0, -1\}$ for VNF i at time step t via neural networks, which is further applied to the NFV system. As mentioned above, $a_t^i = 1$ represents scaling out, $a_t^i = 0$ represents no scaling, and $a_t^i = -1$ represents scaling in.

D. Reward

The goal of cloud providers is to minimize the total cost incurred by the following two cases. In terms of delayed scaling detection, deployed instances do not have enough capability to serve incoming workloads. In terms of premature scaling detection, the extra cost is spent due to resource wasting. Both cases have adverse effects on the NFV system.

Generally, the overall system cost is composed of two parts: packet loss penalty and VNF instance cost. The former refers to the cost of packet loss caused by inadequate scaling detection. The latter refers to the cost of launching and running VNF instances. More specifically, when launching new VNF instances on servers, the process involves transferral of virtual machine (VM) images, booting and attaching them to devices, which is a deployment cost and should be seriously considered [21]. As for running VNF instances, its main cost is due to power consumption to operate servers and cooling facilities, which is primarily determined by the number and the types of VNFs running on VMs. For example, different Amazon EC2 instances are priced differently on different operating systems [22].

TABLE I
NOTATION.

Symbol	Remarks
$\mathcal{C}$	Service function chain
$\mathcal{V}$	Set of VNFs in $\mathcal{C}$
$\mathcal{E}$	Set of direct edges between VNFs in $\mathcal{V}$
$\mathbb{A}$	Scaling action space
d_i	Instance launching cost of VNF i
q_i	Instance running cost of VNF i
r_t^i	Traffic rate of demand i at time t
$\bar{C}_t$	Total packet loss penalty at time step t
$\ddot{C}_t^i$	Instance launching cost for VNF i at time step t
$\tilde{C}_t^i$	Instance running cost for VNF i at time step t
$\hat{C}_t$	Overall system cost at time step t
x_t^i	Initial input state vector of VNF i at time step t
y_t^i	Number of VNF i's deployed instances at time step t
v_t^i	Representation vector of VNF i at time step t
$x_t^{\mathcal{C}}$	Overall input state vector of $\mathcal{C}$ at time step t
$v_t^{\mathcal{C}}$	Representation vector of $\mathcal{C}$ at time step t
M	Length of state sequence
$\mathbf{p}_i$	Probability distribution for all possible actions for VNF i
$\mathbf{v}_i$	Performance of a policy for VNF i
$\mathbf{r}_t$	Reward at time step t

Let l_t be the revenue loss of per Gbit packet loss at time step t, and then the total packet loss penalty is denoted as:

$$\bar{C}_t = (r_t^0 - r_t^{n+1}) * l_t, \tag{1}$$

where r_t^0 identifies the input traffic rate of the whole system, r_t^{n+1} is the output traffic rate of the whole system.

Furthermore, let d_i indicate VNF i's per instance launching cost, and then the total instance launching cost for VNF i is:

$$\ddot{C}_t^i = d_i * [a_t^i]^+, \tag{2}$$

where the notation $[\cdot]^+$ is defined as $[x]^+ = \max\{x, 0\}$.

Similarly, assume q_i represents running cost per instance of VNF i. Therefore, the total instance running cost for VNF i is:

$$\tilde{C}_t^i = q_i * y_t^i, \tag{3}$$

where y_t^i is the number of VNF i's deployed instances at time step t.

Therefore, the overall system cost can be defined as:

$$\hat{C}_t = \lambda * \bar{C}_t + \mu * \sum_{i \in \mathcal{V}} (\ddot{C}_t^i + \tilde{C}_t^i). \tag{4}$$

where $\lambda + \mu = 1, \lambda, \mu \in [0, 1]$. Different cloud providers may have different packet loss penalty and instance cost, which means the values of λ and μ depends on circumstances.

As the goal of the training process is to maximize the reward, it can be defined as:

$$\mathbf{r}_t = -\hat{C}_t. \tag{5}$$

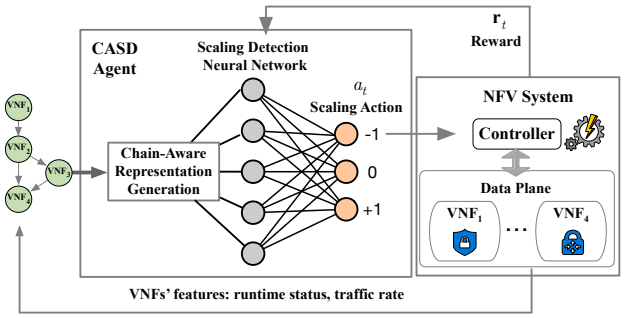


Fig. 2. Overview of CASD architecture. A CASD agent collects VNFs' online features from NFV system, makes scaling actions, and receives a reward. According to received reward $\mathbf{r}_t$, the agent updates parameters in the scaling detection neural network.

