Management and Orchestration of Virtual Network Functions via Deep Reinforcement Learning

Joan S. Pujol Roig, David M. Gutierrez-Estevez, and Deniz Gündüz

***Abstract*—Management and orchestration (MANO) of resources by virtual network functions (VNFs) represents one of the key challenges towards a fully virtualized network architecture as envisaged by 5G standards. Current threshold-based policies inefficiently over-provision network resources and under-utilize available hardware, incurring high cost for network operators, and consequently, the users. In this work, we present a MANO algorithm for VNFs allowing a central unit (CU) to learn to autonomously re-configure resources (processing power and storage), deploy new VNF instances, or offload them to the cloud, depending on the network conditions, available pool of resources, and the VNF requirements, with the goal of minimizing a cost function that takes into account the economical cost as well as latency and the quality-of-service (QoS) experienced by the users. First, we formulate the stochastic resource optimization problem as a parameterized action Markov decision process (PAMDP). Then, we propose a solution based on deep reinforcement learning (DRL). More precisely, we present a novel RL approach, called parameterized action twin (PAT) deterministic policy gradient, which leverages an *actor-critic architecture* to learn to provision resources to the VNFs in an online manner. Finally, we present numerical performance results, and map them to 5G key performance indicators (KPIs). To the best of our knowledge, this is the first work that considers DRL for MANO of VNFs' physical resources.**

***Index Terms*—Deep reinforcement learning, resource allocation, software defined networks, virtual network functions, wireless edge processing.**

I. Introduction

Traditionally the deployment of new network functions (NFs) has been done through the acquisition and installation of a proprietary hardware running a licensed software. This fact reduces the incentives for network operators in updating their network's physical architecture to offer new services or update existing ones, as it represents an increase in both the capital expenditures (CAPEX), i.e., equipment inversion, equipment installation and personnel training, and operational expenditures (OPEX), i.e., the cost of of operating the system [1]. To overcome this limitation, network function virtualization (NFV) has been proposed to curtail constant acquisition of technical hardware, by leveraging virtualization technology to implement NFs using general purpose computers/servers [2]. With virtualization, software implementation of a NF can be decoupled from the underlying hardware, i.e., NFs can be instantiated without the need of new equipment acquisition and installation, and they can run over commercial off-the-shelf hardware. The isolation of software from hardware allows for a set of VNFs to be deployed on a shared pool of resources. This motivates a solution to manage the underlying shared infrastructure (processing power, storage, etc.) in an efficient, scalable and rapid manner.

There has been a lot of work on resource allocation for cloud networks. One of the most popular ways to address resource provisioning is threshold-based reactive approaches, where resources are added or removed if the network's condition reaches certain predefined thresholds [3]–[6]. Although this provides a simple and scalable solution to dynamic resource allocation, threshold-based criteria tend to over-provision and under-utilize network equipment (incurring high costs for the infrastructure provider) and make the management of dynamic traffic and deployment of new types of services difficult as network traffic models must be elaborated beforehand. In [7], authors study the scaling of virtual machines (VMs) in a proactive way. In particular they propose a solution via decision tree approach to resolve whether a VM instance should be *vertically scaled*; that is, more physical resources (e.g., processing power, storage) should be added, or *horizontally scaled*, i.e., by deploying a new VM instance. An autonomous vertical scaling approach is proposed in [8] using Q-learning, where an agent learns how to autonomously provision resources (storage and processing power) to a VM.

With the explosion of machine learning (ML) and virtualization technologies, and their applications to communication networks, the idea of self-governing networks leveraging modern ML techniques is becoming popular among the communications research community. Chen et. al. [9] proposes deep double Q-learning (DDQ) and deep-SARSA solutions for mobile edge computing, where an end user terminal with limited local computation and energy resources jointly optimizes computation offloading and energy consumption selection in an autonomous manner. The end user terminal decides whether to execute a computing task locally or offload it to one or more of the available edge base stations (BSs), also selecting the amount of energy to be allocated for the task in question. A proactive VM orchestration solution is proposed in [10] using Q-learning, where, given the current state, an agent decides to increase, reduce or retain the number of VMs allocated to a VNF. In [**?**], a deep learning approach is introduced to decide the number of VNFs that must be deployed to meet the network traffic demands. The authors formulate a classification problem, where each class corresponds to the

J. Pujol Roig and Dr. Gündüz are with the Department of Electrical and Electronic Engineering at Imperial College London, UK (Emails: d.gunduz@imperial.ac.uk, j.pujol-roig16@imperial.ac.uk). David M. Gutierrez-Estevez is with Samsung Electronics R&D Institute UK, Surrey, TW18 4QE, UK (Email: d.estevez@samsung.com). This work was supported in part by the European Research Council (ERC) Starting Grant BEACON (grant agreement no. 725731). J. S. Pujol Roig acknowledges funding from the Engineering and Physical Sciences Research Council (EPSRC) and Toshiba Research Europe through an iCASE award to carry out his PhD studies.

number of VNFs that must be instantiated to be able to cope with the current traffic, and use historical labelled traffic data to train the proposed algorithm.

More recently, in the management and network orchestration (MANO) domain for wireless networks resources, the use of DRL has gained attraction for network slicing resource orchestration and management. These works formulate a discrete action selection optimization problem, and use well established value-based methods, e.g., Q-learning or SARSA, to solve the formulated problem. In this line of work, [12], proposes a deep Q-learning approach to radio resource slicing and priority-based core network slicing, showing its advantage in addressing demand-aware resource allocation. In [12], authors formulate the problem of frequency band allocation, and the problem of computation resource orchestration for different slices. These problems are reduced to choosing a particular configuration from a finite set of available configurations, which is done leveraging DDQNs. Similarly, in [13], a DRL solution based on DDQN is presented for multi-tenant cross-slice resource orchestration, where again, a discrete number of communication and computation resources have to be allocated to different slice tenants. Finally, a deep deterministic policy gradient (DDPG) method with advantage function is employed in [14] to allocate bandwidth resources to different network slices. Compared to the aforementioned approaches, the continuous nature of DDPG allows for more fine-grained resource allocation.

In our work, we consider 3GPP functional split, where a central unit (CU) deploys and maintains a set of VNFs serving the users of several distributed units (DUs). We first formulate the dynamic allocation of processing and storage resources to VNFs as a Markov decision process (MDP). The optimal solution for this problem is elusive due to prohibitively large state and action spaces. Therefore, we present a novel deep reinforcement learning (DRL) algorithm, called *parameterized action twin* (PAT), where we use DDPG [15] and its novel variant called twin delayed DDPG [16], as well as ideas from the parameterized action Markov decision process (PAMDP) as in [16], [17], so that an agent placed at the CU is trained to learn whether to scale vertically (add processing power and storage), horizontally (instantiate new VNFs), or to offload (send the VNFs to the cloud) based on the system state (service request arrivals, service rates, service level agreement (SLA), etc.), using a cost function that combines the economic cost, SLA requirements, and the latency experienced by the users. The proposed algorithm is deployed in a variety of scenarios and its performance is evaluated according to a defined set of 5G key performance indicators (KPIs).

The feasibility of the proposed solution relies on the assumption that the technology envisaged for NF virtualization is *"containerization"* [18], where containers perform operating-system-level virtualization, i.e., every time a VNF is launched a container is deployed in a physical server. A "container" *is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings and its run by the operating system kernel* [19]. Containers are isolated from one another and can communicate with each other through well-defined channels. We find containers to be a more appropriate virtualization technology, compared to others, such as VMs, as they require less power [20], take less start-up and re-scaling time [21], [22], and, most importantly, can be rescaled on-the-fly without disrupting the service they provide.

Due to the use of an actor-critic architecture, our approach is a joint policy and action-value-based optimization, which generally shows better convergence properties [15] compared to value-based approaches implemented in [9], [10], [12], [12], [13]. Moreover, Q-learning and SARSA are used for discrete action selection [23], which is not feasible for the continuous control problem addressed in this work. Although a continuous action space is considered in [14], it focuses on the allocation of a single resource (bandwidth), while we consider the allocation of two continuous resources plus a discrete action for server selection. In contrast to [10], we consider not only horizontal scaling but also vertical scaling, as well as offloading, significantly increasing the complexity of the problem. Furthermore, our algorithm works in an online manner, i.e., dynamically adapting to the network traffic, which differs from [11], where the algorithms are trained using historic labelled data and cannot adapt to new types (i.e., classes) of traffic that differs significantly from the training set. Moreover, in disagreement with what is stated in [11] for reinforcement learning (RL) approaches, our approach can use unlabelled historical data to learn, as we are interested in the network patterns (captured by the historical arrival and service times) to update the critic value-function estimates accordingly. Finally, in comparison with the cloud management algorithm presented in [7], using deep neural networks (DNNs) for function approximation can handle a higher dimensional state space, which would be challenging to be captured using decisions trees due to the exponential growth in the number of leaves.

The remainder of this paper is organized as follows: In Section II the system model is introduced. The problem formulation using a Markov decision process (MDP) framework is presented in Section III. In Section IV we provide an overview of the RL notation, and review the works upon which our approach is based. The proposed PAT algorithm used to train the agent is explained in Section V. Numerical results illustrating the performance of the PAT algorithm are presented in Section VI. Finally, a summary of the results and conclusions are presented in Section VII.

Notation: $[\cdot]^T$ denotes the transpose operation. $\mathbb{1}(x)$ denotes the logical operator, which equals to 1 if x is true, and 0 otherwise. For positive integer K, $[K]$ denotes the set $\{1, 2, \ldots, K\}$. 1_N denotes the vector of 1s of size N. For set $\mathcal{A}$, we denote its power set, i.e., the set of all subsets of $\mathcal{A}$, by $2^{\mathcal{A}}$. We define the function $clip\left(x, x_{min}, x_{max}\right) \triangleq \max\left\{x_{min}, \min\left\{x_{max}, x\right\}\right\}$.

