

# Optimizing network slice placement using Deep Reinforcement Learning (DRL) on a real platform operated by Open Source MANO (OSM)

Alexandre Sabbadin<sup>1,\*</sup>, Abdel Kader Chabi Sika Boni<sup>1</sup>, Hassan Hassan<sup>1</sup> and Khalil Drira<sup>1</sup>

<sup>1</sup>LAAS-CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France

## Abstract

Optimizing network slice placement in 5G networks requires efficient algorithms. Deep Reinforcement Learning (DRL) has been used to solve this problem successfully. However, few works have tackled the deployment of these algorithms in a real environment. In this paper we present a DRL based algorithm aiming to optimally place network slices in IoT networks. We evaluate the performance of this algorithm in a real network deployed on Grid'5000 platform, operated by an Open Source MANO (OSM) middleware. The simulation results show a good convergence of the algorithm and the deployment in the real environment gives us some insights about a potential slicing architecture using OSM, the processing of a DRL agent in real conditions, and limitations due to consequent instantiation times for Virtual Network Functions (VNF).

## Keywords

Deep Reinforcement Learning, Slicing, IoT systems, Open Source MANO

## 1. Introduction

In the current evolving landscape of modern telecommunications, the concept of **network slicing** has emerged as a groundbreaking paradigm that promises to revolutionize how we manage and optimize network resources [1]. Network slicing allows network operators to divide their physical infrastructure into virtualized, dedicated, and isolated networks. Each slice (e.g. virtualized network) is tailored to specific service requirements, such as low latency for augmented reality services, massive bandwidth for video streaming or ultra-reliability for autonomous cars. As the deployment of network slices becomes more complex, it brings us a consequent challenge: the slicing optimization problem. This problem revolves around efficiently allocating network resources, such as compute and storage, across various slices while ensuring that each slice meets its specific quality-of-service (QoS) requirements. This optimization problem becomes even more challenging in dynamic, real-time environments where network conditions fluctuate, users make requests and resources need to be continuously adjusted. To address the slicing optimization problem, innovative approaches have been introduced

---


TACC 2023: Tunisian-Algerian Joint Conference on Applied Computing, November 6 - 8, Sousse, Tunisia

\*Corresponding author.

✉ alexandre.sabbadin@laas.fr (A. Sabbadin); akchabisik@laas.fr (A. K. Chabi Sika Boni); hassan.hassan@laas.fr (H. Hassan); khalil@laas.fr (K. Drira)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

using **Deep Reinforcement Learning (DRL)** algorithms. DRL algorithms have demonstrated remarkable learning and adaptation capabilities to complex environments, making them a promising tool for network operators who wish to optimize resource allocation in a dynamic slicing context. We decided to evaluate the performance of the DRL algorithm in both simulation and a real-world environment (using Grid'5000 infrastructure [2]). Our choice is driven by the will to ensure the reliability and practical applicability of our research. Simulations offer a controlled setting to fine-tune algorithms, due to low-time runs, but they often simplify the complexity of real-world networks. On the other hand, deploying DRL algorithms in the Grid'5000 infrastructure allows us to confront the unpredictability, noise, and dynamic nature of actual network environments. By undertaking this comparative analysis, we seek to validate the algorithm's performances beyond theoretical expectations and lay a robust foundation for its practical deployment.

The rest of the paper is organized as follows: in Section 2, we present related work about deployment of network slicing management systems, then we introduce the architecture that we used in this study in Section 3. A brief introduction to DRL algorithms is given in Section 4 and, in Section 5, we present the result of our evaluation in simulation and real-world environment. Finally we conclude by giving some directions of our future research.

## 2. Related work

Most network slicing implementations in practical scenarios primarily target 5G networks. One such example is the 5GCity initiative, as outlined in [3], which aims to provide 5G services to both citizens and businesses in a smart city environment. Another example is presented in [4], where the authors have developed a 4G/5G testbed to explore network slicing capabilities. Network slicing is based on two principles, namely **Network Function Virtualization (NFV)** and **Software-Defined Networking (SDN)**. These principles play a central role in enabling the dynamic and efficient deployment of network slices.