IV. CASD Overview

Fig. 2 presents CASD's overall architecture, where RL techniques are applied to solve the scaling detection issue in the NFV system. The CASD agent takes the state information as input and outputs VNFs' scaling actions. The state information includes VNFs' composite features and global chain information. The chain-aware representation generation module is utilized to transform the initial state information as fixed-sized vectors, including VNF representation (Section V-A) and global chain representation (Section V-B). Based on the above representations, the scaling detection neural network computes scaling actions of all VNFs (Section VI-A). After applying the calculated scaling actions, a reward $\mathbf{r}_t$ is passed back to the CASD agent to reflect how good the actions are. Based on $\mathbf{r}_t$, the CASD agent updates parameters in the scaling detection neural network. The goal of learning process is to maximize the expected cumulative discounted reward: $\mathbb{E}[\sum_{t=0}^{\infty}\gamma^t r_t]$, where $\gamma \in (0,1]$ is a factor discounting future rewards (Section VI-B). More specifically, CASD learns scaling detection policies using the A3C [15] algorithm, an actor-critic RL algorithm.

V. Chain-Aware Representation Generation

Commonly, a neural network takes fixed-sized vectors as inputs. At each step, CASD must convert the initial state information (global chain information and VNFs' features) into fixed-sized vectors, which are further passed to the RL-based neural network. However, there are several challenges in depicting the VNF and chain representations adequately. On the one hand, the chain may contain an arbitrary number of VNFs, which makes it hard to scale up if flatting all VNFs' state vectors together. On the other hand, the interconnection relationships between VNFs in the chain are also essential state features. How to effectively capture these relationships is also very critical to represent VNFs and global chain.

To address the above challenges, we design a chain neural network, which embeds the above information in a set of embedding vectors based on graph embedding techniques. As shown in Fig. 3, the graph embedding takes the chain as the input, whose VNF nodes carry a set of online status (*e.g.,* input rate, output rate, and CPU utilization) and outputs the following embedding vectors:

- **VNF Representation** exploits status information about the VNF node and its adjacent downstream VNF nodes.
- **Global Chain Representation** aggregates status information across VNFs within the chain.

A. VNF Representation

The CASD agent periodically receives feature information from the NFV system, and feeds it into neural networks. For VNF i at time step t, the initial state vector x_t^i is composed of five characteristics, i.e., $x_t^i = (input_t^i, output_t^i, latency_t^i, cpu_t^i, memory_t^i)$, where $input_t^i$ represents the input traffic rate measurement, $output_t^i$ is the output traffic rate measurement, $latency_t^i$ is the collected packet latency, cpu_t^i is the CPU utilization feature, $memory_t^i$ is the memory utilization feature.

Assume that VNF j is the child of VNF i as long as there is a directed edge $e : i \longmapsto j$, then we embed a VNF representation vector v_t^i from its state x_t^i and its children's states as follows:

$$h_t^i = W_h^i x_t^i + b_h^i, \tag{6}$$

$$v_t^i = exp(\sum_{(i \longmapsto j) \in \mathcal{E}} log(h_t^j)) + h_t^i, \tag{7}$$

where $h_t^i \in \mathbb{R}^d$ maps the input state of VNF i into hidden state space, $W_h^i \in \mathbb{R}^{d \times 5}$ and $b_h^i \in \mathbb{R}^d$ are weight matrix and bias for VNF i, respectively, both of which are initialized randomly and updated during the training process. The final VNF representation vector v_t^i is computed based on its hidden state and children's hidden states. $log(\cdot)$ and $exp(\cdot)$ are non-linear transformations that can capture complicated correlations among its children and find significant effects [14]. In this way, each VNF representation can not only capture its explicit state but also depict the effects of its children in the chain.

B. Global Chain Representation

Based on the above VNF representation, we compute a summary representation of the global chain. Intuitively, we can regard the chain as a particular VNF summary node, where all the VNFs in the chain $\mathcal{C}$ are its children. Hence, we can obtain the global chain representation $v_{\mathcal{C}}$ at time step t as follows:

$$v_t^{\mathcal{C}} = exp(\sum_{j \in \mathcal{V}} log(v_t^j)). \tag{8}$$

VI. Scaling Detection Model

Given the generated VNF and global chain representations in Section V, we design a neural network model to compute scaling actions of all VNFs. We adopt the A3C algorithm to train the model, a state-of-the-art actor-critic RL algorithm. Therefore, the scaling detection model consists of two components: actor network and critic network. The actor network depicts scaling detection policies, and the critic network evaluates how well these policies perform.

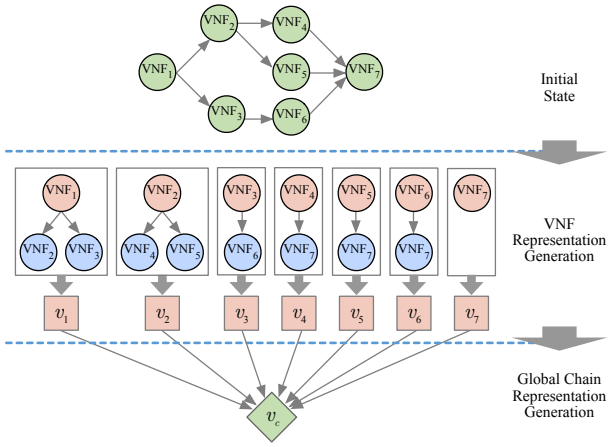


Fig. 3. VNF and global chain representation generation process.