II. System Model

We consider a radio access network (RAN) with the 3GPP CU-DU functional split, consisting of B small-cell BSs (the DUs), denoted by $\mathcal{B} = \{\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_\mathrm{B}\}$, connected to a CU that is in charge of the MANO of the NFs, such as transfer of user data, mobility control, RAN sharing, positioning,

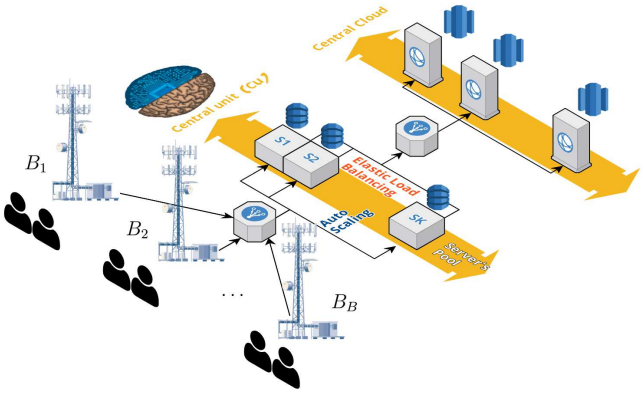


Fig. 1: Considered system architecture.

session management, etc. The BSs work as remote radio heads (RRHs), i.e., relaying all its traffic to the CU. Let $\mathcal{N} = \{\mathrm{N}_0, \ldots, \mathrm{N}_{\mathrm{N}-1}\}$ denote the set of distinct heterogeneous NFs offered by the CU that can be instantiated by any traffic requirement in the network. Based on the users' traffic requirements from $B_i, i \in [B]$, the CU deploys and maintains a subset of VNFs from set $\mathcal{N}$. See Figure 1 for an illustration of the network model.

We envisage an autonomous CU that has a local pool of resources that can be used to instantiate new VNFs, or to maintain deployed ones. The pool of local resources consists of K servers denoted by $\{\mathrm{S}_1, \ldots, \mathrm{S}_\mathrm{K}\}$, each with a limited processing and storage capability. We assume homogeneity across servers, such that each S_k, $k \in [K]$, has the same storage size of $\eta_{max}F$ bits and a central processing unit (CPU) of capability $\rho_{max}C$ Hz. In addition to the local servers, the CU can also employ resources located at a cloud center by offloading VNFs to the cloud, albeit at an increased cost which will be specified later. We consider a central cloud with an infinite capacity resource pool, and denote it by S_{K+1}, so the set of available resources to the CU is denoted by $\mathcal{K} = \{\mathrm{S}_1, \ldots, \mathrm{S}_{\mathrm{K}+1}\}$.

We consider a slotted resource allocation scheme, where new users arriving at the system wait until the start of the next slot to be allocated resources. Thus, the time horizon is discretized into decision epochs, corresponding to slots of duration T, and are indexed by an integer $t \in \mathbb{N}^+$. At the beginning of each epoch, the CU decides how to allocate the network resources. We denote by $\mathcal{N}^{(t)} \subset \mathcal{N}$, the set of active VNFs maintained by the CU during epoch t. The dependency of $\mathcal{N}^{(t)}$ on t emphasizes the fact that VNFs can be added or removed from the set of services over time. Let us refer to $\rho_k^{(t)} \leq \rho_{max}$ and $\eta_k^{(t)} \leq \eta_{max}$ as the total processing power and storage, respectively, of server S_k being used at epoch t. The CU is connected to the cloud via a dedicated link of capacity $R^{(t)}$ Mbps. We denote by $N_{k,j}$ the instance of N_j deployed at server S_k, and allow only for one instance of each VNF to be deployed in a server.

We consider two main physical resources to be provisioned to the VNFs, CPU and memory, and assume that NF instance $N_{k,j}, \forall j \in [N]$, at epoch t uses $c_{k,j}^{(t)}C$ $\mathrm{H_z}$ of CPU capability and $m_{k,j}^{(t)}B$ bits of storage, at server S_k, $k \in [K+1]$, where $c_{k,j}^{(t)} \in \left[\underline{c}_{k,j}^{(t)}, \overline{c}_{k,j}^{(t)}\right]$ and $m_{k,j}^{(t)} \in \left[\underline{m}_{k,j}^{(t)}, \overline{m}_{k,j}^{(t)}\right]$. Thus, each VNF has a different resource range in which it can operate following the definition of an *elastic NF*, which refers to a VNF whose QoS *gracefully* degrades with the scarcity of resources [24]. The range of CPU and memory resources VNF $N_{k,j}$ is able to operate at is given by:

$$\begin{cases} \underline{c}_{k,j}^{(t)} = c_{j,0} + (c_{j,r} - \Delta c_{j,d})\, u_{k,j}^{(t)} \\ \overline{c}_{k,j}^{(t)} = c_{j,0} + (c_{j,r} + \Delta c_{j,d})\, u_{k,j}^{(t)} \\ \underline{m}_{k,j}^{(t)} = m_{j,0} + (m_{j,r} - \Delta m_{j,d})\, u_{k,j}^{(t)} \\ \overline{m}_{k,j}^{(t)} = m_{j,0} + (m_{j,r} + \Delta m_{j,d})\, u_{k,j}^{(t)} \end{cases},$$

where $u_{k,j}^{(t)}$ denotes the number of users being served by VNF instance $N_{k,j}$ at epoch t; $c_{j,0}$ and $m_{j,0}$ represent the offset CPU and memory requirements, respectively, that do not depend on the number of users being served. The variables $c_{j,r}$ and $m_{j,r}$ account for the linear increment of CPU and memory per user being served by the particular deployment of N_j in server S_k. Values $\Delta c_{j,d}$ and $\Delta m_{j,d}$ are referred as the *elastic service coefficients*, and define the resource range under which the VNF is able to operate.

The QoS of the $u_{k,j}^{(t)}$ users served by the instance of VNF $N_{k,j}$ is denoted by $\mathrm{QoS}_{k,j}^{(t)}$, and depends on the resources allocated to this VNF instance. VNF N_j, $j \in [N]$, has a minimum QoS requirement, QoS_j^{min}, that must always be ensured as specified by the SLA, and a maximum perceived QoS, QoS_j^{max}.

We assume that $\mathrm{QoS}_{k,j}^{(t)}$ as a function of $m_{k,j}^{(t)}$ and $c_{k,j}^{(t)}$ is given by the following piecewise function:

$$\mathrm{QoS}_{k,j}^{(t)} = \begin{cases} \mathrm{QoS}_j^{max}, \text{ if } c_{k,j}^{(t)} > \overline{c}_{k,j}^{(t)} \text{ and } m_{k,j}^{(t)} > \overline{m}_{k,j}^{(t)} \\ 0, \text{ if } c_{k,j}^{(t)} < \underline{c}_{k,j}^{(t)} \text{ or } m_{k,j}^{(t)} < \underline{m}_{k,j}^{(t)} \\ \frac{\mathrm{QoS}_j^{max} - \mathrm{QoS}_j^{min}}{\overline{r}_{k,j}^{(t)} - \underline{r}_{k,j}^{(t)}} \left(\min\{m_{k,j}^{(t)}, \overline{m}_{k,j}^{(t)}\} + \min\{c_{k,j}^{(t)}, \right. \\ \left. \overline{c}_{k,j}^{(t)}\} \right) + \frac{\mathrm{QoS}_j^{min} \overline{r}_{k,j}^{(t)} - \mathrm{QoS}_j^{max} \underline{r}_{k,j}^{(t)}}{\overline{r}_{k,j}^{(t)} - \underline{r}_{k,j}^{(t)}}, \text{ otherwise} \end{cases},$$

where we defined $\overline{r}_{k,j}^{(t)} \triangleq \overline{c}_{k,j}^{(t)} + \overline{m}_{k,j}^{(t)}$ and $\underline{r}_{k,j}^{(t)} \triangleq \underline{c}_{k,j}^{(t)} + \underline{m}_{k,j}^{(t)}$. We see that $N_{k,j}$ satisfies the SLA if and only if $m_{k,j}^{(t)} \geq \underline{m}_{k,j}^{(t)}$ and $c_{k,j}^{(t)} \geq \underline{c}_{k,j}^{(t)}$, as these result in $\mathrm{QoS}_{k,j}^{(t)} \geq \mathrm{QoS}_j^{min}$, and that additional resources beyond $\overline{c}_{k,j}^{(t)}$ and $\overline{m}_{k,j}^{(t)}$ do not have an impact on the QoS perceived by the users, which saturates at QoS_j^{max}. Furthermore, based on Eqn. (II) the QoS is the same for all the users served by the same VNF instance $N_{k,j}$, i.e., $u_{k,j}^{(t)}$. The CU can also offload VNFs to the cloud, in which case the CU's local pool is not used.

We define $\mathbf{c}_k^{(t)} \triangleq \left[c_{k,1}^{(t)}, \ldots, c_{k,N}^{(t)}\right]^\mathsf{T}$ and $\mathbf{m}_k^{(t)} \triangleq \left[m_{k,1}^{(t)}, \ldots, m_{k,N}^{(t)}\right]^\mathsf{T}$. Finally, the matrices $\mathbf{C}^{(t)} \triangleq [\mathbf{c}_1^{(t)}, \ldots, \mathbf{c}_K^{(t)}]$ and $\mathbf{S}^{(t)} = [\mathbf{m}_1^{(t)}, \ldots, \mathbf{m}_K^{(t)}]$ represent the CPU and memory allocations across all the K servers at epoch t, such that $\rho_k^{(t)} = 1_N \cdot \mathbf{c}_k^{(t)}$ and $\eta_k^{(t)} = 1_N \cdot \mathbf{m}_j^{(t)}$.

The number of new service requests from all the BSs for VNF $N_j, j \in [N]$, in epoch t, is denoted by $n_j^{(t)}$, and is assumed to follow an independent and identically distributed (i.i.d.) homogeneous Poisson process with parameter $\lambda_j^{(t)}$; in

other words, the probability of $n_j^{(t)}$ new demands to arrive at the CU for VNF N_j in epoch t for a time-slot of duration T is given by:

$$P\left(n_j^{(t)}=n\right)=\frac{\left(\lambda_j^{(t)}T\right)^n}{n!}\mathrm{e}^{-\lambda_\mathrm{j}^{(\mathrm{t})}\mathrm{T}}.$$

Remark 1. In order to capture slow variations of network traffic over time, we consider time-varying $\lambda_j^{(t)}$ values, obtained by sampling a Gaussian distribution with parameters μ_j and σ_j and taking the maximum between the obtained value and 0, i.e., $\lambda_j^{(t)}=\max\{x,0\}$ where $x\sim\mathcal{N}(\mu_j,\sigma_j)$. We assume value of λ_j is kept constant for a block of t_{max} time slots, and changes to an independent realization from the aforementioned truncated Gaussian distribution for the next block. $R^{(t)}$ is also obtained by sampling a truncated Gaussian distribution: $R^{(t)}=\max\{x,R_{min}\}$, where $x\sim\mathcal{N}(\mu_r,\sigma_r)$.

We model users' service times by a geometric distribution; i.e., at the end of each time slot a user will remain in the system with probability p_j, and leave the system with probability $1-p_j$, so the expected service time of a user is $1/p_j,\ \forall j\in[N]$.

There are three objectives the CU may want to optimize simultaneously: latency, financial cost and service quality. In order to simplify this multi-objective optimization problem the CU minimizes the long-term weighted average of these three objectives. Next we explain each of these costs.