The NFV architecture of the European Telecommunications Standards Institute (ETSI) provides a standardized approach to virtualizing network functions, enabling network operators to virtualize and operate network functions as software on hardware devices. This architecture consists of three layers. Firstly, the virtualized infrastructure (NFVI) : this layer provides the infrastructure to host virtualized network functions. It can include servers, storage devices, network devices, and other hardware and software elements necessary for network function virtualization. Then, we have the ETSI standard **MANO** (Management and Orchestration), which controls the creation, deployment, and management of VNFs on the virtualized infrastructure. In [5], two MANO solutions are examined alongside others, namely Open Source MANO (OSM) [6] and Open Network Automation Platform (ONAP). It encompasses resource management, service orchestration, monitoring, and notification functions. The ETSI NFV standard operates with **Virtual Infrastructure Managers (VIM)**, of which two are evaluated in [7]: OpenStack [8] and OpenVIM. Additionally, OSM is compatible with major cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform. Multiple VIMs can be employed simultaneously, as exemplified in [9]. Finally, **Virtualized Network Functions (VNF)** provide the network functions that can be deployed on the virtual infrastructure. Within

the context of 5G network slicing, VNFs that use OpenAirInterface (OAI) as discussed in [10] and [4] serve to emulate a 5G network.

SDN complements NFV by offering centralized control and programmability over network resources, thereby allowing dynamic allocation of bandwidth, routing, and other network elements to optimize the performance of individual slices in real-time. The utilization of an SDN controller is evocated in [9], where the authors present a comprehensive architecture and experimental validation. Collectively, NFV and SDN furnish the agility and flexibility necessary to create, manage, and adapt network slices effectively.

This paper's primary focus is on the implementation of a DRL algorithm for optimizing VNF placement within a real infrastructure managed by OSM. Our cloud infrastructure is based on OpenStack, specifically MicroStack [11]. It is important to note that the implementation of dynamic routing utilizing an SDN controller is a topic left for future research works. Also, our work exclusively deals with resource allocation, and as such, our VNFs do not engage in real network function operations.

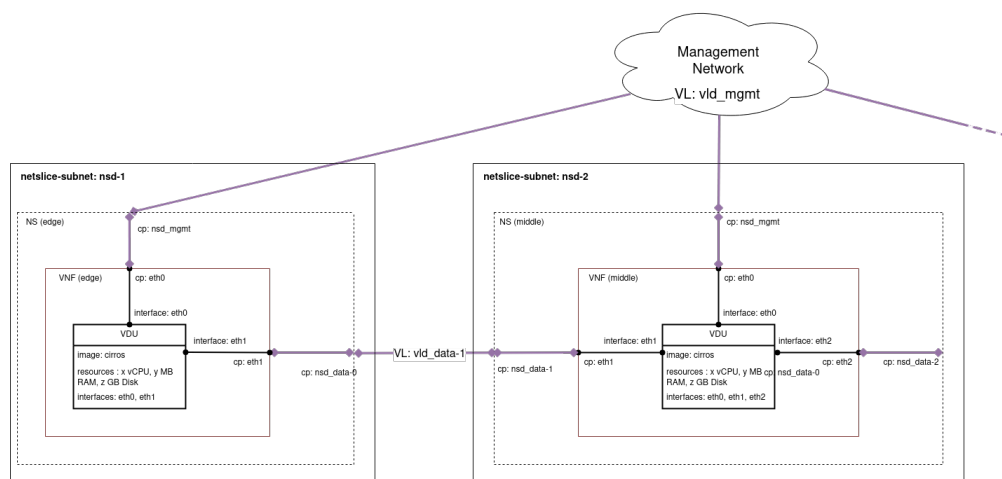
### 3. Architecture concepts

#### 3.1. Open Source MANO

Open Source MANO (OSM) [6] is an open-source project that delivers a comprehensive network management and virtualization service platform for telecommunications networks, based on the principles of Network Function Virtualization (NFV). OSM offers a software infrastructure for creating, orchestrating, managing, and supervising virtualized network services. OSM's primary objective is to ease the adoption of NFV architecture among telecommunications service providers by providing an open and flexible platform aligned with industry standards and compatible with various vendors' equipment. This initiative is sustained by a community of developers and contributors representing various organizations and companies, operating under ETSI. OSM has gained widespread adoption within the telecommunications industry for network function virtualization and cloud service management for the past years.

OSM provides a comprehensive solution for implementing network slicing: *NetSlices*. In OSM, a slice is instantiated by defining a Network Slice Template (NST), which is divided into `netslice-subnets` and `netslice-vld`, that can be duplicated as depicted in Figure 1. The former corresponds to the network services (NS) within the slice, while the latter represents the virtual links (VLs) that interconnect them. It's worth noting that OSM employs a management network for the deployment and management of slice instances. The NS, previously mentioned, serve as the network services to be deployed within our infrastructure. They act as wrappers for our VNFs and establish connection points for linking them through VLs. Not all NS have the same number of connection points, with an extra connection point for those in the "middle" of the slice. Within an NS, there is one or more VNFs, each defining the network function used, such as a firewall or router. Finally, at the level of each VNF, characteristics of one or more Virtual Deployment Units (VDUs) must be specified to define the properties of the virtual machine (image, number of CPUs, RAM, disk space, etc.). All these configurations are done using YAML-format descriptors, allowing connections between the various layers mentioned earlier through an identification and referencing system. To simplify the creation of slices, we

developed generic template descriptors files<sup>1</sup>, which can be customized for diverse cases. It is important to emphasize that all the descriptors that compose a slice must be determined in advance before initiating the instantiation request to OSM. As a result, our algorithms need to consider what we refer to as "oneshot" placement, where the placement of all VNFs must be determined simultaneously.