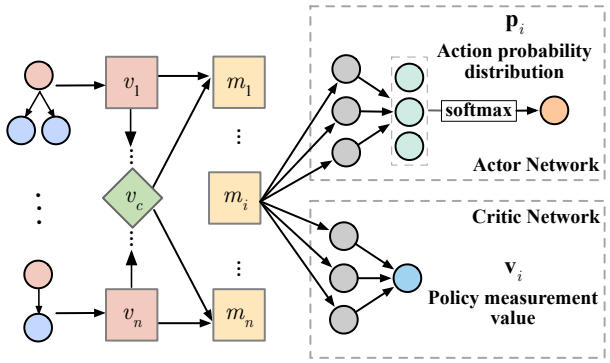


Fig. 4. For each VNF i, VNF embedding v_i and global chain embedding v_C are fed to the actor network and critic network, which compute scaling action distribution and measure policy performance, respectively.

A. Neural Network Model

Fig. 4 visualizes the structure of the designed neural network model. The actor and critic networks employ the same network structure under the output layer, while their output layers are quite different due to their different functionality. We first introduce the shared part of neural networks and then describe the output layers of the actor and critic networks, respectively.

As the VNF and chain status often contain noises, the scaling action might be choppy due to suddenly peaking/dropping signals if only taking the state at a specific time step into consideration. Hence, we propose to utilize states from the latest M time steps to make the final scaling action. To capture the sequential correlations among states at different time steps, we employ recurrent neural network (RNN), which is widely used in sequential pattern modeling problems, to model the VNF and chain state sequence. Formally, for VNF i, we obtain its input sequence in the latest M steps $X_i = \{x_1^i, x_2^i, \cdots, x_M^i\}$ and chain state sequence $X_{\mathcal{C}} = \{x_1^{\mathcal{C}}, x_2^{\mathcal{C}}, \cdots, x_M^{\mathcal{C}}\}$, then we apply aforementioned representation transformation to obtain the VNF representation sequence $\{v_1^i, v_2^i, \cdots, v_M^i\}$ and chain representation sequence $\{v_1^{\mathcal{C}}, v_2^{\mathcal{C}}, \cdots, v_M^{\mathcal{C}}\}$. Specifically, the local VNF representation v_k^i is further concatenated with global chain representation $v_k^{\mathcal{C}}$ to generate the overall VNF state $m_k^i = [v_k^i; v_k^{\mathcal{C}}] \in \mathbb{R}^{2d}$ at time step $k \in [1, M]$. For the overall state sequence $\{m_1^i, m_2^i, \cdots, m_M^i\}$, we apply Gated Recurrent Unit (GRU) [23], a special version of RNN. At time step k, GRU updates the model as follows:

$$z_k^i = \sigma(W_z^i m_k^i + U_z^i h_{k-1}^i), \tag{9}$$

$$r_k^i = \sigma(W_r^i m_k^i + U_r^i h_{k-1}^i), \tag{10}$$

$$\tilde{h}_k^i = tanh(\tilde{W}_h^i m_k^i + r_k^i \odot \tilde{U}_h^i h_{k-1}^i), \tag{11}$$

$$h_k^i = z_k^i \odot h_{k-1}^i + (1 - z_k^i) \odot \tilde{h}_k^i, \tag{12}$$

where σ is the sigmoid function, z_k^i is the update gate for VNF i which determines how much of the past information (from previous steps) needs to be passed to the future, r_k^i is the reset gate for VNF i which is employed to decide how much of the past information to forget, $\tilde{h}_k^i$ is current memory content for VNF i which uses the reset gate to store the relevant information from the past, h_k^i is the final memory output for VNF i which holds information from current unit and passes it down to the network, $\odot$ is Hadamard (element-wise) product operation, $W_z^i, W_r^i, \tilde{W}_h^i \in \mathbb{R}^{d \times 2d}, U_z^i, U_r^i, \tilde{U}_h^i \in \mathbb{R}^{d \times d}$ are weight matrices for VNF i.

Then we feed the final output h_M^i to actor and critic networks. As for the actor network, we apply a softmax layer to obtain the action probability distribution:

$$\mathbf{p}_i = softmax(W_p^i h_M^i + b_p^i), \tag{13}$$

where $\mathbf{p}_i \in \mathbb{R}^{|\mathbb{A}|}$ is the probability distribution for all possible actions for VNF i, $W_p^i \in \mathbb{R}^{|\mathbb{A}| \times d}$ and $b_p^i \in \mathbb{R}^{|\mathbb{A}|}$ are the weight matrix and bias for VNF i, respectively. In our implemented prototype, we set $|\mathbb{A}| = 3$, which is the number of all possible actions.

As for the critic network, we apply a fully connected layer to obtain a single value, which measures how well the policy performs:

$$\mathbf{v}_i = W_v^i h_M^i + b_v^i, \tag{14}$$

where $W_v^i \in \mathbb{R}^d$ and $b_v^i \in \mathbb{R}$ are the weight matrix and bias for VNF i, respectively.

B. Training Method

1) Theoretical Analysis: We utilize the A3C algorithm to train the above two neural networks. The detailed information is described below.

