Towards autonomous VNF auto-scaling using Deep Reinforcement Learning

Towards Autonomous VNF Auto-scaling using Deep Reinforcement Learning

Paola Soto*†, Danny De Vleeschauwer‡, Miguel Camelo*, Yorick De Bock*, Koen De Schepper‡, Chia-Yu Chang‡, Peter Hellinckx*, Juan F. Botero†, and Steven Latré*
*University of Antwerp - imec, IDLab - Department of Computer Science, Sint-Pietersvliet 7, 2000 Antwerp, Belgium
†Universidad de Antioquia, Department of Telecommunications Engineering, Calle 67 No. 53-108, Medellín, Colombia
‡Nokia Bell Labs, Copernicuslaan 50, 2018 Antwerp, Belgium
Corresponding author: paola.soto-arenas@uantwerpen.be

***Abstract*—Network Function Virtualization (NFV) is one of the main enablers behind the promised improvements in the Fifth Generation (5G) networking era. Thanks to this concept, Network Functions (NFs) are evolving into software components (e.g., Virtual Network Functions (VNFs)) that can be deployed in general-purpose servers following a cloud-based approach. In this way, NFs can be deployed at scale, fulfilling a great variety of service requirements. Unfortunately, the complexity in the management and orchestration of NFV-based networks has increased due to the diverse demands from a growing number of network services. Such complexity calls for an automated and autonomous solution that self adapts to the needs of those network services. In this paper, we propose and compare a Deep Reinforcement Learning (DRL) agent, a classical Proportional–Integral–Derivative (PID) controller, and a Threshold (THD)-based algorithm for autonomously determining the amount of VNF instances to fulfill a service latency requirement without knowing or predicting the expected demand. Finally, we present a comparison of the three approaches in terms of created VNFs and peak latency performed in a discrete event simulator.**

***Index Terms*—Auto-scaling, Beyond 5G, PID Controller, Reinforcement Learning**

I. Introduction

In 2015, the International Telecommunication Union (ITU) defined three usage scenarios for International Mobile Telecommunications (IMT) for 2020 and beyond, namely, Enhanced Mobile Broadband (eMBB), Ultra-Reliable Low-Latency Communication (URLLC), and Massive Machine Type Communication (mMTC) [1]. These three usage scenarios must coexist in the same network infrastructure and impose highly demanding requirements in terms of Quality of Service (QoS), throughput, reliability, and latency. One of the key enablers of 5G is NFV, which allows the network infrastructure to be shared among different stakeholders by creating virtual instantiations of physical resources.

However, the virtualization of physical resources comes hand in hand with increased network management overhead. With NFV, network managers can create different network slices to serve multiple tenants, fulfilling specific requirements. Human operators cannot keep track of all Virtual Networks (VNs) given the diverse demands of network services. Therefore, automatic solutions are needed to manage NFV-based networks that are flexible enough to adapt to various working conditions without human intervention.

The European Telecommunications Standards Institute (ETSI) has defined several Management and Orchestration (MANO) operations that are critical for the correct operation of NFV-based networks [2], and scaling is one of them. The auto-scaling problem can be defined as dynamically adding or removing network resources (e.g., VNF instances) to serve a variable workload [3]. Resource scaling can be classified as horizontal or vertical. In horizontal scaling, new VNF instances are added or removed as needed. In vertical scaling, the size of the VNF (e.g., its assigned computing and storage resources) is changed accordingly. Moreover, the scaling mode can be proactive or reactive. Proactive scaling requires the ability to infer the upcoming workload so that the resources are scheduled beforehand. In contrast, in reactive scaling, the resources are changed in response to the perceived variations. All the scaling strategies mentioned above try to find a balance between resource cost and fulfilling a given Service Level Objective (SLO) (e.g., service latency). A network operator can easily guarantee an SLO by over-dimensioning the number of VNFs but at the expense of increasing the economical cost. At the same time, the economic cost decreases when under-dimensioning the number of VNFs, but seriously compromises the SLO fulfillment.