Latency ($\delta^{(t)}_{T_{k,j}}$) *:* The latency cost associated with VNF instance $N_{k,j}$ during epoch t is due to three potential causes:

- **VNF resizing latency** $\left(\delta^{(t)}_{r_{k,j}}\right)$ is associated with resizing the containers. Resizing a VNF consists of varying the amount of allocated CPU and memory resources. Docker allows to resize containers on-the-fly by using the command `docker update` (from Docker v1.11.1). We assume that any instantiated container incurs a delay of $\delta_{r,c}$ per unit C of CPU added/removed, and $\delta_{r,m}$ per block of memory of size F added/removed. Thus, resizing latency of the VNF instance $N_{k,j}$ is:

$$\delta^{(t)}_{r_{k,j}}=|c^{(t)}_{k,j}-c^{(t-1)}_{k,j}|\delta_{r,c}+|m^{(t)}_{k,j}-m^{(t-1)}_{k,j}|\delta_{r,m}.$$

- **Deployment latency** $\left(\delta^{(t)}_{d_{k,j}}\right)$: When a new VNF $N_{k,j}$ is instantiated on a server $S_k, k\in[K]$, we consider a boot-up delay of $\delta_{d,b}$ per container. The total deployment latency of instance $N_{k,j}$ is

$$\delta^{(t)}_{d_{k,j}}=\mathbb{1}\left(c^{(t-1)}_{k,j}=0 \text{ and } c^{(t)}_{k,j}>0\right)\delta_{d,b}.$$

- **Offloading latency** $\left(\delta_{off_{K+1,j}}\right)$: If a VNF instance is deployed on the cloud, in order to keep the service running, a continuous flow of information between the cloud and the CU must be retained until the VNF is terminated. This incurs a total latency of

$$\delta_{off_{K+1,j}}=2\overline{m}_{K+1,j}B/R^{(t)}$$

for the offloaded VNF. Once a VNF is deployed in the cloud, we consider that the maximum resource utilization $\overline{r}^{(t)}_j$ is guaranteed, so that $\mathrm{QoS}^{(t)}_{K+1,j}=\mathrm{QoS}^{max}$.

The total latency incurred by VNF instance $N_{k,j}$ at epoch t is

$$\delta^{(t)}_{T_{k,j}}=u^{(t)}_{k,j}\cdot\begin{cases}\delta^{(t)}_{d_{k,j}}+\delta^{(t)}_{r_{k,j}}, \text{ if } k\in[K]\\ \delta_{off_{K+1,j}}, \text{ otherwise}\end{cases}.$$

All the users, being served by instance $N_{k,j}$ experience the same latency, and hence the scaling by $u^{(t)}_{k,j}$, the number of users being served for each instance.

Financial Cost ($C^{(t)}_{T_{k,j}}$) *:* A price model that takes into account the economic implications of each $N_{k,j}$ VNF instance configuration is developed.

- **Resource cost** $\left(C^{(t)}_{r_{k,j}}\right)$**:** is a financial cost of $C_{r,m}$ per B bits of memory per epoch and $C_{r,p}$ per C units of CPU resource per epoch for server $S_k, k\in[K]$, i.e.,

$$C^{(t)}_{r_{k,j}}=c^{(t)}_{k,j}C_{r,p}+m^{(t)}_{k,j}C_{r,m}.$$

- **Server cost** $\left(C^{(t)}_{i_{k,j}}\right)$**:** Every time a server is powered on, we consider a one-time payment of $C_{i,0}$ plus a rental cost of $C_{i,v}$ per epoch. Hence, the server cost is given by:

$$C^{(t)}_{i_{k,j}}=\mathbb{1}\left(1_N\mathbf{c}^{(t-1)}_k=0 \text{ and } 1_N\mathbf{c}^{(t)}_k>0\right)\frac{C_{i,0}}{N}+\mathbb{1}\left(1_N\mathbf{c}^{(t)}_k>0\right)\frac{C_{i,v}}{N}.$$

- **Cloud cost**$\left(C^{(t)}_{c_{K+1,j}}\right)$**:** The financial cost of offloading a VNF to the cloud is modelled as a one-time payment of $C_{c,0}$ plus a rental payment of $C_{c,v}$ per user per epoch until the VNF is terminated. Thus the cloud cost of VNF N_j is given by:

$$C^{(t)}_{c_{K+1,j}}=\mathbb{1}\left(c^{(t-1)}_{K+1,j}=0 \text{ and } c^{(t)}_{K+1,j}>0\right)C_{c,0}+\overline{m}_{K+1,j}C_{c,v}.$$

The total financial cost of VNF instance $N_{k,j}$ at epoch t is given by:

$$C^{(t)}_{r_{k,j}}=u^{(t)}_{k,j}\cdot\begin{cases}C^{(t)}_{r_{k,j}}+C^{(t)}_{i_k}, \text{ if } k\in[K]\\ C^{(t)}_{c_{K+1,j}}, \text{ otherwise}\end{cases},$$

Service Level Agreement ($SLA^{(t)}_{k,j}$) *:* Each VNF instance $N_{k,j}$, $k\in[K]$ $j\in[N]$, is associated with a minimum QoS requirement, QoS^{min}_j, and the failure to provision resources accordingly might incur service disruption, which violates the SLA. Accordingly, we define the SLA cost at VNF instance $N_{k,j}$ as

$$SLA^{(t)}_{k,j}=\left(\gamma_j\mathbb{1}\left(\mathrm{QoS}^{(t)}_{k,j}<\mathrm{QoS}^{min}_j\right)-\mathrm{QoS}^{(t)}_{k,j})\right)u^{(t)}_{k,j},$$

where $\mathrm{QoS}^{(t)}_{k,j}$ is the perceived QoS of VNF N_j at server S_k, and γ_j is the penalty for not fulfilling QoS_j. Furthermore, the SLA cost scales with the number of users being served by the VNF instance $N_{k,j}$ as all of them experience the same QoS.

We remark that, in order to capture the impact of reconfiguration of a VNF container on the whole network, each objective cost function is scaled by the number of users, such that a reconfiguration that affects more users is penalized/rewarded more than those affecting less users.

***Network Cost**:* We define the overall network cost as the total cost incurred by all the instances deployed in the network at decision epoch t, defined as:

$$C_T^{(t)} = \frac{\sum_{j\in\mathcal{N}^{(t)}}\sum_{k\in[K+1]}\omega_1\delta_{T_{k,j}^{(t)}} + \omega_3 SLA_{k,j}^{(t)} + \omega_2 C_{T_{k,j}}^{(t)}}{\sum_{j\in\mathcal{N}^{(t)}}\sum_{k\in[K+1]} u_{k,j}},$$

where $\omega_1, \omega_2, \omega_3 \in \mathbb{R}^+$ are fixed weights independent of the VNF and the server. These weights can be tuned based on the preferences of the network operator, e.g., a network operator might be more concerned about reducing the economic cost rather than providing a high quality service, etc. The normalization by the number of users is to balance the network cost between heavy and low traffic periods. Without such a normalization busy traffic periods would incur higher costs regardless of the CU's performance.

***VNF Instance Cost**:* For purposes that will be explained in Section IV, we define the VNF instance cost $C_{k,j}^{(t)}$ incurred by instance $N_{k,j}$ at epoch t, as follows:

$$C_{k,j}^{(t)} = \frac{\omega_1\delta_{T_{k,j}^{(t)}} + \omega_3 SLA_{k,j}^{(t)} + \omega_2 C_{T_{k,j}}^{(t)}}{u_{k,j}}.$$

This cost measures the contribution of a particular VNF instance to the global network cost.

III. Problem Formulation

In this section, we formulate the resource allocation problem as a MDP. We envisage an autonomous CU with the goal of minimizing the long-term cost. To this end, we define the state space and the set of actions that the CU can take at each decision epoch.

A. MDP

At each decision epoch of a MDP an agent observes a state $s^{(t)} \in \mathcal{S}$, where $\mathcal{S}$ is the state space, and selects and action $a^{(t)} \in \mathcal{A}\left(s^{(t)}\right)$, where $\mathcal{A}\left(s^{(t)}\right)$ is the set of all possible actions in state $s^{(t)}$. Set $A = \cup_{s^{(t)}\in\mathcal{S}}\mathcal{A}\left(s^{(t)}\right)$ is referred as the action space. Action $a^{(t)}$ in state $s^{(t)}$ incurs a certain cost $R\left(s^{(t)}, a^{(t)}\right)$, where $R : \mathcal{S}\times\mathcal{A} \to \mathbb{R}$ denotes the cost function, and the agent transitions to a new state $s^{(t+1)} \in \mathcal{S}$ with probability $p\left(s^{(t+1)} \mid s^{(t)}, a^{(t)}\right) \in \mathcal{P}$, where $\mathcal{P} : \mathcal{S}\times\mathcal{A}\times\mathcal{S} \to [0,1]$ is a probability kernel. At each interaction, the agent maps the observed state $s^{(t)}$ to a probability distribution over the action set $\mathcal{A}\left(s^{(t)}\right)$. This MDP model is thus characterized by the 4-tuple $\langle s, a, r, p\left(s' \mid s, a\right)\rangle$. The *policy* of the agent, denoted by π, specifies the probability of selecting action $a^{(t)} = a$ in state $s^{(t)} = s$, given by $\pi\left(a \mid s\right)$.

The *state-value function* $V_\pi\left(s\right)$ for policy π at state s is defined as the expected discounted cost the agent would accumulate starting at state s following policy π:

$$V_\pi\left(s\right) \doteq \mathbb{E}_\pi\left[\sum_{t=1}^{\infty}\gamma^{(t-1)}R\left(s^{(t)}, \pi\left(s^{(t)}\right)\right) \mid s^{(1)} = s\right],$$

where $0 \le \gamma \le 1$ is the discount factor that determines how far into the future the CU "looks", i.e., $\gamma = 0$ corresponds to a "myopic" CU, that focus only on its immediate cost, while $\gamma = 1$ represents an CU concerned with the cost over the whole time horizon. The action-value function, also referred as Q-function, is defined as:

$$Q_\pi\left(s, a\right) \doteq \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R\left(s^{(t+k)}, \pi\left(s^{(t+k)}\right)\right) \mid s^{(t)} = s, a^{(t)} = a\right].$$

We define the optimal value function, $V^*(s)$, as the minimum expected total discounted cost obtained starting in state s and following the optimal policy:

$$V^*(s) = \min_\pi \mathbb{E}_\pi\left[V_\pi\left(s\right)\right].$$

The goal is to find a policy $\pi*$ whose value function is the same as the optimal value function $V_{\pi*}\left(s\right) = V^*$.

Next, we define the state and action spaces for our problem.