**Figure 1:** NetSlice template architecture in OSM

### 3.2. Grid'5000

Grid'5000 [2] is a dedicated computer research infrastructure designed for large-scale experimentation and validation of technologies and applications in the fields of Cloud computing, High Performance Computing (HPC), Big Data, and Artificial Intelligence (AI). It consists of a network of high-performance computing clusters distributed across nine university sites in France: Grenoble, Lille, Luxembourg, Lyon, Nancy, Nantes, Rennes, Sophia Antipolis, and Toulouse. Each cluster consists of interconnected compute nodes, storage resources and high-speed networking, providing a highly configurable and isolated test environment for researchers. Grid'5000 facilitates large-scale experiments on distributed computing and storage infrastructures, allowing the assessment of the performance of new architectures, algorithms, applications, scheduling policies, and more. Users can access Grid'5000 through a command-line interface, an API, or specific tools. Grid'5000 is widely used within the computer research community in France and internationally.

To access Grid'5000 resources, users need to make a reservation from a frontend server. Once access to these resources is granted, users are free to use superuser privileges on the reserved machines. This means they can install whatever they need for their experiments, as the machines will be automatically reinstalled at the end of their reservation period.

<sup>1</sup>Based on OSM Information Model : <https://osm.etsi.org/docs/user-guide/latest/11-osm-im.html>

### 3.3. Proposed architecture

We propose an architecture to address resource allocation challenges in the context of NFV and cloud computing environments. In this architecture, we define key terms as follows:

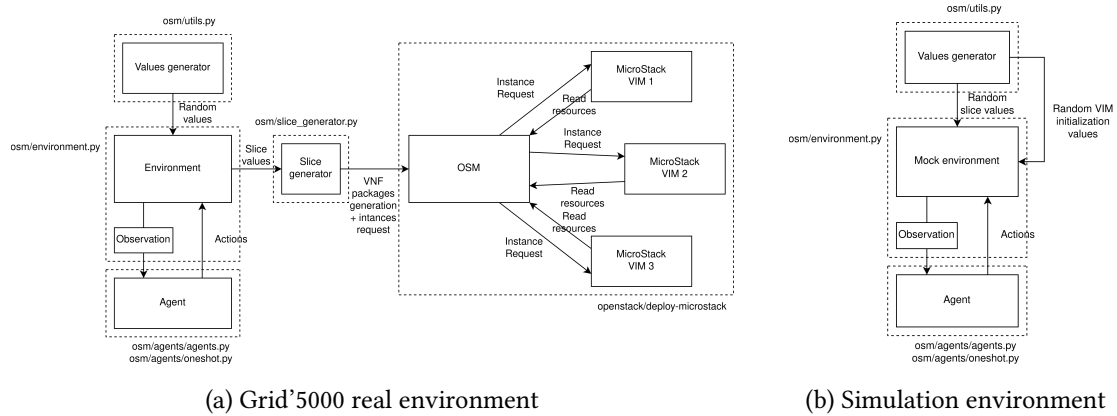
- **SLICE**: A slice represents a sequential chain of VNFs, each with specific CPU, RAM, and storage requirements.
- **ITERATION**: An iteration corresponds to the instantiation of a single slice, with VNFs initialized with randomly generated resource values.
- **EPISODE**: An episode is a loop of iterations that ends when a slice can not be instantiated. Essentially, it answers the question: "How many slices can we create?"
- **TEST ENVIRONMENT**: We conduct tests on both a simulation and a real environment using the same sequence of slices to ensure comparability across environments. Each slice maintains a consistent number of VNFs. It is subject to change in future research works.
- **VIM**: In our architecture, each VIM corresponds to a distinct datacenter. All VIMs must be accessible by a central node which hosts OSM.