Recently, Machine Learning (ML) strategies are being proposed for flexible resource scaling in NFV-based networks, given their ability to learn from data and past experiences. Most of the proposed strategies focused on predictive scaling, exploiting the historical data by using time-series forecasting [4]. Although it has been shown that predictive scaling produces better results in reducing the boot-up time of a new instance and yielding few SLO violations, reactive scaling is still widely used by the cloud industry due to its easiness to deploy, decent performance and low computational cost. Moreover, predicting the workload can be an easy task if the workload follows regular patterns (e.g., during the daytime, peak hours on weekdays, etc.). However, the prediction accuracy might degrade under unseen workload patterns. More recently, Reinforcement Learning (RL) is also explored as a solution for scaling network resources. An agent's objective

in RL is to learn a policy that maximizes an expected reward function by interacting with an environment through actions. Following the learned policy, the agent proactively adapts the network resources, similar to predictive auto-scalers, but without any a priori knowledge of the system.

Nonetheless, advanced predicting modules and modern RL approaches are nowadays based on Deep Neural Networks (DNNs), which require powerful computing processing. These resource-hungry DNNs might not suit resource-constrained environments (e.g., Multi-access Edge Computing (MEC) hosts). In this sense, selecting an appropriate auto-scaler is a problem that depends on the network context. Therefore, an orchestrator might select an auto-scaler depending on the available infrastructure, SLOs, operational budget, among others.

In this paper, we propose and compare three auto-scaling mechanisms that do not require any information about the workload and yet can dynamically adapt the number of VNF instances while keeping them at a reasonable level without over- nor under-dimensioning the problem. Numerical evaluations are obtained in a discrete-time event simulator. Our contributions are summarized as follows.

- We design and evaluate three auto-scaling methods: an RL-, a PID- and a THD-based algorithm that autonomously change the number of VNF instances to guarantee an SLO. Unlike predictive scalers, the proposed approaches do not require any information about the workload and yet can maintain the number of VNF instances at a level that avoids incurring in over- or under-dimensioning.
- We propose a methodology to define the RL-based auto-scaler that first maps the auto-scaling problem to a Markov Decision Process (MDP) of a well-known RL problem and then adapt it accordingly instead of defining it from scratch. This approach is contrary to most of the work in the literature where the MDPs are traditionally directly determined by the networking problem with all the difficulties of designing a properly working MDP.

The remainder of this paper is organized as follows. An overview of the related work is given in Section II. We describe the proposed auto-scalers in Section III. Section IV describes the discrete-time event simulator we use to train, tune and test the proposed solutions. Section V summarizes the evaluation of the three approaches. Finally, Section VI concludes the paper.

II. Related Work

In NFV-based networks, it is of vital importance to fulfill the SLOs of different services. By assigning more resources, a network operator can cope with the requirements imposed by the network services. Scaling is a challenging problem, mainly because it decides the exact amount of resources that a running service requires to meet an SLO. Although different techniques have been explored over a decade, we focus on the use of ML to solve the scaling problem. Lorido-Botran et al. [5] and Chen et al. [6] review the proposed solutions for autoscaling in cloud environments, while Duc et al. [4] give a broader view on resource provisioning in edge-cloud computing using ML.

Usually, scaling can be divided into reactive and predictive, as mentioned in Section I. Reactive scalers respond to the current system status. In contrast, predictive scaling involves scheduling resources for upcoming network states. Such states are generally represented by the inferred future demands that the network needs to provide. As for predictive scaling, the delay between the scaling action and execution is reduced. The most recent applications of ML are in the area of predictive scaling [7]–[9]. For instance, Subramanya and Riggio [10] developed a proactive auto-scaler focused on a distributed MEC-NFV deployment. In their work, a Neural Network (NN) was proposed whose input is the traffic load in a time-series form to determine the number of VNF instances per cell at a given time. However, as the authors characterized the scaling as a classification problem, building the training dataset requires defining how many VNF instances are needed to serve a given traffic load, a non-trivial task requiring expert knowledge. Additionally, the traffic load traces seen in testing might differ from the traces seen in training; therefore, a considerable amount of training data must be available and labeled. As a final remark, they leveraged the access to real traces from a network operator, but their results cannot be replicated due to privacy.