The critic network is utilized to learn an approximation of v^{π_θ} empirically, which indicates the expected reward corresponding to state m_t and policy π_θ. The critic network is trained using the standard Temporal Difference method [24]. The parameters of critic network θ_v are updated as follows:

$$\theta_v \leftarrow \theta_v - \alpha^c \sum_t \nabla_{\theta_v} (\mathbf{r}_t + \gamma V^{\pi_\theta}(m_{t+1}; \theta_v) - V^{\pi_\theta}(m_t; \theta_v))^2, \tag{15}$$

where α^c is the learning rate of the critic network, and $V^{\pi_\theta}(\cdot; \theta_v)$ is the estimate value of $v^{\pi_\theta}(\cdot)$, which is computed by the critic network.

According to the computation in [15], the actor network parameters θ are updated based on the following equation:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(m_t, a_t) A^{\pi_\theta}(m_t, a_t) + \beta \nabla_\theta H(\pi_\theta(\cdot|m_t)), \quad (16)$$

$$A(m_t, a_t) = \mathbf{r}_t + \gamma V^{\pi_\theta}(m_{t+1}; \theta_v) - V^{\pi_\theta}(m_t; \theta_v). \quad (17)$$

where α is the learning rate, $A^{\pi_\theta}(m, a)$ indicates the advantage function, which is the difference value between the reward when we conduct action a in state m, and the expected reward for action drawn from policy π_θ. $H(\cdot)$ indicates the entropy of the policy, namely action distribution probability. The parameter β is set to a great value at the beginning and decreases to improve rewards during the training process.

2) Training Process: The training process is summarized in Algorithm 1. The A3C algorithm is composed of one global network and multiple agent networks. Each agent has its own network parameters and a copy of environment, and runs on a separate thread. Firstly, each agent's parameters are reset to those of global networks. Then, each agent interacts with its environment and collects training tuples, including states, action and reward. Once the collected tuples are enough, these tuples are used to compute discount return and gradients with respect to its network parameters. Then, these gradients are used to update the global network's parameters. In this way, the global network's parameters could be updated by each agent. If a successful update is made to the global network, the above operations restart.

Algorithm 1: Training Process for CASD

Input: CASD model setting
prototype for generating states and rewards
Output: Trained scaling detection model
Initialization: Global shared parameter vectors: θ, θ_v
Global shared counter: $T \leftarrow 0$
Thread-specific parameter vector: $\theta^{'}$, $\theta_v^{'}$
Thread step counter: $t \leftarrow 1$
while $T \leq T_{max}$ **do**
  Reset gradients: $\mathrm{d}\theta \leftarrow 0$ and $\mathrm{d}\theta_v \leftarrow 0$
  Synchronize each thread's parameters $\theta^{'} = \theta$ and $\theta_v^{'} = \theta_v$
  $t_{start} = t$
  Get state m_t
  while *(m_t != terminal m_t)* $\wedge$ *($t - t_{start}$!= t_{max})* **do**
    Select an action a_t according to Section VI-A
    Conduct a_t to get return $\mathbf{r}_t$ and m_{t+1}
    $t \leftarrow t + 1$; $T \leftarrow T + 1$
  end
  if m_t == *terminal* m_t **then**
    $R = 0$
  else
    $R = V(m_t; \theta_v^{'})$
  end
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow \mathbf{r}_i + \gamma R$
    Accumulate gradients wrt $\theta^{'}$: $\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \nabla_{\theta'} \log \pi_{\theta'}(m_i, a_i)(R - V(m_i; \theta_v^{'}))$
    Accumulate gradients wrt $\theta_v^{'}$: $\mathrm{d}\theta_v \leftarrow \mathrm{d}\theta_v + \nabla_{\theta_v'}(R - V(m_i; \theta_v^{'}))^2$
  end
  Perform asynchronous update of θ and θ_v based on Equation 16 and 15
end

VII. Implementation

A. Prototype Implementation

We implement a prototype and evaluate system performance based on a testbed with several servers, each of which is equipped with two Intel(R) Xeon(R) E5-2620 V3 CPUs (2.40GHz, 12 physical cores). The above servers run Linux kernel 3.10.0.

The CASD prototype is implemented on top of OpenNetVM [16] version 18.05 and DPDK [25] version 17.08. We achieve a feature monitoring module in the data plane, which periodically pushes the collected VNFs' features to the controller. In the controller, the scaling detection module is installed to conduct online scaling detection.

The implementation of the CASD model includes two phases: offline training and online computing. In the offline training phase, we conduct a series of training experiments, which cover as many workload types as possible. In the online computing phase, scaling actions are computed according to the received features and the pre-trained model.

B. Hyper-parameter Setting

We train the CASD model using deep learning library TensorFlow [26]. For hyper-parameter setting, we set discount factor γ, learning rate α and α^c of actor and critic networks as 0.99, 10^{-2}, and 10^{-3}, respectively. The entropy factor β is controlled to decay from 1 to 0.1 over 10^4 iterations. Besides, the embedding size and dimension size d is set to 150.

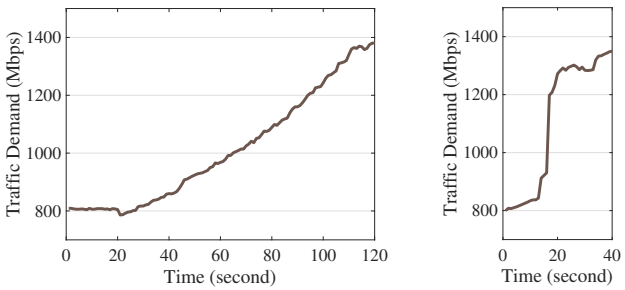
C. Traffic Generation

Due to the lack of public traces for VNF benchmarking, we synthetically generate workloads to evaluate system performance. We use MoonGen [27], a DPDK based packet generator, to generate the traffic. MoonGen runs in a separate server and is directly connected to the server that deploys CASD. The packet generator sends and receives test traffic to measure system performance, including latency and throughput. We use two types of traffic patterns to evaluate system performance: **Moderate Increase** and **Sharp Increase**, as shown in Fig. 5.