We employ MicroStack [11], which offers us a single or multi-node OpenStack [8] deployment. While initially designed for developers to prototype and test, MicroStack is also suitable for edge computing, IoT applications, and appliances. This technology packages all OpenStack services and supporting libraries into a single, easily installable, upgradable, or removable package, simplifying deployment. In our specific use case, each VIM corresponds to a single machine equipped with a single-node MicroStack installation. However, MicroStack also supports multi-node clustering configurations. In this multi-node case, the VIM's amount of resources is the total of all nodes' resources (in CPU, RAM and storage). We developed a collection of *Python* scripts to facilitate communication between an environment (either simulated or real) and an agent. In the real environment, we employ the `osmcClient` library to establish communication with an OSM server instance hosted on Grid'5000.

Our reinforcement learning (RL) model operates within a dynamic environment characterized by the allocation of VIMs resources to satisfy the requirements of different network slices. Slices requirements are randomly generated by the `partValues` generator in Figure 2 at each iteration. An observation is generated at the beginning of each iteration. This observation combines the available resources within the VIMs with the slice resource requirements. It provides a comprehensive view of our current network to the agent. With this observation, our agent's objective is to decide where to instantiate a VNF. The agent uses its learned policy (i.e. behaviour) to make a decision. Our choice of a an agent is detailed in Section 4. The environment responds to the agent's decision by applying the chosen action. In other words, it initiates the instantiation of the network slice, but only if the chosen VIM has the necessary resources available. Then, a new observation, reflecting the new state of the network, is generated alongside a reward for the agent. The reward is based on the agent performance on its action choice and its purpose is to improve the agent's policy.

The major difference between the simulated and real implementation comes with the environment's modification. In our real environment displayed Figure 2a, OSM is responsible

for orchestrating VNFs and VLs instantiation, using OpenStack (MicroStack) API to create Virtual Machines (VMs) and networks based on descriptors mentioned in Section 3.1. For this, descriptors are generated based on slice resources requirements and placement decisions. In our simulation environment, the resources are "emulated" within the Mock Environment shown in Figure 2b. A slice placement amounts to subtract the required resources to the available resources.



**Figure 2:** Developed architectures

## 4. DRL algorithm description

### 4.1. DRL

DRL establishes an interaction between an agent equipped with neural networks and an environment. The latter is characterized by states whose transitions occur through actions. In the Network Slice Placement problem, the physical infrastructure (comprising VIMs) and slice placement requests are either present or received within the environment, and the number of possible actions is equal to the number of VIMs where VNFs can be placed. The various pieces of information exchanged between the agent and the environment are as follows:

- **STATE:** When a slice placement request is received by the environment, a real-time description (i.e., state or observation) of the physical infrastructure's VIMs and the elements of the request (VNFs) is transmitted to the agent. We define our set of VIMs by  $\mathcal{N}$  and VNFs by  $\mathcal{F}$ . The description of VIMs (respectively VNFs) includes the available (respectively required) CPU, RAM, and storage space  $c_i^r, \forall i \in \mathcal{N}$  (respectively  $\nu_k^r, \forall k \in \mathcal{F}$ ).
- **ACTION:** The agent takes the observation as input and outputs a VIM identifier (action) it believes to be most suitable for placing the current VNF under processing. This action is sent back to the environment for execution and evaluation of its optimality. It's important to note that in DRL, only the environment executes the actions and has the ability to assess their optimality. This assessment is reflected in a value provided to the agent to reward or penalize it.

- **REWARD:** The reward function aims to incentivize the agent to improve its future actions. For each VNF placement, there is an associated value, i.e., a reward calculated by the environment. The higher the reward, the better the placement suggested by the agent is. The agent's objective is precisely to maximize the cumulative sum of rewards it receives from the placements. In this paper, we have introduced and employed a reward function for VNF  $k \in \mathcal{F}$  defined by (1), where  $\eta$  is a small constant used to avoid division by zero.
- **NEXT STATE:** Following the application of the action, the VIMs' amounts of available resources in the physical infrastructure change, and a new slice placement request can now be processed. This new real-time description of the environment become the next state.
- **EXPERIENCE:** The combination of state  $s_t$ , action  $a_t$ , reward  $r_t$ , and next state  $s_{t+1}$  is referred as an experience, denoted  $e = (s_t, a_t, r_t, s_{t+1})$ . Experiences are used by the agent for self-improvement, as elaborated in the following section.