More recently, RL-based auto-scalers are being proposed. Q-Learning is one of the most used RL methods for autoscaling. Q-Learning assigns a Q-Value to an action-state pair (Q-Function) in a tabular representation. This tabular representation helps the agent to improve its policy by selecting the actions with the highest Q-Value. In continuous state or action problems, tabular Q-Learning is not scalable, i.e., the size of the Q-Table explodes in complex problems. To overcome this limitation, the Q-Table is replaced by a NN as a Q-Function approximator [11], creating Deep Q-Networks (DQNs).

In [12], Lee et al. designed a multi-tier autoscaling module. In NFV-based networks, services can also be provided by composing multiple VNF in Service Function Chaining (SFC). In these multi-tier applications, the auto-scaler must decide how many instances of VNF are inside the SFC and the tier to scale. The authors proposed an RL-based autoscaling method based on DQN that uses an own-defined NN. The NN's input and output are defined as the status of each tier and the actions the agent can take (e.g., add or remove one VNF per tier or maintain them), respectively. The definition of the NN is tightly coupled with the problem size since it depends on the length of the SFC, which makes their approach not general enough to be implemented. Although they showed that the DQN method outperforms two THD-based approaches, DQN creates more VNF instances than the other two, incurring in over-provisioning.

Another RL-based auto-scaler is proposed in [13]. The authors used Gaussian processes to improve the scaling policy to reduce the agent's errors during the exploratory phase in training. The Gaussian processes allowed them to model the system as a regression function that predicts the workload;

then, they used those predictions to run hypothetical interactions in the training of the RL-based agent. The agent runs two processes: policy improvement and policy evaluation. The former allows the agent to foresee the results of its action, while the latter is the current execution of the policy. The authors showed that, the scaling policy could be improved by using a Gaussian Process as system model, leading to a more stable response in terms of mean response time and number of created instances.

An auto-scaler for video conferencing systems is proposed in [14]. There, Gabriela et al. defined the selective forwarding unit of a video conferencing system as the VNF to be scaled. They used a DQN-based auto-scaler consisting of three layers: a Long Short-Term Memory (LSTM), a linear, and an output layer. The LSTM layer allows capturing the trend of the traffic. For doing this, the authors deployed a time-series database, so the features of the last four time steps are stored. To define the state of the RL agent, the authors did not use infrastructure-level metrics such as Central Processing Unit (CPU) usage, but application-level metrics such as the number of conferences currently existing and the total number of participants in the meetings. These application-level metrics are indirect estimators of the workload.

With NFV, network resources can change dynamically to cope with the workload. Before NFV, traditional solutions were based on over-dimensioning the network resources to support workload peaks. Reactive and proactive approaches have been proposed for the scaling problem. Due to limited space, we have reviewed recent scaling methods that employ ML techniques in this section. However, it is worth mentioning that most of the solutions are focused on predictive scaling, exploiting the capabilities of ML to forecast/predict the future workload. The key differences between our work and previous works are summarized as follows:

- To achieve high prediction accuracy in ML-based predictive scaling, a large amount of data is needed. However, there is no guarantee that the obtained ML model will generalize in unseen workloads. Contrary, our agent does not require any knowledge from the workload, making it more flexible under unseen data.
- Auto-scaling can also be solved with a Supervised Learning (SL) approach. In this case, labeled data is required. However, to define how many instances of a given VNF are needed based on a set of input features is not a trivial task, requiring expert knowledge. Our RL-based agent does not need to have labeled data to learn since the agent learns by interactions with the network, simplifying the training process.

III. Autonomous Autoscaling using Deep Reinforcement Learning

A. System Model

In NFV-based networks, multiple network functions can be deployed as small software pieces in the form of VNFs. To work properly, these VNFs require a minimum amount of resources, e.g., processing power in terms of CPUs, which indicates the number of jobs per second it can process. Moreover, these VNFs can be deployed in Commercial Off-The-Shelf (COTS) servers with limited CPU capacity, and as a consequence, the number of VNF instances is limited as well. Furthermore, the network receives a workload, which must be served under a given SLO, e.g., maximum latency. To meet this SLO, the number of computing resources must be dynamically and efficiently changed without incurring in under- or over-provisioning.