D. Baseline Algorithms

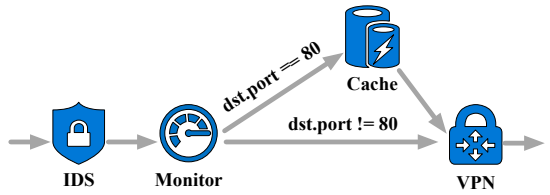
In our evaluation, we compare CASD's performance with that of the following three baseline algorithms:

- **Status-NN:** We implemented the proposed method in ENVI [11] using a fully connected neural network, which

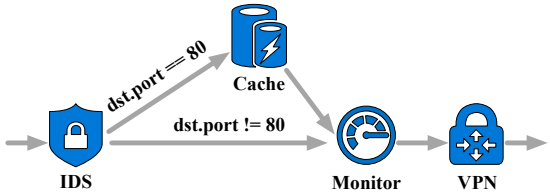


(a) Moderate Increase (b) Sharp Increase

Fig. 5. Two types of traffic patterns.



(a) Chain A



(b) Chain B

Fig. 6. Two tested service function chains.

utilizes each VNF's online status (i.e., latency, CPU and memory utilization) to achieve scaling detection but ignores the traffic rate. Similar to CASD, this neural network is also trained with RL.

- **Composite-NN:** We employ each VNF's traffic rate and online status as composite features. A neural network is used to establish the relationships between scaling actions and composite features. Composite-NN is also trained with RL.
- **Composite-DT:** We utilize a decision tree to achieve classification based on composite features. As for model training, we have labeled 5000 groups of data for each VNF.

E. Service Function Chain

Four network functions are implemented in the data plane to compose test service function chains, including intrusion detection system (IDS), network monitor, web cache, and VPN. The details of them are as follows:

- *IDS*: We implement it with 100 signature inspection rules using Snort [28], which is a widely used lightweight intrusion detection for networks.
- *Network Monitor*: It inspects incoming packets and maintains per-flow counters. Each flow is identified by the hash value of the 5-tuple.
- *Web Cache*: We implement it based on open-source caching proxy Squid [29]. The incoming packet with destination port 80 is forwarded to web cache.
- *VPN*: It processes incoming packets by two steps. Firstly, it encrypts packets with the AES algorithm. Secondly, it wraps the encrypted packets using the IPsec Authentication Header protocol.

The experiments are conducted on two chains, which are shown in Fig. 6. For chain A and B, we exchange the order of web cache and network monitor.

VIII. Evaluation

In this section, we evaluate the performance of the CASD prototype. Our experiments cover two types of traffic patterns and chains. Evaluation results answer the following questions:

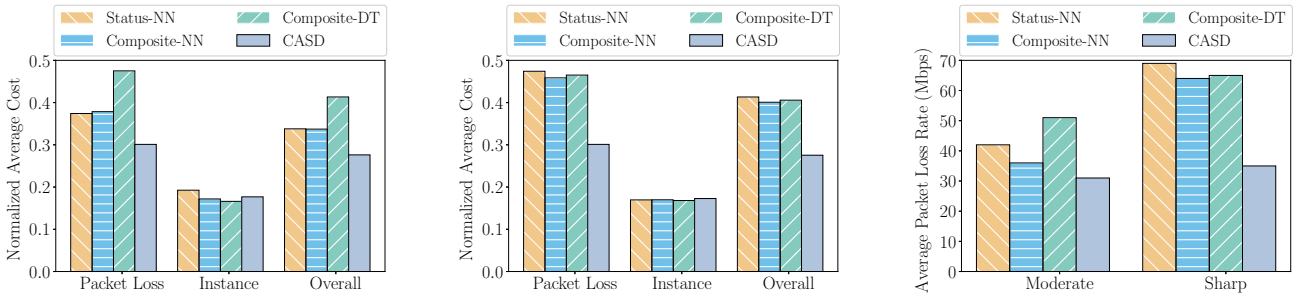
(1) *How does CASD perform compared with baseline scaling detection algorithms in terms of overall system cost? How does CASD improve system performance in terms of packet loss?* (Section VIII-A)

(2) *How does CASD work? How about the time agility of the proposed chain-aware scaling detection model?* (Section VIII-B)