B. State Space

The network state space is the set of all possible configurations of the network. The state at epoch t consists of:

1) the number of arrivals for each VNF $\left\{n_j^{(t)}\right\}_{j\in\mathcal{N}}$.
2) deployed VNFs $\mathcal{N}^{(t)}$.
3) number of users being served by VNF N_j at each server, $u_{k,j}^{(t)}$.
4) cloud link capacity, $R^{(t)}$.
5) CPU resources allocated to each VNF at each server, $\mathbf{C}^{(t)}$.
6) memory resources allocated to each VNF at each server, $\mathbf{S}^{(t)}$.

The network state at epoch t is characterized by $s^{(t)} = \left(\bigcup_{j\in\mathcal{N}} n_j^{(t)},\ \mathcal{N}^{(t)},\ \bigcup_{k\in\mathcal{K}, j\in\mathcal{N}} u_{k,j}^{(t)},\ \mathbf{C}^{(t)}, \mathbf{S}^{(t)}, R^{(t)}\right) \in \mathcal{S}$, where

$$\mathcal{S} = \left\{\left(\mathbb{Z}^+\right)^{\mathcal{N}}\right\} \times 2^{\mathcal{N}} \times \left\{\left(\mathbb{Z}^+\right)^{\mathcal{K}\times\mathcal{N}}\right\} \times \left\{\{0,\dots,\rho_{max}\}^{\mathcal{N}\times\mathcal{K}}\right\} \times \left\{\{0,\dots,\eta_{max}\}^{\mathcal{K}}\right\} \times \mathbb{R}^+.$$

C. Action Space

The CU can react to variations in the workload in three ways: *vertical scaling*, *horizontal scaling* and *offloading*. The CU actions are taken at the user level, that is, the CU selects an action for each user request arriving at the system. This allows the CU to allocate users requesting the same VNF to different servers.

Following [25], we employ a PAMDP formulation, where a discrete action set is defined as $\mathcal{A}_D = \{a_1, a_2, \dots, a_D\}$, and each action $a \in \mathcal{A}_D$ is associated with n_a continuous parameters $\left\{p_1^a, \dots, p_{n_a}^a\right\}$, $p_i^a \in \mathbb{R}$. Thus, each tuple $\left(a, p_1^a, \dots, p_{n_a}^a\right)$ represents a distinct action, and the action space is given by $\mathcal{A} = \cup_{a\in\mathcal{A}_D}\left\{a, p_1^a, \dots, p_{n_a}^a\right\}$. In our problem, the first discrete component denotes the server at which the user is assigned to, while the remaining continuous components denote how the associated resources are updated.

***Vertical Scaling**:* The vertical scaling action space, denoted by $\mathcal{A}_V = [K]$, refers to actions adding resources to, or removing from, an already deployed VNF instance at epoch t [7]. Taking into account the traffic fluctuations and VNF requirements, a CU might decide to increase (decrease) the CPU, and/or memory resources allocated to a deployed VNF instance independently, i.e., the memory allocation can be increased while the CPU allocation is decreased, or vice-versa. Hence, we define the vertical scaling actions separately for the CPU and memory resources, as $p_{CPU}^{(t)}$ and $p_M^{(t)}$, respectively, as the change in the allocated resources with respect to time slot $t-1$. We have

$$\begin{cases} p_{CPU}^{(t)} \in \left\{ i \cdot B \mid \ i \in \mathbb{R},\ -\rho_k^{(t)} \leq i \leq \rho_{max} - \rho_k^{(t)} \right\} \\ p_M^{(t)} \in \left\{ i \cdot C \mid \ i \in \mathbb{R},\ -\eta_k^{(t)} \leq i \leq \eta_{max} - \eta_k^{(t)} \right\} \end{cases}.$$

Vertical scaling is limited by the resources of the physical server in which a container is deployed, thus, the limitation of $\rho_{max} - \rho_k^{(t)}$ and $\pm\eta_{max} - \eta_k^{(t)}$.

Note that the parameters $p_{CPU}^{(t)}$ and $p_M^{(t)}$ represent increment/decrement of the resources already allocated to VNF N_j at server S_k, i.e., $c_{k,j}^{(t)} = c_{k,j}^{(t-1)} + p_{CPU}^{(t)}$ and $s_{k,j}^{(t)} = s_{k,j}^{(t-1)} + p_M^{(t)}$; hence, $p_{CPU}^{(t)}$ and $p_M^{(t)}$ can also take negative values. As mentioned before, all the users of a server's VNF instance equally share the allocated resources, thus, all of them are affected by the reshuffling of resources.

***Horizontal Scaling**:* Horizontal scaling refers to the deployment of new containers to support an existing VNF N_j at epoch t. If the load of a VNF increases, and the CU estimates that server k at epoch $t+1$ will not be able to support its operation, the CU might create another instance of the same VNF in another server.

We have $\mathcal{A}_H = [K]$ and

$$\begin{cases} p_{CPU}^{(t)} \in \left\{ i \cdot B \mid \ i \in \mathbb{R},\ -\rho_k^{(t)} \leq i \leq \rho_{max} - \rho_k^{(t)} \right\} \\ p_M^{(t)} \in \left\{ i \cdot C \mid \ i \in \mathbb{R},\ -\eta_k^{(t)} \leq i \leq \eta_{max} - \eta_k^{(t)} \right\} \end{cases},$$

where k denotes the server at which a new VNF instance is to be deployed using horizontal scaling. $p_{CPU}^{(t)}$ and $p_M^{(t)}$ account for the amount of CPU and memory resources to be allocated for the new deployment of $N_{k,j}$ at server k.

***Work offloading**:* If the CU foresees that it cannot cope with a traffic fluctuation by scaling vertically or horizontally, it can decide to offload a VNF to the cloud. We define the offloading action as $\mathcal{A}_{off}^{(t)}$. This action is not associated with any parameter due to the assumption of unlimited CPU and memory resources at the cloud.

***Parametrized Action Space**:* Following the PAMDP notation the complete parameterized action space at epoch t is given by

$$\mathcal{A}^{(t)} \triangleq \left(\mathcal{A}_V^{(t)}, p_{CPU}^{(t)}, p_M^{(t)}, \right) \cup \left(\mathcal{A}_H^{(t)}, p_{CPU}^{(t)}, p_M^{(t)} \right) \cup \left\{ \mathcal{A}_{off}^{(t)} \right\}.$$

The cost function for our problem has been defined in Section II in detail. Note that the CU's action at each time slot consists of $n_j(t)$ actions in the PAMDP formulation, one for each user request. Note that the randomness in our problem is due to random user arrivals for each VNF, and the random service time for each user in the system. If these statistics are known, the optimal policy can be identified through dynamic programming (DP), e.g., by the value iteration algorithm. However, estimating these probabilities for our problem, which has large state and action spaces is prohibitive, making the DP solution practically infeasible. Hence, we will instead exploit DRL to find an approximation to the optimal value function.

IV. Reinforcement Learning For VNF Management

In the proposed RL method, the agent does not necessarily exploit (or even know) the transition probabilities governing the underlying MDP as it learns directly a policy as well as the action-value functions based on its past experience (model-free). The formulated problem suffers from the curse of dimensionality due to the prohibitively large size of the state and action spaces (continuous state space). Therefore, we employ the actor-critic method with NNs as a function approximator to parameterize the policy, which allows the learning agent to directly search over the action space. Another DNN is employed to approximate the state-value functions, which are used as feedback to determine how good the current policy is.

A. Deep Reinforcement Learning (DRL)

The use of DNNs as general function approximators have been proven to work very well in a wide range of areas, such as computer vision, speech recognition, natural language processing, and more recently, wireless networks [**?**]. Traditional RL methods struggle to address real-world problems due to their high complexity. In these problems, high-dimensional state spaces need to be managed in order to obtain a model that can generalize past experiences to new states. For example, tabular Q-learning uses a hash table to store the estimated cost of state-action pairs, so for continuous input states, even if quantized, this solution deems intractable, since even with modest 5-level quantization and a state vector of size N, 5^N entries would have to be stored ($\approx 10^{13}$ entries if $N = 20$). DRL aims to solve this problem by employing NNs as function approximators to reduce the complexity of classical RL methods.

In [27] authors introduce deep Q-learning network (DQN), where a DNN is used as a function approximator for action selection on a discrete action space, based on Q-learning. Given a state, Q-learning updates the action-value estimate with the immediate reward plus a weighted version of the highest Q-estimate for the next state. Using a combination of 3 convolutional layers (for computer vision) and two fully connected layers (Q-learning part), they obtain human-level results for a wide range of Atari games. Other architectures based on DQN have also been proposed, such as the duelling DQN [28], which has two distinct DNNs, one to estimate $V_\pi(s)$, and the other to estimate the so-called *advantage* function $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$. These methods work well for a continuous state space but are limited to discrete action space, suffering from the curse of dimensionality when the action space is large.

To overcome the limitation of discrete action selection, in [15] the idea of DQN is extended to continuous action spaces using the deterministic policy gradient (DPG) theorem [29], in particular the deep-DPG (DDPG) method. DDPG extends the use of DNN to the actor-critic method leveraging off-policy learning, where a deterministic policy is learned using a combination of *replay buffer* and *target networks* to ensure stability and a zero-mean Gaussian noise is added to the actions for action space exploration.

In [16] it is shown that DDPG may lead to overestimation of action values, thereby to suboptimal policies. To overcome this problem, authors present a novel method called twin delayed DDPG (TD3). A novel actor-critic architecture is proposed which comprises two critic networks, hence, two different Q-functions are learned, and the smaller of the two estimates is used as the update rule for the critics. The proposed algorithm adds clipped noise to the target action to make it harder for the policy to exploit Q-function errors. They also propose to update the targets and the policy less frequently than the Q-functions, helping to reduce the variance.

Given the high dimensionality of both the state and the action spaces in our model we propose a solution that leverages DNNs as policy and action-value function approximators, while exploiting the results from [16] for continuous action selection. To this end, we implement a novel architecture for PAMDP to address the problem defined in Section III.

V. Actor-Critic method in PAMDP

The actor-critic method is leveraged is a combination of value based and policy optimization approaches. It combines the benefits of both methods as the critic estimates the action-value function $Q_\phi(s,a)$, while the actor derives a policy $\pi_\theta(s)$ critically using the value estimates of the critic to update the policy. In this section we present our novel algorithm (see Algorithm 1), which implements the actor-critic method for a PAMDP, which we call the parameterized action twin (PAT). For ease of notation in the rest of the section we use $s^{(t)} = s$, $s^{(t+1)} = s'$, $a^{(t)} = a$, $R\left(s^{(t)}, a^{(t)}\right) = r$.