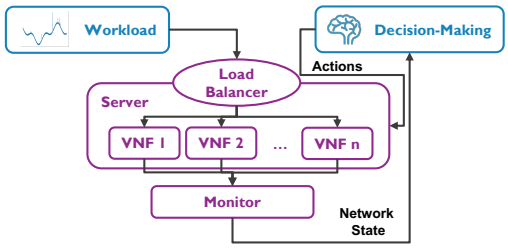


Fig. 1. Considered use-case

In this paper, we consider horizontal scaling instead of vertical scaling. Horizontal scaling can be exploited by distributed applications where the workload is shared among different instances of the same application. While vertical scaling is limited to the capacity of a single server, horizontal scaling leverages the virtually infinite computing capacity in cloud environments, making it easier to deploy several instances of the same application. Additionally, horizontal scaling allows more granular control of VNFs in multi-tier applications where each tier is scaled independently of each other. Moreover, depending on the business model, acquiring more general-purpose hardware (i.e., adding more instances) is preferable to obtaining more powerful hardware due to its cost.

The workload enters a load balancer that distributes it according to some weights[1] among the active VNFs. Each VNF has a First In, First Out (FIFO) queue for processing the assigned workload. When the queue is empty, the workload is processed immediately. If the workload cannot be processed, it waits in the FIFO queue until it can be processed. Additionally, we consider a monitor that delivers usage metrics to a decision-making agent, which automatically determines the amount of VNF replicas. Fig. 1 gives an overview of the system we consider in this paper.

B. DRL-based Agent

As stated in Section II, SL can be used to solve the auto-scaling problem. However, SL requires a labeled dataset for learning. To build this dataset, an expert must map a combination of the input features (e.g., latency, CPU usage, workload) to a scaling decision. Moreover, there is no guarantee that the trained model will perform well with input features not seen in the training phase, even if the model obtains high accuracy in

[1]In this paper, we assume equal weights for all VNF instances so that the load balancer continuously distributes the work evenly.

the inference phase. On the contrary, RL is an online-learning approach, where the agent does not need a labeled dataset since it learns from interacting with an environment.

To determine the adequate amount of VNF instances to fulfill a maximum latency SLO using DRL, the problem needs to be formulated as an MDP. An MDP is a discrete-time stochastic framework for modeling decision-making problems. This process is defined by a tuple $(\mathcal{S}, \mathcal{A}, p, r)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, p is the transition probability between states s and s' after action a is taken, and r is the immediate reward obtained for performing action a. The policy, defined as π, is a mapping function from states to actions. The solution to an MDP is an optimal policy that maximizes the expected long-term reward (which is a discounted sum of immediate rewards). To solve the MDP, several tools can be employed, RL being one of those. The optimal policy is found in RL after many agent interactions with the environment (i.e., in steady-state). Q-Learning is the most used algorithm to find this optimal policy by defining a Q-function for all state-action pairs to measure how good the policy is. Therefore, finding the optimal policy is reduced to finding the action that maximizes this Q-function. Note that the state-action pairs can be stored in a table (i.e., Q-table); however, its size will grow substantially according the number of states (e.g., a continuous state space), preventing its wide adoption. Instead, DQN uses a NN as a Q-function approximator to overcome the Q-Table's size limitation.

Contrary to most of the RL applications in networking, where the states, actions, and reward function are defined using a networking rationale, in this paper, we map the auto-scaling problem to known applications of RL. Specifically, the Gym Open-AI project provides a set of classical problems for RL algorithm benchmarking. We notice that our problem closely resembles the *Cart-Pole* [2] environment. In our problem, the agent tries to guarantee a given SLO by taking discrete actions (i.e., increase, decrease or maintain). Similarly, in the *Cart-Pole*, the cart tries to keep the pole upright by taking discrete actions (i.e., go left or right).

Following the same rationale as in the *Cart-Pole* environment, we define the information retrieved by the monitor as the network state. At time step t, the state $s^{(t)}$ is defined as:

1) Mean CPU usage among the active VNFs.
2) Mean number of jobs waiting in the queue.
3) Peak (maximum) latency from the active VNFs.
4) The number of active VNFs.