A. Performance Comparison

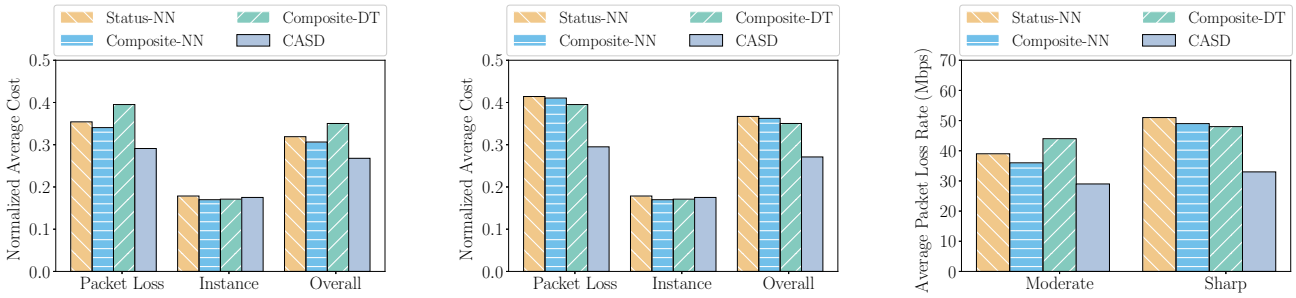
1) Overall System Cost: To evaluate CASD, we compare it with baseline algorithms on the overall system cost metric that is defined in Equation 4. Furthermore, to better understand the final results, we analyze the performance of subitems in our cost definition. More specifically, Fig. 7(a), 7(b), 8(a), and 8(b) compare CASD with baseline algorithms in terms of packet loss penalty, instance cost, and overall system cost. The packet loss penalty describes performance degradation due to scaling detection delay, which reflects the agility of a scheme. Instance cost is composed of VNF launching cost and running cost, which reflects a scheme's ability to filter noises. To compute the overall system cost, we set λ and μ to 0.8 and 0.2, respectively. Commonly, the violation of SLAs incurs penalties significantly. Therefore, we set a greater weight (λ) to packet loss penalty than that (μ) to instance cost.

In general, we find that CASD exceeds the performance of all baseline algorithms under different traffic patterns on different chains. In terms of packet loss penalty, CASD improves agility by using global chain information. Composite-DT has the worst system performance due to the inaccuracy of the hand-labeled data set. Status-NN and Composite-NN can adapt to different network environments better by utilizing RL techniques. However, they all have limited performance due to ignoring other VNFs' status information. In terms of instance cost, Status-NN sometimes has poor performance because of noises in the measured data. It is common that measurement results of status (*e.g.,* latency) contain noises due to system instability. We can also observe that CASD sometimes has a slightly higher instance cost compared with other baseline algorithms. This happens due to the inherent conflict between packet loss penalty and instance cost. More specifically, the more agile the scaling detection is, the lower the packet loss



(a) Average Cost under Moderate Increase (b) Average Cost under Sharp Increase (c) Average Packet Loss Rate

Fig. 7. Performance comparison between CASD and baseline algorithms on chain A.



(a) Average Cost under Moderate Increase (b) Average Cost under Sharp Increase (c) Average Packet Loss Rate

Fig. 8. Performance comparison between CASD and baseline algorithms on chain B.

penalty is. Although this is achieved at the expense of slightly higher instance cost, the overall system cost is becoming lower. Moreover, we gain some insights from these results as follows:

- **Effectiveness of Reinforcement Learning:** Firstly, we demonstrate the effectiveness of RL. Under the moderately-increasing traffic pattern, there is a vast difference between the performance of Composite-DT and that of the other three schemes. Under the sharply-increasing traffic pattern, Composite-DT, Status-NN, and Composite-NN have similar performance. Therefore, we can conclude that Composite-DT cannot agilely detect scaling events in a finer-grained manner. Composite-DT is trained based on hand-labeled data sets. However, it is not very easy to always obtain accurate data labels due to the complexity of network environments. Compared with Composite-DT, RL-based methods learn scaling detection policies directly through observations of the resulting performance of past decisions, which enable them to adapt to complex network environments.
- **Effectiveness of Composite Features:** Secondly, it is better to employ composite features as inputs than solely status features. We find that the performance of Composite-NN either matches or exceeds the performance of Status-NN. Composite-NN has better performance due to adding traffic rate as input features. Compared with all baseline algorithms, CASD further takes the composite feature sequence in time series as input, which is helpful to filter noises and better predict future demand.
- **Effectiveness of Chain Information:** Thirdly, the introduction of chain information can improve system performance. We observe that the performance of baseline algorithms on chain B is better than that on chain A. On chain A, the web cache with low throughput impedes the scaling detection of network monitoring with higher throughput. This case does not happen to chain B due to the order exchange of them. Furthermore, we also find that baseline algorithms' performance under the moderately-increasing traffic pattern is better than that under the sharply-increasing traffic pattern. Compared with them, CASD can adapt to different kinds of traffic patterns and chains and has relatively stable and high performance in all scenarios. CASD incorporates global chain information into scaling detection policies, which allows CASD to plan VNFs' scaling sequence from a high level efficiently.