$$R(k) = \begin{cases} - \sum_{r \in \mathcal{R}} \sum_{i \in \mathcal{N}} x(i, k) * \frac{1}{c_i^r - \nu_k^r + \eta} & \text{for a successful VNF placement} \\ \frac{|\mathcal{R}| + \sum_{r \in \mathcal{R}} \sum_{i \in \mathcal{N}} x(i, k) * y^r(i, k) * (c_i^r - \nu_k^r)}{\eta} & \text{otherwise} \end{cases} \quad (1)$$

with

$$x(i, k) = \begin{cases} 1 & \text{if VNF } k \in \mathcal{F} \text{ is placed on VIM } i \\ 0 & \text{otherwise.} \end{cases} \quad \text{and} \quad y^r(i, k) = \begin{cases} 1 & \text{if } c_i^r < \nu_k^r \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

## 4.2. DDQN

The Double Deep Q-Network (DDQN) [12] is a value-based DRL algorithm that computes Q-values representing an estimation of expected future rewards. This algorithm is primarily designed for discrete environments, i.e., those with a finite action space. Each action is associated with a Q-value that gets updated as the DDQN agent learns from its environment.

The agent's objective is to maximize the cumulative rewards obtained throughout episodes. Achieving this goal is closely tied to the use of Q-values because whenever the agent selects the action with the highest Q-value, it aims to maximize an estimate of the cumulative reward. This estimation plays a critical role in the algorithm's behavior. DDQN employs two neural networks to better estimate Q-values: an evaluation network and a target network following equation (3). Action selection relies on Q-values from the evaluation network, while the update of these Q-values is based on former Q-values from the target network, which is an older version of the evaluation network. In DRL, the targets are not known in advance and are not fixed, hence the need for the target network, which, for a certain number of iterations, freezes the Q-values used in target calculations, effectively "freezing" the targets. Periodically, the target network is updated by copying the weights from the evaluation network.

$$Y^{DDQN} = r_t + \gamma Q_{target}(s_{t+1}, \underset{a}{\operatorname{argmax}} Q_{eval}(s_{t+1}, a)) \quad (3)$$

The trial-and-error nature of DDQN is managed through an exploration-exploitation trade-off. To ensure the selection of actions that yield the maximum reward, various actions are tested

in different states (exploration), even if this means occasionally choosing actions with lower Q-values. However, to truly maximize cumulative rewards and achieve the ultimate objective, actions with the highest estimated Q-values must be selected (exploitation). To strike a balance, we have chosen the linear decay epsilon-greedy policy, which involves selecting an action based on the maximum Q-value with a probability of  $\epsilon$  and choosing a random action with a probability of  $1 - \epsilon$ . DDQN is an off-policy algorithm, meaning that it leverages past experiences to update its policy. To achieve this, it utilizes a replay buffer where experiences are stored, and samples are periodically drawn from it to train the evaluation network, thereby improving the estimation of its Q-values. We provide the pseudo-code of DDQN in algorithm (1).

---

**Algorithm 1** Double Deep Q-Network (DDQN) pseudo-code
 

---

```

1: INPUTS :  $N_{episodes}$  (number of episodes),  $T$  (number of steps per episode),  $C$  (target
   network update frequency)
2: OUTPUT : trained DDQN agent
3:
4: Initialize  $\mathbf{R}$  (replay buffer)
5: Initialize  $Q_{eval}$  (evaluation network)
6: Initialize  $Q_{target}$  (target network)
7: Set  $Q_{target}$  weights to be same as  $Q_{eval}$  weights
8:
9: For episode = 1 to  $N_{episodes}$  do
10:   Initialize state  $s$ 
11:   For timestep = 1 to  $T$  do
12:     Choose action  $a$  with  $\epsilon$ -greedy policy based on  $Q_{eval}$ 
13:     Execute action  $a$ , observe reward  $r$  and next state  $s'$ 
14:     Store  $(s, a, r, s')$  in replay buffer  $R$ 
15:     Sample a random minibatch from  $R$ 
16:     Compute target  $\mathbf{Y}^{DDQN}$  using equation (3)
17:     Update evaluation network using gradient descent:  $\theta \leftarrow \theta - \alpha \nabla_{\theta} [Q_{eval}(s, a) - y]^2$ 
18:     Every  $C$  timesteps, update target network weights:  $Q_{target} \leftarrow Q_{eval}$ 
19:     Set  $s \leftarrow s'$ 
20: return DDQN agent

```

---

### 4.3. Real deployment considerations

Deploying our algorithm within a real environment requires us to consider several key factors. Firstly, our observation space include infrastructure state and current slice requirements (CPU, RAM and storage). All these measurements are emulated in simulation and easily retrievable with OSM. One noteworthy advantage is that the agent receive observations independently of the environment type. This means that we have no need to develop separate agents for each environment. Consequently, we can use the same model for both environments evaluations, which not only minimizes potential errors but also ensure consistent behaviour across environments. The model's training is conducted within the simulation environment, justified by the



significant time needed to instantiate a slice in a real environment, which could extend training times to several months. By training in simulation, we can significantly reduce the learning time of our algorithm.