Based on this information, the DQN agent decides if the number of VNF instances must be increased, decreased, or kept the same. The reward function is also defined in a similar way as in the *Cart-Pole* problem. Our agent takes discrete actions to maintain a given continuous variable (e.g., latency) at a certain level. Consequently, the agent is rewarded if the actions are leading towards that goal. More specifically, the reward function at time step t is defined as

$$r^{(t)} = \begin{cases} 1 & |d^{(t)} - d_{tgt}| < \epsilon \cdot d_{tgt} \vee \\ & |cpu^{(t)} - cpu_{tgt}| < \epsilon \cdot cpu_{tgt} \\ 0 & |d^{(t)} - d_{tgt}| \geq \epsilon \cdot d_{tgt} \vee \\ & |cpu^{(t)} - cpu_{tgt}| \geq \epsilon \cdot cpu_{tgt} \\ -100 & \text{in episode termination cases} \end{cases} \quad (1)$$

where $d^{(t)}$ is the peak latency from the active VNFs at time step t (taken from the network state), d_{tgt} is the target latency as defined by the SLO and ϵ is a range of tolerance (e.g., 20%). Notice that if the reward function is only defined based on the perceived latency, the agent will take the most obvious action: to keep increasing the number of VNF instances, disregarding the economic impact of such a decision. To keep the number of VNF instances at an adequate level, we also let the agent be rewarded if the current CPU usage is within a predefined range. If the CPU usage is low, probably the workload can be served using fewer VNFs and vice versa. Moreover, the agent is hardly penalized if it incurs in episode termination situations (defined in Section V-A).

C. THD-based Agent

Reactive scalers are widely used in cloud applications due to their simplicity and ease of deployment. They use THD-based rules, which depend on the monitored performance metric (e.g., service latency) to perform the predefined scaling actions. Note that one disadvantage of reactive scalers is their questionable effectiveness under bursty workloads [5]. Based on the above eq. (1), a THD-based agent can be defined as follows, in which the tolerance range of CPU usage and peak latency is made up of ϵ and respective cpu_{tgt} and d_{tgt} values:

- if the current CPU usage or the peak latency is above their respective tolerance range, the number of VNF instances is increased,
- if the current CPU usage or the peak latency is below their respective tolerance range, the number of VNF instances is decreased.

As with the RL-based auto-scaler, the number of VNF instances to increase/decrease is limited to one per time step.

D. PID-based Agent

The PID agent uses the current $d^{(t)}$ and previous $d^{(t-1)}$ peak latency to decide how to set the number of VNF instances. In particular, it keeps track of a variable $\delta^{(t)}$ at time step t:

$$\delta^{(t+1)} = \delta^{(t)} + \alpha \left(d^{(t)} - d_{tgt} \right) + \beta \left(d^{(t)} - d^{(t-1)} \right) \quad (2)$$

If, at the beginning of time step $t+1$,

- $\delta^{(t+1)} \geq 1$, then the number of VNF instances is increased by 1 and $\delta^{(t+1)}$ is decreased by 1,
- $\delta^{(t+1)} \leq -1$, then the number of VNF instances is decreased by 1 and $\delta^{(t+1)}$ is increased by 1,
- $-1 < \delta^{(t+1)} < 1$ the number of VNF instances is kept the same.

[2]https://gym.openai.com/envs/CartPole-v1/

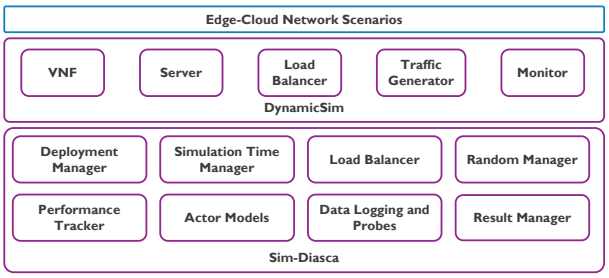


Fig. 2. Architecture of Sim-Diasca

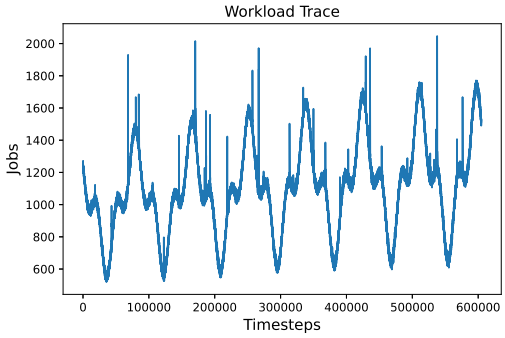


Fig. 3. Complete workload trace

TABLE I
PARAMETERS USED DURING THE SIMULATION

Parameter	Value
VNF Thread Limit	300
Server Processing Capacity	12000
Latency Target (d_{tgt})	20 ms
Cpu Usage Target (cpu_{tgt})	0.75
Tolerance (ϵ)	0.2