2) Packet Loss: In this section, we evaluate the system performance of each scheme on the metric of average package loss rate. The average packet loss rate of a scheme is computed by the difference between the optimal offline average packet rate and the actual average packet process rate of a scheme. We first compute the result of the optimal offline scheme using a complete input traffic rate in the assumption of no packet loss. The optimal offline scheme serves as an upper bound on the average package process rate, which requires complete

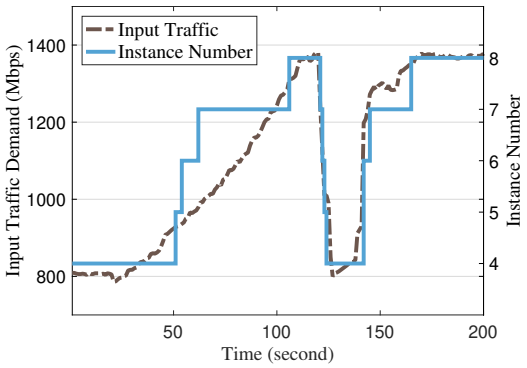


Fig. 9. The variation of total VNF instance number of chain A with respect to input traffic demand.

and perfect knowledge of the future traffic demand.

Fig. 7(c) and 8(c) show the average packet loss rate that each scheme achieves under two types of traffic patterns and chains. We can observe that under any conditions, CASD performs best and is closest to the optimal offline scheme. However, in the real world, an optimal offline scheme cannot be achieved. Therefore, there is little room to improve system performance over CASD for online algorithms.

B. CASD Deep Dive

In this section, we describe benchmarks that provide a deeper understanding of CASD and shed light on some practical concerns in the real world.

1) CASD Working Process: The behavior of CASD is shown in Fig. 9, which depicts the variation of the number of all VNF instances concerning the input traffic demand over time. This experiment is conducted under the condition of the moderately-increasing traffic pattern and service function chain A. At the beginning, the total number of all instances is 4, which corresponds to 4 VNFs in the chain. The first scaling action is to increase one instance of the network monitor. The second is to run a new web cache instance, which happens during the setup process of the second network monitor. In the meantime, the output traffic volume from the booting network monitor is lower than the system's input traffic demand, which may impede scaling detection of the downstream web cache. However, CASD can still agilely detect the network monitor's scaling requirement by using the global chain's information. The third and fourth scaling actions are to increase to 2 VPN and 2 IDS instances, respectively. With the slumping traffic demand coming next, the total number of VNF instances decreases to 4 again. As the traffic demand is then sharply increasing, the web cache, network monitor, and VPN are almost synchronously scaled up. As we can see, during the setup phase, the upstream network monitor with lower throughput does not impede scaling detections of the downstream web cache and VPN with higher throughputs. Finally, the IDS is also scaled up to meet the increased traffic demand.

2) Offline Training Time: As for offline training, we measure the time of generating a scaling detection algorithm using RL. Training an algorithm requires about 40,000 iterations, each of which took about 500 ms. Hence, in total, the whole training process took about 6 hours. Traditional methods usually require several weeks/months for algorithm design and significant tuning in different environments. Compared with these methods, CASD dramatically reduces development time.

3) Online Computing Time: In terms of online computing, the run time of the installed scaling detection algorithm is about 0.14-0.16 seconds. Commonly, a new instance setup requires several seconds/minutes. Therefore, the above scaling action computing time is negligible for whole system performance.

IX. Related Work

VNF Placement and Elastic Scaling in NFV: In recent years, lots of efforts have been devoted to tackling the VNF placement problem, including NFVdeep [30], DDQN-VNFPA [31]. At the same time, elastic scaling in NFV aims to make dynamic and efficient resource allocation to serve varying workloads. These efforts include high-performance instance migration [3], [4], [20], [32] and migration flow selection [33], [34]. These proposals can ensure safe and high-performance packet processing once a VNF's scaling requirement is detected. CASD solves scaling detection issue, which is orthogonal and complementary to all above proposals.

Scaling Detection in NFV: Most of scaling detection algorithms can be primarily classified into two types: rate-based [7]–[10], [35], [36] and status-based [11], [12]. The rate-based approaches first estimate the upcoming traffic rate and then compute the number of required instances that can process the estimated traffic demand. However, the dynamics of upcoming traffic in packet size and type affect instance number computation. Status-based approaches achieve scaling detection based on VNFs' runtime status, including the application- and hardware-level parameters. However, it is affected by the collected "raw" status information, which causes imprecise scaling decisions. Compared with them, CASD jointly considers composite features as inputs to improve precision and agility. Besides, CASD also incorporates global chain information into control policies to further improve system performance.

Scaling Detection in Cloud Computing: In cloud computing, automatic scaling is also a significant problem to serve dynamically varying workload [37]. There have been some scaling detection mechanisms based on reinforcement learning, which can achieve autonomic resource allocation in clouds [38]–[41]. Compared with them, the scaling detection mechanism in NFV must consider the influence of the service function chain. The interconnection relationships between VNFs profoundly affect agility. CASD is a chain-aware scaling detection mechanism, which incorporates the global chain information into control policy.

X. Conclusion

We present CASD which utilizes reinforcement learning and neural networks to automatically learn scaling detection

policies without any human instructions. To further improve agility and system performance, CASD incorporates global chain information into control policies to efficiently plan the scaling sequence of VNFs within the chain. To build CASD, we develop scalable representations for VNFs and global chain, design neural networks based on feature sequence, and utilize the A3C algorithm for model training. We have implemented a prototype on top of the NFV system and compare it with multiple baseline algorithms over different traffic patterns and chains. Evaluation results show that CASD outperforms the state of the arts in terms of overall system cost and packet processing rate.

REFERENCES

[1] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-art and Research Challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[2] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "NFV: State of the Art, Challenges and Implementation in Next Generation Mobile Networks (vEPC)," *IEEE Network*, vol. 28, no. 6, pp. 18–26, 2014.