A. VNF MANO meets PAT

Before detailing the proposed PAT algorithm, we clarify its integration into the CU, and how it interacts with the environment described in Section II, as we believe it will ease the comprehension of the algorithm. At the beginning of each decision epoch t, we randomly select a VNF and proceed to serve new demands for this VNF. The random selection of VNF is motivated by fairness, so that we avoid starting the process of resource allocation (when more resources are available) with always the same VNF. Following the random VNF selection, we iterate over all requests of this VNF to allocate the network resources using the PAT method. For resource allocation, a snapshot of the network state is used as input to the PAT method. Based on the network state, the proposed RL algorithm decides the actions to be taken and from which server/cloud the user is served. After the allocation, a new snapshot of the network state is obtained and the agent cost described in Section V-C is computed. These transitions are stored in the memory buffer of the agent and will later be used to train the PAT algorithm, so that it can adapt to previously seen as well as new traffic patterns. Even if a VNF does not have any new requests, we nevertheless select that VNF and apply the PAT algorithm for action selection so that, in case it was already deployed, resources can be added or removed; if not, the VNF can be deployed ahead of future traffic. In this case, the VNF instance only incurs an economical cost as no user is served.

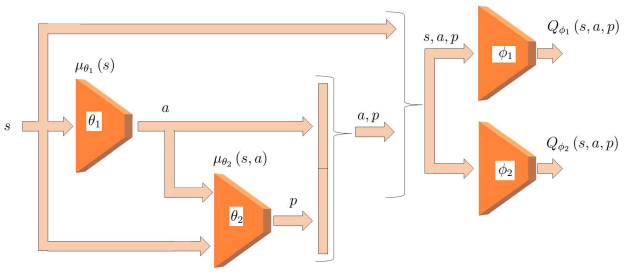


Fig. 2: The information flow between different DNNs in the proposed architecture.

B. PAT algorithm

Following [16], we use two critics in order to obtain two distinct estimates of the action-values; thus, two different DNNs, parameterized by $\boldsymbol{\phi}_1$ and $\boldsymbol{\phi}_2$, are used to estimate two different action-value functions. The aim of the two critic networks, as explained in [16], is to avoid overestimation. We find that clipping the critics' updates to the minimum between the two estimates yields better policies. Note that this update rule might introduce underestimation bias; however, we find it more convenient in the long term to avoid convergence to suboptimal policies.

Two more DNNs, parameterized by $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, are used for the policy parameterization of the actor. The goal of the first actor network is to select the discrete action a based on the current state s, while the second network generates the continuous action parameters $p = [p^{(t)}_{CPU}, p^{(t)}_{M}]^T$ based on the outcome of the first actor network a and the current system state s. Thus, the joint selection (a,p) determined by two distinct networks, in contrast to the approach in [17], where a single DNN architecture is used to determine both, the action and the parameters associated with it. We find this architecture to reflect a more natural process of action selection by first deciding the discrete action a, and then, choosing the associated parameters p as defined in Subsection III-C. We use a stochastic policy for discrete action selection while deterministic policy is leveraged for parameter selection, which we denote by μ, i.e., parameters $\boldsymbol{\theta}_2$ map state and action (s,a) to parameters $\mu_{\theta_2}(s,a) = p$. Figure 2 illustrates the DNN structure and the flow of information.

Finally, four more DNNs are employed, corresponding to the mirroring target networks, and are parametrized by $\boldsymbol{\phi}_1^-, \boldsymbol{\phi}_2^-, \boldsymbol{\theta}_1^-, \boldsymbol{\theta}_2^-$, respectively. Their function is explained later in this section.

1) Parameter Updates: The critics take the network state s and the action (a, p), and estimate the value function $Q_{\phi_i}(s, a, p)$, $i = 1, 2$. As is typical in actor-critic methods, we use off-policy temporal difference of 0, i.e., TD(0), for action-value function approximation, with clipped update rules:

$$Q_{\phi_i}^{t+1}(s,a,p) = Q_{\phi_i}^{t}(s,a,p) + \alpha\left(r + \gamma \min_{i=1,2}\{ Q_{\phi_{i^-}}^{t}\left(s', \mu_{\theta_1^-}(s'), \mu_{\theta_2^-}\left(s', \mu_{\theta_{1^-}^-}(s')\right)\right)\right\} - Q_{\phi_i}^{t}(s,a,p)\Big),$$

which minimizes the following loss function, for $i = 1, 2$:

$$L_{Q_{\phi_i}}(s,a,p) = \frac{1}{2}\left(r + \gamma \min_{i=1,2}\left\{Q_{\phi_i^-}\left(s', \mu_{\theta_1^-}(s'), \mu_{\theta_2^-}\left(s', \mu_{\theta_{1^-}^-}(s')\right)\right)\right\} - Q_{\phi_i}(s,a,p)\right)^2.$$

The action-value functions of the critics are learned through gradient descent with the update rule:

$$\phi_i^{t+1} = \phi_i^t + \alpha\left(r + \gamma \min_{i=1,2}\left\{Q_{\phi_i^-}\left(s', \mu_{\theta_1^-}(s'), \mu_{\theta_2^-}\left(s', \mu_{\theta_{1^-}^-}(s')\right)\right)\right\} - Q_{\phi_i^t}(s,a,p)\right)\nabla_{\phi_i^t} Q_{\phi_i^t}(s,a,p).$$

The critics estimations of the joint action and parameters are gathered by the actors to update the policy. In continuous action space, the greedy update of the policy becomes infeasible as it requires a global maximization at every step, and going through all the action space to maximize the estimated expected return is infeasible. Following [29], we use the critic's network's gradient that indicates the direction the global Q-value estimate increases, to update the policy parameters. In order to obtain the gradients, we need to perform back-propagation through one of the critics networks (we chose critic 1). It must be noted here that this gradient is not the conventional gradient with respect to the network parameters, but with respect to the input, such that for the action network θ_1 the update rule is

$$\theta_1^{t+1} = \theta_1^t + \alpha\, \mathbb{E}_{s\sim\rho_{\theta_1}}\left[\nabla_{\theta_1}\mu_{\theta_2}(s)\, \nabla_a Q_{\phi_1}(s,a,p)\big|_{a=\mu_{\theta_1}(s)}\right],$$

while the update rule for network θ_2 is

$$\theta_2^{t+1} = \theta_2^t + \alpha\, \mathbb{E}_{s\sim\rho_{\theta_2}}\left[\nabla_{\theta_2}\mu_{\theta_2}(s,a)\, \nabla_p Q_{\phi_1}(s,a,p)\big|_{p=\mu_{\theta_2}(s,a)}\right],$$

where $s \sim \rho_{\theta_i}$ refers to the trajectory sample using network i.

2) Stabilizing updates: Once both the critic and the actor networks are updated, the target networks should also be updated. Target networks are used to stabilize the updates. If the same $\phi_1 = \phi_2 = \phi$ network is used for bootstrapping (i.e., estimating the value function of the next state $Q(s', a')$) and estimating $Q(s, a)$ in (V-B1), the ϕ network will be updated with each iteration to move closer to the target Q-values; but, at the same time, the target Q-values, which are given by the same network, will also be changing in the same direction, like a dog chasing its tail. By introducing the target networks, we reduce this constant movement of the target estimates by delaying its update. The target networks are updated as

$$\begin{cases}\phi_i^- = \tau\phi_i^t + (1-\tau)\phi_i^- \\ \theta_i^- = \tau\theta_i^t + (1-\tau)\theta_i^-\end{cases},$$

where $\tau \leq 1$ is an hyper-parameter to regulate the update speed.

Algorithm 1 Proposed PAT

Initialize the actors and critics networks, i.e., θ_1, θ_2 and ϕ_1, ϕ_2 using Gaussian initialization with $\mu = 0, \sigma = 10^{-2}$.
Copy the parameters to the target networks, i.e., $\phi_1^- \leftarrow \phi_1, \phi_2^- \leftarrow \phi_2, \theta_1^- \leftarrow \theta_1, \theta_2^- \leftarrow \theta_2$
Initialize the replay buffer $\mathcal{M}$
$t = 0$
while $t < total\_timesteps$ **do**
  for i= 0,...,T **do**
    Observe state s.
    Select action $a = \mu_{\theta_2}(s)$ with probability $1 - \epsilon$ or a random action a with probability ϵ.
    Select

$$p = \text{clip}\left(\mu_{\theta_2}(s,a) + w, p_{min}, p_{max}\right),$$

    where $w \sim \text{clip}\left(\mathcal{N}\left(0, \sigma^2\right)\cdot(p_{max}, \eta_{max}), -c, c\right)$
    Store transaction $\langle s, (a,p), r, s'\rangle \in \mathcal{M}$
  end for
  Get a batch $\mathcal{B} = \{(s, (a,p), r, s')\}$ of randomly sampled trajectories from the replay buffer $\mathcal{M}$
  Compute the action and parameter targets:

$$a^-(s') = \mu_{\theta_1^-}(s)$$

$$p^-(s', a^-) = \text{clip}\left(\mu_{\theta_2}\left(s', a^-(s')\right) + w, p_{min}, p_{max}\right)$$

  Compute the target estimates

$$y(r, s') = r + \gamma \min_{i=1,2}\left\{Q_{\phi_i^-}\left(s', a^-(s'), p^-\left(s', a^-(s')\right)\right)\right\}$$

  Update Q-functions:

$$\nabla_{\phi_i}\frac{1}{|\mathcal{B}|}\sum_{\rho\in\mathcal{B}}\left(Q_{\phi_i}(s,a,p) - y(r,s')\right)^2 \text{ for } i = 1, 2$$

  Update action policy:

$$\nabla_{\theta_1}\frac{1}{|\mathcal{B}|}\sum_{(s,p)\in\mathcal{B}} Q_{\phi_1}\left(s, \mu_{\theta_1}(s), p\right)$$

  Update parameter policy:

$$\nabla_{\theta_2}\frac{1}{|\mathcal{B}|}\sum_{(s,a)\in\mathcal{B}} Q_{\phi_1}\left(s, a, \mu_{\theta_2}(s,a)\right)$$

  Update target networks:

$$\phi_i^- = \tau\phi_i + (1-\tau)\phi_i^- \text{ for } i = 1, 2$$
$$\theta^- = \tau\theta_i + (1-\tau)\theta_i^- \text{ for } i = 1, 2$$

  $t = t + 1$
end while

Another tool to stabilize the network parameter updates is the memory buffer $\mathcal{M}$, which stores the interactions of the agent with the environment, to be more precise, we store on-step trajectories, i.e., s, a, r, s'. Once the memory is filled with enough samples ($\approx 100K$), we uniformly sample the memory to obtain mini-batches of N samples, which are used to compute the losses of the actor and critic. Most optimization algorithms, including gradient descent, assume that the samples, from which the gradient estimate is obtained, are i.i.d. Clearly this is not the case in the defined environment; however, by sampling uniformly from the memory buffer the correlation between consecutive samples is reduced, leading to a more stable optimization of the action-parameter selection.