The second and third terms in eq. (2) are the integral and proportional terms, respectively. The former tries to keep the peak delay around the target, while the latter tries to proactively react to trends in the latency evolution. There is no differential term in eq. (2). Notice that the PID agent only needs the peak latency (current and previous value) as input. In particular, it does not need CPU usage. A PID agent is typically not designed to learn. Good values for its parameters α and β are often determined based on some runs on training data or on linearizing the system to control.

IV. Simulation Setup

To train, tune, and evaluate the performance of all the agents, we use Simulation of Discrete Systems of All Scales (Sim-Diasca), a general-purpose, parallel, and distributed discrete-time simulation engine for complex systems written in the Erlang language [15][3]. Fig. 2 shows the architecture of the simulator. Sim-Diasca (lower layer) is in charge of synchronizing time between the actors, evolving the system state, sending and receiving messages to and from the controller, and managing the results. Its built-in support for distributed simulation enables deploying a simulation case over multiple computers. Through the base actor model, own-defined models can be created. Therefore, we establish the middle layer called *DynamicSim*, in which we define an actor model for the VNFs, the server, and the load balancer. The traffic generator and the monitor modules act as an interface between the actors in the lower layer and the high-level functions defined in Python. Finally, we design several user-defined simulation cases in the topmost layer, including the one presented in Section III-A.

In a simulation case, the duration of a time step is user-defined. Within a time step, the actors simulate its functionality representing the work done in such a duration. After each actor finishes its simulated work, the time manager increases the time step by one, and the simulation goes to the next tick. At the beginning of the simulation, an initial set of actors are generated based on the defined simulation case in Sim-Diasca. This simulation case consists of a server with two VNFs and a load balancer between them. The load balancer evenly divides the incoming workload among the created VNFs. The communication between the blue and purple modules in Fig. 2 is based on pub/sub paradigm implemented in ZeroMQ to transfer structured data based on Google protobuf. The decision-making agent interacts with the simulator (i.e., environment) in regular time ticks. We make the duration of an interval between time ticks the same as the time step mentioned above. In practice, the agent communicates its scaling decision every tick and then waits until the monitor module generates a new report. Once a report is ready, the agent will receive it and evaluate the impact of its decisions.

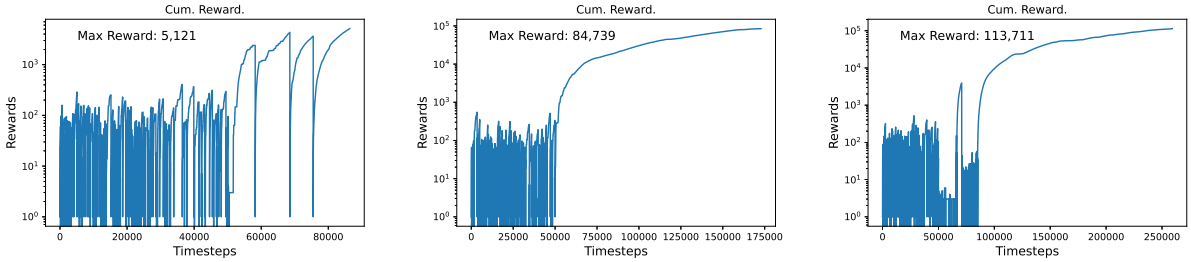
Moreover, the traffic generator (workload module in Fig 1) follows a known pattern in data centers, as shown in Fig. 3. Generally, the traffic to a data center is low at night and peaks during working hours. This pattern repeats more or less during the weekday. The traffic is generated using:

$$\begin{aligned} W(t) = \max(0, 300 \cdot (0.9 + 0.1\cos(\pi \cdot T/10)) \cdot \\ (4 + 1.2\sin(2\pi \cdot T) - 0.6\sin(6\pi \cdot T) + \\ 0.02(\sin(503\pi \cdot T) - \sin(709\pi \cdot T)))) \\ + 5N(t) + I(t) \end{aligned} \tag{3}$$

where $T = \frac{t}{86400}$, which re-expresses the time t expressed in ticks (i.e., seconds) in T days, the term $\sin(2 \cdot \pi \cdot T)$ introduces a daily pattern and $\sin(6 \cdot \pi \cdot T)$ an 8h pattern. The rest of the terms introduce some randomness so that this pattern does not repeat itself every day. In particular, $N(t)$ is a zero-mean, unit-variance Gaussian random variable and $I(t)$ introduces exponentially decaying impulses on average every 10 000s of average height 200 jobs lasting about 500s.

Every tick, a VNF asks resources to the server accordingly with the jobs they need to process but without exceeding a thread limit. This thread limit is set to 300 jobs, which represents the capacity of a CPU. Similarly, the server has a capacity of hosting maximum 40 VNFs. Table I summarizes the remaining parameters used during the simulation.

[3]https://github.com/Olivier-Boudeville-EDF/Sim-Diasca.



(a) $86.4K$ (b) $172.8K$ (c) $259.2K$

Fig. 4. Cumulative rewards from different training lengths of the DQN agent

V. Performance Evaluation

In this section, we show the details of the DQN agent training and the PID agent tuning. We finalize the section with a comparison of the three auto-scaling approaches.

A. DQN Agent

We implemented our DQN agent using Stable-Baselines3 (SB3) [16], a framework that implements popular RL algorithms in Pytorch[4]. In the definition of the DQN agent, we used the default values given by the SB3 framework. For training our DQN agent, we used episodes. An episode is defined as the number of timesteps until the simulation has to be restarted. Accordingly, we defined two situations where the episode is terminated: when the agent creates more than 20 VNFs or the jobs that are waiting in the queue (overload) are above 200. These two situations represent wrong agent behavior and must be penalized. In such situations, the episode ends, and the simulation is restarted. After the simulation is restarted, the initial scenario is again deployed.

During training, the agent tries to maximize the reward function defined in eq. (1). If the peak latency or the CPU usage of the active VNFs are within a tolerance range, the agent is rewarded; otherwise, the agent is not rewarded. If the agent falls into the episode termination cases, the agent is hardly penalized. Typically, the agent is more likely to take actions that produced a reward in the past by taking the actions that led to that situation (exploitation). However, the agent must take random and possibly new actions (exploration) to discover the actions that maximize its reward. Initially, we wanted to experimentally determine how many training steps the agent would require to perform well. Therefore, we trained our DQN agent using different lengths of the trace defined in eq. (3). In particular, we used the first [10, 30, 50, 86.4, 172.8 and 259.2K] values of the workload as training length. We noticed that using training lengths shorter than the frequency of the trace (86.4K timesteps, equivalent to one day), the agent cannot converge to a policy that keeps the simulation running[5].

Fig. 4 shows the obtained cumulative reward on a logarithmic scale during training. The figure shows the training phase of the agents that obtained better testing results from a set of three runs. As long as the simulation is running, the agent can get a reward if the metrics are within the tolerance range; otherwise, the reward is zero. Therefore, the cumulative reward is continually increasing. However, when the agent falls in the episode termination cases, the agent is hardly penalized, affecting the cumulative reward. As seen from the graphs, the agent needs at least 50K timesteps to keep the simulation running and get higher cumulative rewards. We selected the agent shown in Fig. 4b for comparison against the baseline approaches. This agent showed faster convergence, steepest slope in the positive reward area, and the testing phase results were closely similar to the agent in Fig. 4c, requiring less training time.