[3] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling Innovation in Network Function Control," in *Proc. of ACM SIGCOMM*, 2014, pp. 163–174.

[4] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic Scaling of Stateful Network Functions," in *Proc. of USENIX NSDI*, 2018, pp. 299–312.

[5] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proc. of ACM IMC*, 2010, pp. 267–280.

[6] G.Linden, "Make data useful," in *https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt*.

[7] X. Wang, C. Wu, F. Le, A. Liu, Z. Li, and F. Lau, "Online VNF Scaling in Datacenters," in *Proc. of IEEE CLOUD*, 2016, pp. 140–147.

[8] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic Virtual Network Function Placement," in *Proc. of IEEE CloudNet*, 2015, pp. 255–260.

[9] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive VNF Scaling and Flow Routing with Proactive Demand Prediction," in *Proc. of IEEE INFOCOM*, 2018, pp. 486–494.

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

[11] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "ENVI: Elastic Resource Flexing for Network Function Virtualization," in *Proc. of USENIX HotCloud*, 2017.

[12] L. Cao, S. Fahmy, P. Sharma, and S. Zhe, "Data-Driven Resource Flexing for Network Functions Visualization," in *Proc. of ACM ANCS*, 2018, pp. 111–124.

[13] H. Mao, R. Netravali, and M. Alizadeh, "Neural Adaptive Video Streaming with Pensieve," in *Proc. of ACM SIGCOMM*, 2017, pp. 197–210.

[14] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning Scheduling Algorithms for Data Processing Clusters," in *Proc. of ACM SIGCOMM*, 2019, pp. 270–288.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in *Proc. of the 33nd International Conference on Machine Learning*, 2016, pp. 1928–1937.

[16] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A Platform for High Performance Network Service Chains," in *Proc. of the ACM SIGCOMM Workshop on Hot topics in Middleboxes and Network Function Virtualization*, 2016, pp. 26–31.

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

[18] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization," in *Proc. of ACM SIGCOMM*, 2018, pp. 191–205.

[19] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *Proc. of ACM SIGCOMM*, 2016, pp. 511–524.

[20] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *Proc. of USENIX NSDI*, 2013, pp. 227–240.

[21] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska, "Dynamic Right-sizing for Power-proportional Data Centers," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 5, pp. 1378–1391, 2013.

[22] "Amazon EC2 Pricing on Demand," 2019. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[23] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-decoder for Statistical Machine Translation," in *Proc. of EMNLP*, 2014, pp. 1724–1734.

[24] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, ser. Adaptive Computation and Machine Learning. MIT Press, 1998. [Online]. Available: http://www.worldcat.org/oclc/37293240

[25] Intel, "Data Plane Development Kit," 2014.

[26] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A System for Large-Scale Machine Learning," in *Proc. of USENIX OSDI*, 2016, pp. 265–283.

[27] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A Scriptable High-speed Packet Generator," in *Proc. of ACM IMC*, 2015, pp. 275–287.

[28] M. Roesch *et al.*, "Snort: Lightweight Intrusion Detection for Networks," in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[29] "Squid. http://www.squid-cache.org/."

[30] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, "Nfvdeep: Adaptive online service function chain deployment with deep reinforcement learning," in *Proceedings of the International Symposium on Quality of Service*, 2019, pp. 1–10.

[31] J. Pei, P. Hong, M. Pan, J. Liu, and J. Zhou, "Optimal vnf placement via deep reinforcement learning in sdn/nfv-enabled networks," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 263–278, 2019.

[32] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamatian, "Transparent Flow Migration for NFV," in *Proc. of IEEE ICNP)*, 2016, pp. 1–10.

[33] C. Sun, J. Bi, Z. Meng, X. Zhang, and H. Hu, "OFM: Optimized Flow Migration for NFV Elasticity Control," in *Proc. of IEEE/ACM IWQoS*, 2018, pp. 1–10.

[34] B. Zhang, P. Zhang, Y. Zhao, Y. Wang, X. Luo, and Y. Jin, "Co-Scaler: Cooperative Scaling of Software-Defined NFV Service Function Chain," in *Proc. of IEEE NFV-SDN*, 2016, pp. 33–38.

[35] H. Tang, D. Zhou, and D. Chen, "Dynamic Network Function Instance Scaling Based on Traffic Forecasting and VNF Placement in Operator Data Centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 530–543, 2018.

[36] Y. Gu, Y. Hu, Y. Ding, J. Lu, and J. Xie, "Elastic Virtual Network Function Orchestration Policy Based on Workload Prediction," *IEEE Access*, vol. 7, pp. 96 868–96 878, 2019.

[37] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[38] E. Barrett, E. Howley, and J. Duggan, "Applying Reinforcement Learning Towards Automating Resource Allocation and Application Scalability in the Cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[39] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: Towards a Fully Automated Workflow," in *Proc. of ICAS*, 2011, pp. 67–74.

[40] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling," in *Proc. of IEEE/ACM CCGrid*, 2017, pp. 64–73.

[41] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and Vertical Scaling of Container-Based Applications using Reinforcement Learning," in *Proc. of IEEE CLOUD*, 2019, pp. 329–338.