## 5. Results

In this section, we introduce the configuration of our experimental testbed, which serves as the foundation for our research and evaluation. The testbed includes six VIMs deployed on separate Grid'5000 nodes. Each VIM is equipped with 32 virtual CPUs (vCPUs), 128 GB of RAM, and 256 GB of disk space. We use a Ubuntu 20.04 LTS distribution for each node. For our cloud infrastructure, we have deployed MicroStack instances with the OpenStack Ussuri version. To orchestrate and manage network services and resources, we have integrated OSM on a separate node with the same characteristics as the MicroStack nodes. We use OSM Release THIRTEEN for our experiment. Each generated slice contains exactly 10 VNFs whose values are randomly selected within ranges defined in Table 1.

**Table 1**

Random functions and ranges used for slice requirements generation

Resource	Type	Random function	Range	Unit
CPU	Integer	<code>numpy.random.choice</code> <sup>2</sup>	{1, 2} with probabilities {0.7, 0.3}	-
RAM	Float	<code>numpy.random.uniform</code> <sup>2</sup>	[0, 10[	GB
Disk	Float	<code>numpy.random.uniform</code> <sup>2</sup>	[0, 20[	GB

In the initial phase of our experiments, we monitored the resources state of both simulation and real environments at each iteration, with CPU in Figure 3, RAM in Figure 4 and storage in Figure 5. We successfully instantiated 12 slices, equivalent to 120 VNFs in both the simulation and real environments. The last slice (12) could not be created in either environment because of a lack of resources. We observed that differences in the allocation of resources appears just before the middle of the episode. These disparities may be attributed to variations in the agent's behavior.

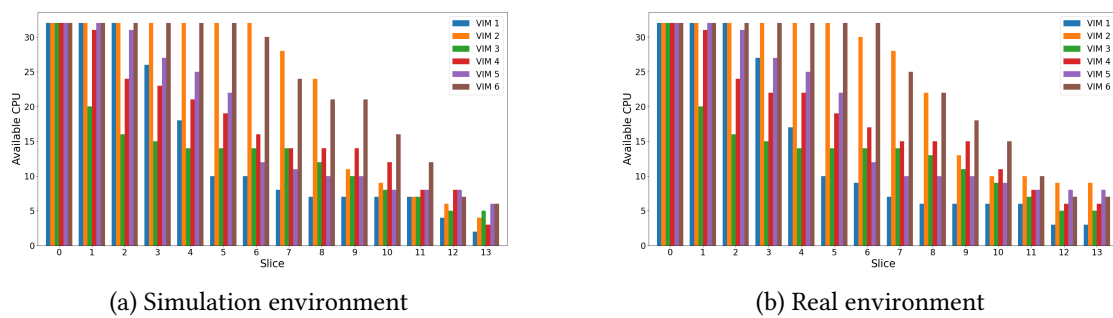
Therefore, to delve deeper into the agent's behavior, we examined its choice of VNF placement at each iteration. We plotted in Figure 6 the number of VNFs assigned to each VIM, whether or not the slice can be created. Initially, there was not any difference between the decision graphs for both environments in the early iterations. However, differences started to manifest around slice 5. Most of the differences observed in the previous figures were primarily caused by variations in the agent's decisions. During evaluation, exploration was deactivated, meaning that any small changes in the agent's actions should be attributed to disparities in the two observations.

We still needed to assess the overall impact of these disparities, so we plotted in Figure 7a the total amount of available resources for both the simulation and real environments at each iteration. The objective was to investigate whether there was a consequent difference in the overall available resources of the infrastructure. Our observations revealed that the behavior of

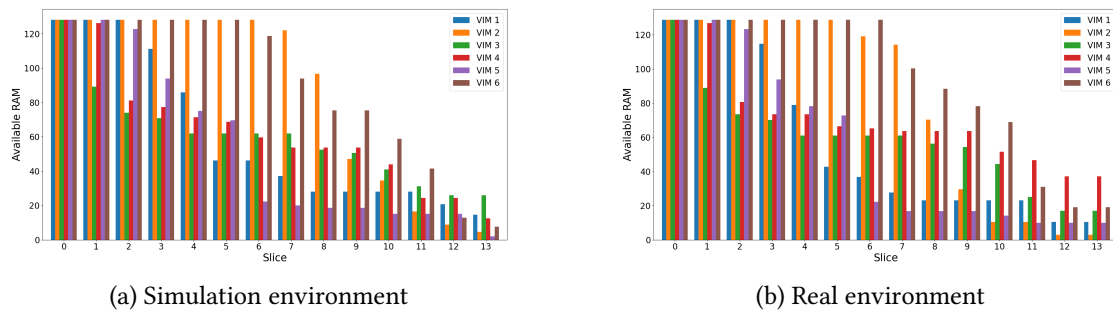
<sup>2</sup>Based on `numpy.random` functions : <https://numpy.org/doc/stable/reference/random/legacy.html>

resource utilization appeared to be linear for both experiments and across the three resource types (CPU, RAM, and Disk space). While there were some differences between the real and simulation curves, these disparities did not appear excessively significant.