3) Exploitation vs. exploration: Any RL algorithm with a deterministic policy entails the trade-off between exploitation and exploration. For discrete action selection, we use the $\epsilon - greedy$ policy to ensure exploration, where with probability ϵ a random action is selected by sampling a uniform random

distribution over all possible discrete actions. A high value for ϵ is set at the beginning to encourage exploration, but its value is reduced gradually over time until it reaches a certain minimum ϵ_{min}, where it remains stable.

Ensuring the exploration of all possible continuous parameters is not possible. We use the approach proposed in [16], where a clipped zero-mean Gaussian noise is constantly added to the parameter selection policy (see Eqn. (V-B3)). This approach is motivated by the assumption that similar parameters should have similar costs, and thus, similar estimates; and the noise addition is used to encourage exploration. After the addition of noise the parameter values are clipped to the allowed range $[p_{min}, p_{max}]$, as defined in Section III-C:

$$\mu'(s,a) = \text{clip}\left(\mu_{\theta_2}(s,a) + w, p_{min}, p_{max}\right),$$
$$w \sim \text{clip}\left(\mathcal{N}\left(0,\sigma^2\right), -c, c\right),$$

where σ and c are hyperparameters. Similarly to the ϵ parameter for the action selection, we gradually reduce the value of c, until it reaches a minimum value c_{min}.

4) Architecture: The DNN architectures for the action, action parameter, and critic networks are the same. For all the networks, the inputs are processed by three fully connected layers consisting of 128 and 64 units, respectively. Each fully connected layer is followed by a rectified linear unit (ReLU) activation function with negative slope 10^{-2}. The weights of the fully connected layers are initialized using Xavier initialization with a standard deviation of 10^{-2} [30].

The input of the actor action network is the network state, and connected to its final inner product layer there are $K+1$ linear outputs corresponding to the discrete action selection (K servers plus the cloud). For the actor parameter network, the last layer comprises an hyperbolic tangent activation function scaled by p_{max} and η_{max} with two outputs, corresponding to the CPU and memory values allocated to the discrete action selected, while its inputs are the state and the selected action. Finally the critic network gathers the state, the action, and the action parameters, and a single output value is obtained, the estimate of $Q(s,a,p)$. We use ADAM optimizer for both the actor and the critic, with a learning rate of l_r.

C. Agent Cost Function

In Section II, and more precisely in Eqn. (II), we defined the global network cost as the main metric this work aims to minimize. However, we do not directly use (II) as the metric the agent optimizes, as we found it to be too general to guide the agent in its initial learning steps towards finding resource allocation policies that lead to good results. The goal of this subsection is to define the cost function $\Psi^{(t)}$ that we use in learning to provide feedback to the agent regarding its actions.

Individual actions taken by the agent have direct impact on the performance of the selected VNF instance, but also contribute to the total network performance. Thus, in order to guide the agent to learn to allocate resources for different VNF instances, we use the VNF instance cost of (II). However, the minimization goal of this work is the total network cost, hence, we need to include it in the global picture. To this end we define the cost as follows:

$$\Psi^{(t)} = \frac{C_{k,j}^{(t)} + \beta C_T^{(t)}}{\Gamma_{max}},$$

where Γ_{max} is a hyperparameter that guarantees $\Psi^{(t)} \in [-1, 1]$, while β determines how much the agent accounts for the total network cost. Eqn. (V-C) is what we use in the DNN training, while Eqn. ((II)) is the objective function of the CU.

Furthermore, during the training phase, the proposed RL approach needs to learn the physics of the environment, that is, at the beginning of the learning process the agent might try to add/subtract more CPU or memory to a server than the one that is available/possible. In order to teach the agent the environment's physical limitations, whenever the algorithm outputs an infeasible action we offload the user to the cloud, and impose a cost of $\Psi^{(t)} = -1$.

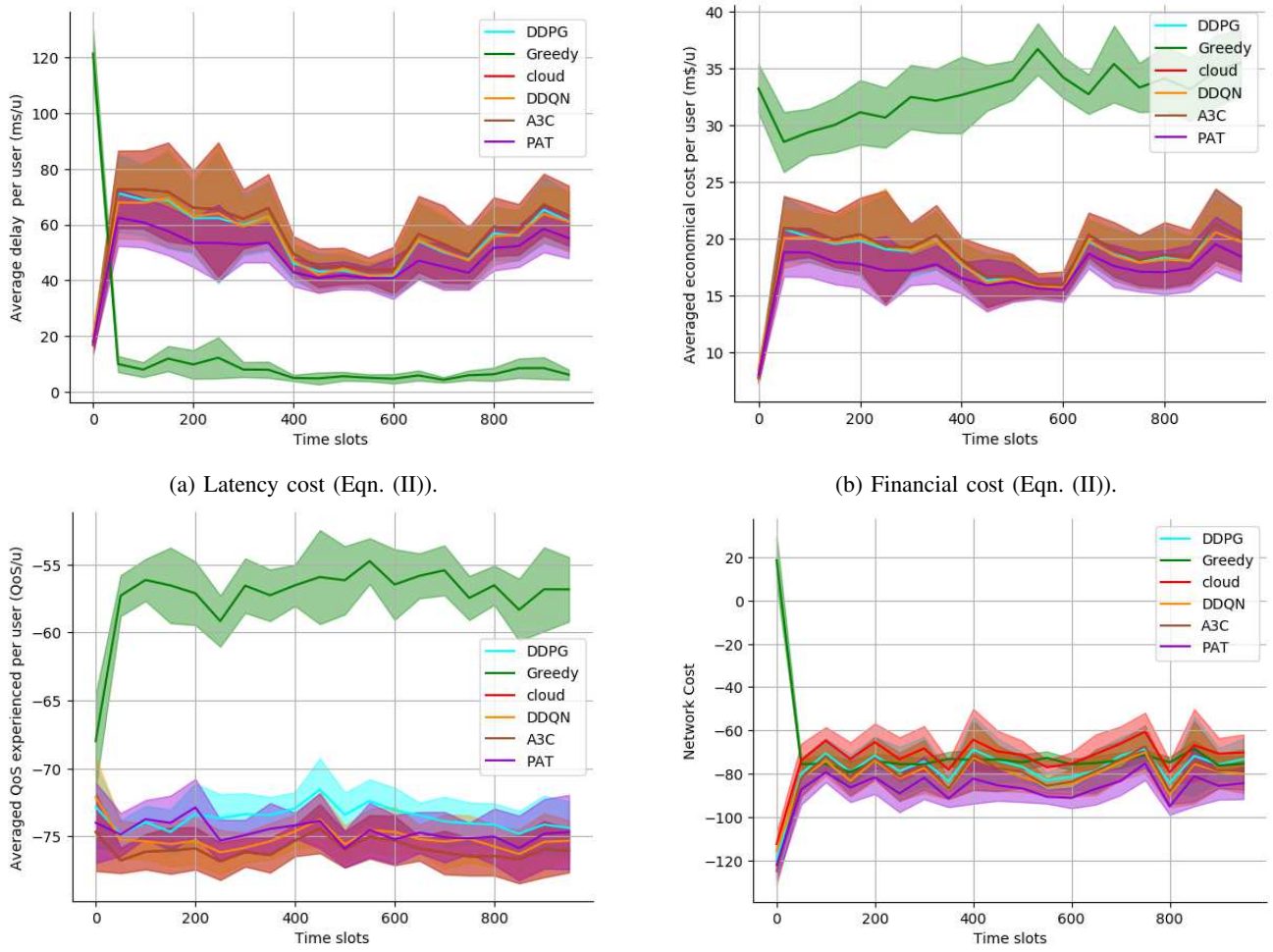
VI. Numerical Results

In this section we present numerical results obtained with the PAT method described in Section V. We start by presenting some benchmarks to compare the proposed algorithm with, followed by the experimental setup and the parameters used in the simulations.

A. Benchmarks

In order to assess the quality of the proposed algorithm, we compare the PAT agent with other DRL benchmark algorithms.

- Greedy: For each new user in the system, the *greedy* algorithm checks whether the new user's VNF is already deployed in one of the CU servers. If so, computes the CPU and memory that the server would need to allocate to that VNF such that the new and existing users can be served from that VNF instance, i.e., the resulting VNF's QoS lies within $[QoS^{min}, QoS^{max}]$. If this is feasible, the VNF is resized, and the user is allocated to that server (vertical scaling). If not, another server is checked until resources for the new user can be assigned (horizontal scaling). If no server is able to allocate this new user, it is offloaded to the cloud.
- Cloud: This policy offloads all the traffic to the cloud.
- **DRL benchmarks**: To overcome the problem of discrete and continuous action selection formulated in this work, we use two distinct state of the art DRL algorithms. For server selection we use DDQN, while for parameter selection we use the following algorithms:
  1) DDPG [29] with an hyperbolic tangent activation function in the outer layer scaled by the maximum values of the CPU and memory, respectively.
  2) A3C [31], where the output of the DNNs provide the mean and variance values of the Gaussian distributions used to sample the values of the CPU and memory. The parameter T of [31] is chosen to be 128.
  3) DDQN [27], where we discretize the CPU and memory action spaces, with a resolution of 5, meaning that the total number of actions is given by $\eta_{max}/5 \times \rho_{max}/5$.



(a) Latency cost (Eqn. (II)).

(b) Financial cost (Eqn. (II)).

(c) SLA cost (Eqn. (II)).

(d) Network cost (Eqn. (II)).

Fig. 3: Defined costs comparison between PAT and the other DRL benchmarks. The shaded regions demonstrate the standard deviation of the average evaluation over 10 trials.

The DDQN for discrete action selection, and the previous set of algorithms are trained recursively (discrete action network training first, followed by parameter network training) for 1000 times. Each algorithm interacts with the environment for 10000 time slots.

B. Parameters

We consider $N = 10$ VNFs with features shown in Table I. We consider a CU with $K = 10$ servers each with a CPU capability of $50C$ Hz ($\rho_{max} = 50$) and memory capacity of 50 B bits ($\eta_{max} = 50$). The arrival rates ($\lambda_j^{(t)}$) for different VNFs at each epoch are sampled from a normal distribution, where each of the VNFs is characterized by different mean and variance values, listed in Table I. The values of the other parameters involved in the calculation of the cost function and those used to reinforce the actor behaviours are given in Table II. The PAT algorithm parameters are collected in Table III. We note that the values presented in Tables I, II and III for the numerical simulations are chosen as reasonable values that would lead to a solution with a balanced allocation of available resources in the servers and the cloud. Naturally, the value of these parameters in practice depends highly on the implementation and the technology used (memory/CPU capability) as well as the VNFs being considered; however, our problem formulation is general, and we have reached similar observations with a large variety of parameter values considered.