As a final aspect of our analysis, we examined the relative differences ( $Relative\_diff = \frac{Sim-Real}{Real}$ ) between the total amount of resources allocated in the real and simulation environments for each iteration. Notably, we observed in Figure 7b that the simulation appeared to prioritize RAM consumption over storage consumption. Interestingly, the relative difference in CPU resources remained consistently zero throughout all iterations. These results are likely due to the use of integer values for CPU and float for the others. Then, our investigations pointed towards a potential issue with OSM, which appeared to round RAM and Disk space values. Specifically, OSM returned RAM values with three decimal numbers and provided Disk space values without any decimal precision, whereas it uses decimal precision for slice storage requirements.

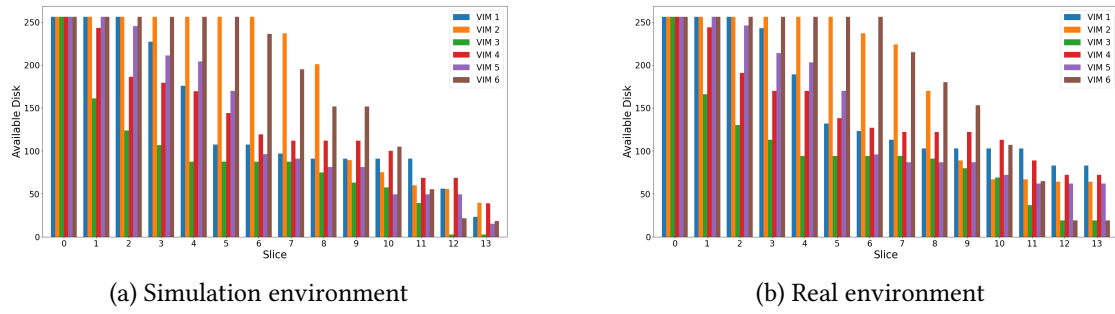


**Figure 3:** Number of available vCPUs per VIM at each iteration

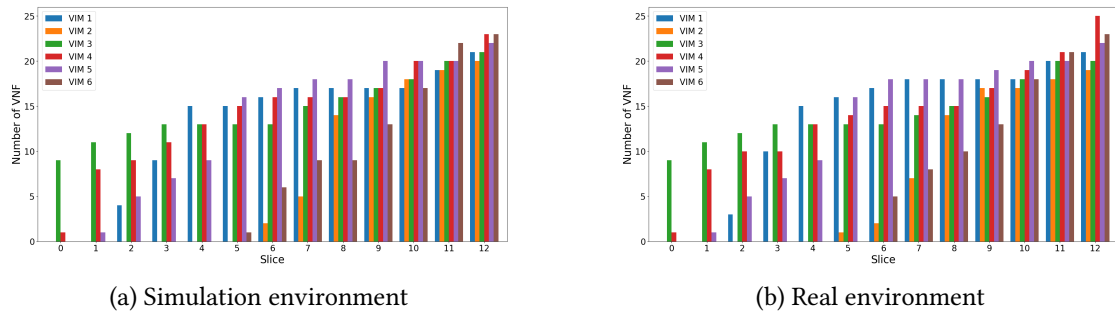


**Figure 4:** Amount of available RAM (in GB) per VIM at each iteration

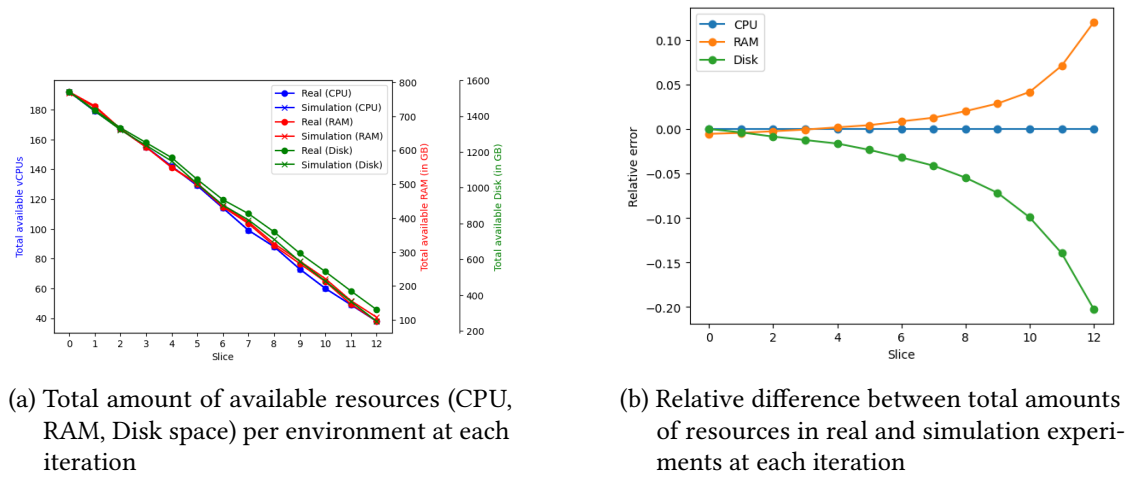
Our experiments revealed that there were no major differences in the performance (i.e. number of slices instantiated), between the simulation and real environments. However, the disparities identified in the OSM observation of the infrastructure were significant enough to impact the overall behavior of the agent.



**Figure 5:** Amount of available Disk space (in GB) per VIM at each iteration



**Figure 6:** Agent’s placement decisions per VIM at each iteration



**Figure 7:** Computation of differences between real and simulation experiments

## 6. Conclusion and perspectives

In this paper we presented a DRL based algorithm to solve the problem of optimizing network slice placement in future networks. The algorithm was trained in a simulated environment

and then evaluated in a real environment deployed on the Grid'5000 platform and managed by OSM. The results show good performances of the algorithm in real conditions with minor differences due to the collected observations in the infrastructure. However, the consequent delays required to instantiate VNFs in the infrastructure make it difficult to continue the training in real conditions once the algorithm has been deployed. In future work, we would like to introduce mechanisms to accelerate the generation of experiences, either by augmenting the training experience dataset or by using analytical models.