C. PAT performance

The proposed PAT algorithm is run using 10 different seeds, and the average learning curves are depicted in Figure 4. Given the cost function in (II), it can be seen that the agent maximizes the SLA to the point where the cost function is negative, meaning that the perceived QoS is greater than the weighted combination of the latency and the economical costs. Thus, given the predefined cost function, the agent learns to minimize the cost and to utilize the servers in an online manner. That is, since the agent generates its own dataset on the fly, by interacting with the environment, variations in the environment are directly fedback into the model by adding new

TABLE I: VNF resource requirements.

VNF	CPU			Memory			QoS		Others	Mean	Variance
	c_0	c_r	c_d	s_0	s_r	s_d	QoS_{min}	QoS_{max}	γ_j	μ_j	σ_j
N_1	3	5	4	6	5	3	35	70	2	2	1.5
N_2	2	3	2	4	4	2	36	80	2	2.5	0.2
N_3	1	4	2	2	3	2	27	63	2	4	0.5
N_4	1	4	3	1	3	1	40	90	2	1	1
N_5	2	6	2	3	4	3	20	100	2	2.5	1
N_6	1	2	1	0	3	2	5	30	2	2	1.5
N_7	2	3	2	2	5	3	56	80	2	5	1
N_8	3	4	2	3	6	5	20	53	2	2	1
N_9	1	4	3	4	4	2	40	90	2	3	0.5
N_{10}	2	6	2	3	4	3	20	100	2	2	1

TABLE II: Delay parameters.

Delay (ms)	$\delta_{r,c} = 3$	$\delta_{r,s} = 4$	$\delta_{d,b} = 20$	$\delta_{d,t} = 10$		
Cost (m\$)	$C_{r,s} = 3$	$C_{r,p} = 6$	$C_{i,o} = 2$	$C_{i,v} = 1$	$C_{c,0} = 1$	$C_{c,v} = 3$

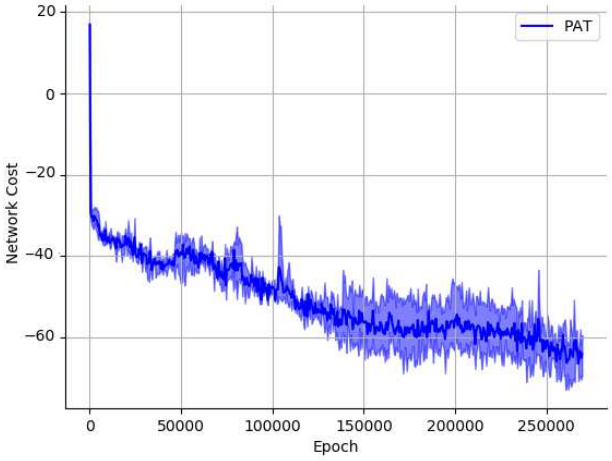


Fig. 4: Evolution of the network cost in Eqn. (II).

traces into the memory buffer. Therefore, when a statistically significant change occurs, this is captured by the model.

Discount factor γ = 0.99	Target update $\tau = 5 \cdot 10^{-3}$
ϵ = 0.8	Learning rate $l_r = 10^{-3}$
ϵ_{min} = 0.05	Policy noise σ = 0.2
$\epsilon_{decay} = 10^{-3}$	c = 0.5
$(\omega_1, \omega_2, \omega_3)$= $(1, 1, 2)$	c_{min} = 0.1
$R_{min} = 1$	$\beta = 0.2$
$t_{max} = 100$	$\Gamma_{max} = 100$

TABLE III: PAT parameters.

D. Mapping to KPIs

We now define three KPIs for MANO in 5G networks, and map the results obtained with the PAT algorithm to these KPIs. The following KPIs are of interest for future 5G networks [32]:

- Resource utilisation efficiency: Given the CPU and memory resources, resource utilisation efficiency is defined as the ratio of utilized resources with respect the total available resources for the execution of a VNF, for a particular number of users. With the elastic functions employed in our model, the system should achieve a higher resource utilisation efficiency, since it can shelter a much larger number of users over the same physical infrastructure.
- Cost efficiency gain: This metric captures the average cost of deploying and maintaining the network infrastructure to provide the required service to its users. Given the elastic nature of the VNFs deployed, the CU should be able to optimally dimension the network such that less resources are required to support the same services; in addition, the elastic system should avoid the usage of unnecessary resources.

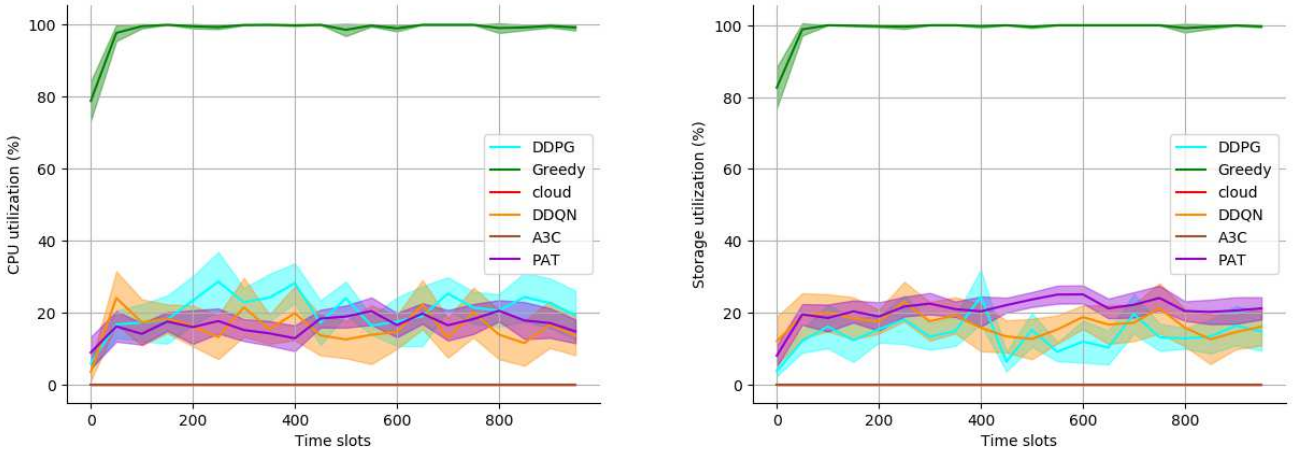
E. PAT evaluation

In Figure 3 and 5, we present the results of the proposed PAT approach and that of the other benchmark algorithms. The comparison is carried out under exactly the same traffic patterns, i.e., same arrival and departure times.

From Figures 3(a), 3(b) and 6 it can be observed that, for the particular set of parameters chosen, the users offloaded to the cloud experience higher delays than those served by the CU. On the contrary, for the financial cost, VNFs instantiated in the CU have a higher cost than those instantiated in the cloud. That explains why the scheme with lowest cloud utilization, i.e., greedy, has the lowest delay but the highest financial cost per user.

Furthermore, Figure 6 shows how all the DRL algorithms decide to allocate more users to the cloud than the CU, this is mainly for two reasons affecting the learning process:

1) Cloud offloading does not carry any penalty. Contrary to VNF allocation in the CU, where algorithms are penalised if the physical limitations of the servers are not respected, or if the allocated resources are not enough to fulfill the SLA, the deployment of VNFs at the CU carry a positive reward.



(a) CPU utilization.

(b) Memory utilization.

Fig. 5: Servers resources utilization.

2) QoS drives the learning experience. Since ω_3 is greater than the other two weights in Eqns. (II) and (II), qalgorithms aim to maximize SLA cost (given the RL reward function of (V-C)). As allocating VNF to CU may lead to lower QoS if the CU fails to provide the maximum demanded resources, the DRL tends to use the cloud where QoS^{max} is guaranteed.

1) Resource utilisation efficiency: Figure 5 shows that the PAT algorithm leads to a more efficient usage of the CPU and memory resources compared to A3C, DDPG and DDQN, as for a similar CPU and memory utilization (Figures 5(a) and 5(b)) less traffic is offloaded to the cloud (Figure 6). The efficient usage of resources accomplished by PAT is also visible in Figures 3(a) and 3(b). Even though similar resources are utilized by the the other DRL algorithms, the latency cost and the financial cost achieved by the PAT are lower (on average). The greedy approach aims to allocate as many users as possible to the CU, that is why its CU resources are fully utilised most of the time and the average delay of the users is the lowest, while its financial cost is the highest.

2) Cost efficiency gain: The comparison between the average economical cost of the PAT deployment and the other schemes is presented in Figure 3(b), where a gain in economical cost by the PAT algorithm is clearly visible. The economical cost difference between the PAT and the greedy is straightforward, as the network configuration entails a higher cost for the use of CU resources. The reduction of financial cost of the PAT compared to the other DRL algorithms is because the latter allocate more CPU and memory than needed, given the fact that most of their traffic is directed to the cloud, and the CU resources are underutilized, incurring higher cost per user.

3) Network Cost: Figure 3(d) shows that the proposed PAT algorithm outperforms the other approaches on the main metric of this work, the network cost defined in Eqn. II. The PAT finds a middle point, between directing traffic to the CU and offloading it to the cloud, and the optimization of the server selection and resource allocation are done jointly.

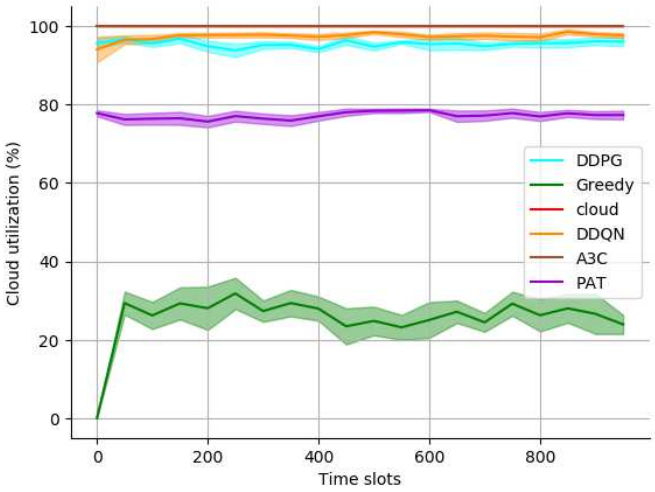


Fig. 6: Cloud utilization.

Contrary to A3C, DDQN and DDPG, where the training is done iteratively, the PAT algorithm propagates the gradient of the value-estimates obtained by the critics to the parameter network and the server selection network at the same time, pushing both networks to more optimal points simultaneously, improving the efficiency of each training update.

It is somewhat surprising that the greedy algorithm performs comparably, or sometimes even better than the baseline DRL algorithms at some time periods. This is because the DRL algorithms, in contrast to PAT, have delays in adapting to the randomness of the environment, or tend to slightly overprovision resources to be able to cope with highly time-variant traffic demands. If, however, we keep the traffic statistics (arrival rates) constant, we can see in Figure 7 that baseline DRL algorithms outperform greedy, while PAT is still the best performing algorithm. This shows that PAT not only outperforms other baselines in exploiting the resources in the most efficient manner in a static environment, but also is the fastest in terms of adapting to variations in the environment.

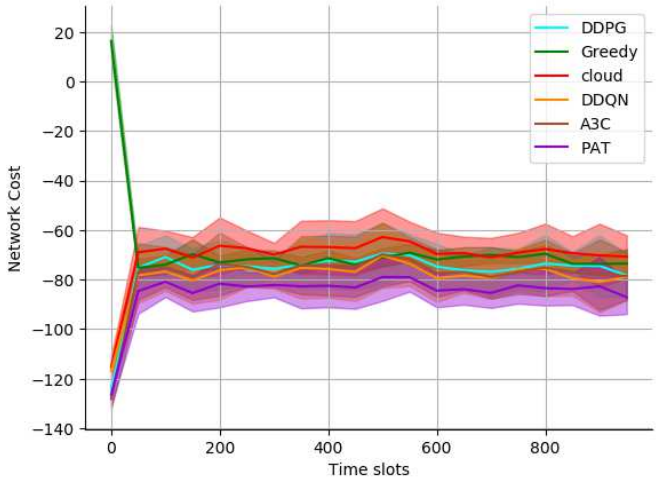


Fig. 7: Network cost with constants λ_j.

VII. Conclusions and Future Work

We presented a novel DRL algorithm for autonomous MANO of VNFs, where the CU learns to re-configure resources (CPU and memory), to deploy new VNF instances, or to offload VNFs to a central cloud. We formulated the corresponding stochastic resource allocation problem as a MDP. Then, we proposed a DRL-based solution for this MANO problem, presenting a novel algorithm named PAT, which leverages the actor-critic method to learn to provision network resources in an online manner, given the current network state and the requirements of the deployed VNFs. The novel architecture implements two critics for action value function estimation (twin), and two actor networks are used to determine the action and the parameter. A deterministic policy is implemented for both action and parameter selection. We have shown that the proposed solution outperforms all benchmark DRL schemes as well as heuristic greedy allocation in a variety of network scenarios, including static traffic arrivals as well as highly time-varying traffic settings. To the best of our knowledge, this is the first work that considers DRL for network MANO of VNFs.

As future research directions, we consider addressing the MANO of VNF chains. The problem addressed in this work does not take into account the likely relationship between different VNFs to form VNF chains, where NFs may have a temporal ordering in which they are requested by users. This factor highly increases the complexity of resource allocation as the overall user experience might be affected by a subtle VNF resource modification.

References

[1] C. Liang and F. R. Yu, "Wireless network virtualization: A survey, some research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 358–380, 2015.

[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[3] M. M. Murthy, H. Sanjay, and J. Anand, "Threshold based auto scaling of virtual machines in cloud environment," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2014, pp. 247–256.

[4] J. M.-A. Tania Lorido-Botran 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.

[5] A. W. Services. (2016) Fleet management made easy with auto scaling. [Online]. Available: https://aws.amazon.com/blogs/compute/fleet-management-made-easy-with-auto-scaling/

[6] M. Azure. (2018) Azure resource manager overview. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview

[7] S. Dutta, , S. Gera, , A. Verma, and B. Viswanathan, "Smartscale: Automatic application scaling in enterprise clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 221–228.

[8] L. Yazdanov and C. Fetzer, "Vscaler: Autonomic virtual machine scaling," in *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, 2013, pp. 212–219.

[9] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet of Things Journal*, 2018.

[10] P. Tang, F. Li, W. Zhou, W. Hu, and L. Yang, "Efficient auto-scaling approach in the telco cloud using self-learning algorithm," in *Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[11] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, "Auto-scaling network resources using machine learning to improve qos and reduce cost," *arXiv preprint arXiv:1808.02975*, 2018.

[12] R. Li, Z. Zhao, Q. Sun, I. Chih-Lin, C. Yang, X. Chen, M. Zhao, and H. Zhang, "Deep reinforcement learning for resource management in network slicing," *IEEE Access*, vol. 6, pp. 74 429–74 441, 2018.

[13] X. Chen, Z. Zhao, C. Wu, M. Bennis, H. Liu, Y. Ji, and H. Zhang, "Multi-tenant cross-slice resource orchestration: A deep reinforcement learning approach," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 10, pp. 2377–2392, 2019.

[14] C. Qi, Y. Hua, R. Li, Z. Zhao, and H. Zhang, "Deep reinforcement learning with discrete normalized advantage functions for resource management in network slicing," *IEEE Communications Letters*, vol. 23, no. 8, pp. 1337–1341, Aug 2019.

[15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[16] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *preprint arXiv:1802.09477*, 2018.

[17] M. Hausknecht and P. Stone, "Deep reinforcement learning in parameterized action space," *International Conference on Learning Representations (ICLR)*, 2016.

[18] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*. ACM, 2016, p. 1.

[19] Docker. (2018) What is a container. [Online]. Available: https://www.docker.com/resources/what-container

[20] R. Morabito, "Power consumption of virtualization technologies: An empirical investigation," in *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. IEEE, 2015, pp. 522–527.

[21] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A framework and algorithm for energy efficient container consolidation in cloud data centers," in *IEEE Int'l Conf. on Data Science and Data Intensive Sys. (DSDIS),*.

[22] R. D. Sumit Maheshwari, Saurabh Deochake and A. Grover, "Comparative study of virtual machines and containers for devops developers instructor: Prof. richard martin," *arXiv preprint arXiv:1808.08192*, 2018.

[23] R. Sutton, , and A. G. Barto, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 135.

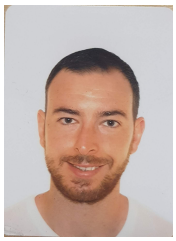
[24] D. Gutierrez-Estevez et al., "The path towards resource elasticity for 5g network architecture," in *IEEE Wireless Comm. and Netw. Conf.(WCNCW),*, 2018, pp. 214–219.

[25] K. Narasimhan, T. D. Kulkarni, and R. Barzilay, "Language understanding for textbased games using deep reinforcement learning," in *In Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Citeseer, 2015.

[26] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Communications Surveys & Tutorials*, 2019.

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

[28] Z. Wang, T. Schaul, M. Hessel, H. V. Hasselt, M. Lanctot, and N. D. Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.
[29] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.
[30] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.
[31] 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 *International conference on machine learning*, 2016, pp. 1928–1937.
[32] "Architecture and mechanisms for resource elasticity provisioning," *EU H2020 project 5G-MoNArch, Deliverable D4.1*, 2018.



Joan S. Pujol Roig received a joint B.S. and M.S. degree in telecommunication engineering from Polytechnic University of Catalonia in 2014. He is currently pursuing the Ph.D. degree at Imperial College London with the Department of Electronic Engineering. His current research interests include information and coding theory, interference management in cache-aided networks, resource optimization of wireless networks, deep learning and deep reinforcement learning. From Aug. 2018 to Feb. 2019, Joan held an internship position at 5G division of Samsung Electronics R&D Institute UK.

David M. Gutierrez Estevez is a Principal Research and Standards Engineer at Samsung Electronics R&D Institute UK, where he currently attends 3GPP meetings as a global Samsung delegate in SA2 on data analytics for automation and other network architecture topics. David obtained his Engineering Degree in Telecommunications (Hons.) from the Universidad de Granada, Spain, and his M.S. and Ph.D. degrees from Georgia Institute of Technology in Atlanta, USA, the latter supported by graduate fellowships from Fundacion la Caixa and Fundacion Caja Madrid from Spain. He developed his Ph.D. thesis at the Broadband Wireless Networking Laboratory under the supervision of Prof. Ian F. Akyildiz, where he obtained the Researcher of the Year Award in 2013 for outstanding research contributions. From Sept. 2014 to Sept. 2015, David worked for Huawei Technologies as Principal Research Engineer in Silicon Valley. Previous to that, David had held an internship position at the Corporate R&D Division of Qualcomm as well as research assistant and intern positions at Fraunhofer Heinrich Hertz Institute and Fraunhofer Institute for Integrated Circuits, both in Germany. David joined Samsung in January 2016, where he has been involved in several 5GPPP projects and working groups, leading Samsungs involvement in the phase II project 5G-MoNArch on end-to-end network architecture as work package leader. David also led as technical manager the successful phase III project proposal 5G-TOURS, a 15M EUR effort with nearly 30 partners that started in June 2019 aimed at developing 5G advanced vertical trials for Europe. Davids published work accumulates over 1000 citations and is co-inventor of multiple patents and patent applications. He has served as Associate Editor of Elsevier Computer Networks Journal and as track chair and TPC member of major IEEE conferences such as INFOCOM, ICC, GLOBECOM and VTC.

Deniz Gündüz (S'03-M'08-SM'13) received his Ph.D. degree from NYU Tandon School of Engineering (formerly Polytechnic University) in 2007. After his PhD, he served as a postdoctoral research associate at Princeton University, and as a consulting assistant professor at Stanford University. He was a research associate at CTTC in Barcelona, Spain until September 2012, when he joined the Electrical and Electronic Engineering Department of Imperial College London, UK, where he is currently a Reader (Associate Professor) in information theory and communications, and leads the Information Processing and Communications Laboratory (IPC-Lab).

His research interests lie in the areas of communications and information theory, machine learning, and privacy. Dr. Gündüz is an Editor of the IEEE Transactions on Wireless Communications and IEEE Transactions on Green Communications and Networking. He also served as a Guest Editor of the IEEE JSAC Special Issue on Machine Learning in Wireless Communication (2019). He is a Distinguished Lecturer for the IEEE Information Theory Society (2020-2021). He is the recipient of the IEEE Communications Society - Communication Theory Technical Committee (CTTC) Early Achievement Award in 2017, a Starting Grant of the European Research Council (ERC) in 2016, several best paper awards (IEEE ISIT, WCNC, GlobalSIP). He served as the General Co-chair of the 2019 London Symposium on Information Theory, 2018 International ITG Workshop on Smart Antennas, and 2016 IEEE Information Theory Workshop.