## References

- [1] N. Alliance, Description of network slicing concept, NGMN 5G P 1 (2016) 1–11.
- [2] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, L. Sarzyniec, Adding virtualization capabilities to the Grid'5000 testbed, in: I. I. Ivanov, M. van Sinderen, F. Leymann, T. Shan (Eds.), *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, Springer International Publishing, 2013, pp. 3–20. doi:10.1007/978-3-319-04519-1\_1.
- [3] 5GCity - A distributed cloud & radio platform for 5G Neutral Hosts, <https://www.5gcity.eu/>, 2023. [Online].
- [4] A. Esmaily, K. Kravlevska, D. Gligoroski, A cloud-based sdn/nfv testbed for end-to-end network slicing in 4g/5g, in: 2020 6th IEEE Conference on Network Softwarization (NetSoft), 2020, pp. 29–35. doi:10.1109/NetSoft48620.2020.9165419.
- [5] G. M. Yilma, Z. F. Yousaf, V. Sciancalepore, X. Costa-Perez, Benchmarking open source NFV MANO systems: OSM and ONAP, *Computer Communications* 161 (2020) 86–98. URL: <https://www.sciencedirect.com/science/article/pii/S0140366420305946>. doi:https://doi.org/10.1016/j.comcom.2020.07.013.
- [6] Open Source MANO (OSM), <https://osm.etsi.org/>, 2023. [Online].
- [7] M.-I. Csoma, B. Koné, R. Botez, I.-A. Ivanciu, A. Kora, V. Dobrota, Management and orchestration for network function virtualization: An open source mano approach, in: 2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet), IEEE, 2020, pp. 1–6.
- [8] O. Sefraoui, M. Aissaoui, M. Eleuldj, et al., OpenStack: toward an open-source solution for cloud computing, *International Journal of Computer Applications* 55 (2012) 38–42.
- [9] P. Karamichailidis, K. Choumas, T. Korakis, Enabling multi-domain orchestration using open source MANO, OpenStack and OpenDaylight, in: 2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), IEEE, 2019, pp. 1–6.
- [10] T. Dreibholz, A 4g/5g packet core as vnf with open source mano and openairinterface, in: 2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM), IEEE, 2020, pp. 1–3.
- [11] OpenStack on Kubernetes | Ubuntu, <https://microstack.run/>, 2023. [Online].
- [12] H. Van Hasselt, A. Guez, D. Silver, Deep reinforcement learning with double q-learning, in: *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016, p. 1.