B. PID Agent

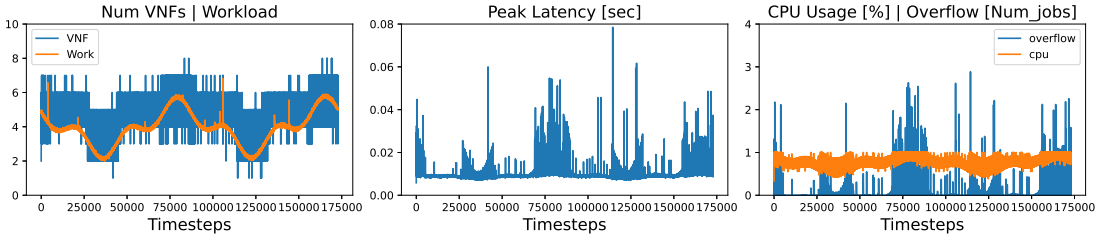
As stated above, the PID agent tries to keep the peak latency around $d_{tgt} = 20\text{ms}$. The optimal values for its parameters α and β were determined by an exhaustive search. The parameter space (α, β) was sampled by letting α range over the values $\{0.125, 0.25, 0.5, 1, 2, 4, 8\}$ and β over $\{50, 100, 200, 400\}$. Then it was determined for which of all these combinations the latency was the least amount of time above the tolerated upper bound of $(1+\epsilon)d_{tgt}$, when the PID agent controlls the first part of the workload trace, i.e., the training set. It turns out that if the training set spans the first day, the optimal parameters are $(\alpha, \beta) = (16, 200)$, while if the training set spans the first two days, the optimal parameters are $(\alpha, \beta) = (0.25, 200)$. In both these cases, the minimum is broad: relatively small changes in α and β do not alter the number of latency violations drastically so that the choice of α and β is not critical.

C. Comparison

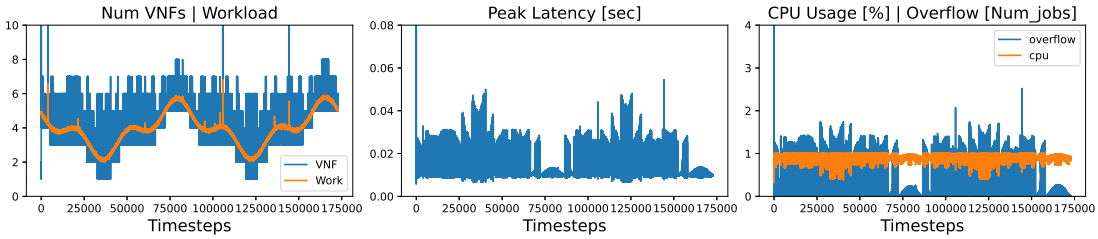
To test the agents' behavior in unseen workload traces, they were tested using the last 172.8K workload values. It is important to notice that the DQN (and THD-based) agent and the PID agent use different information as input. The former uses the instant peak latency and CPU load, while the latter uses the instant and previous peak latency. Also, the RL agent learns automatically, while the PID agent is manually tuned. Both of these facts mean that care should be taken when comparing the performance of these agents.

[4]https://pytorch.org/

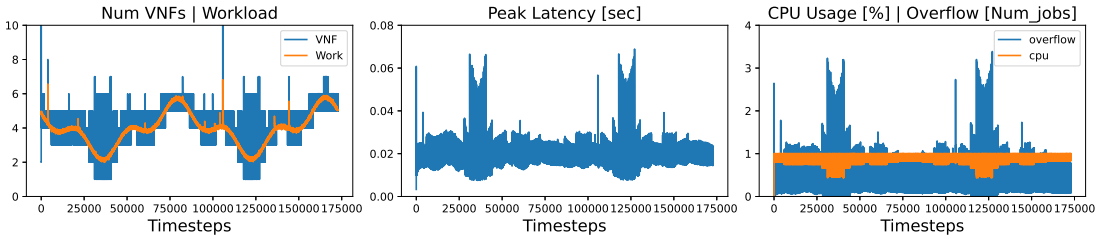
[5]We don't include these results giving the limitation on the number of pages.



(a) DQN Agent



(b) THD-based Agent



(c) PID Agent

Fig. 5. Performance of the three proposed agents in terms of the number of created VNFs, peak latency, CPU usage and overflow

TABLE II
COMPARISON RESULTS

Metric	Approach	Mean	Std	Min	25%	50%	75%	Max
Number of VNFs	DQN	4.87	0.84	1	5	5	5	8
	THD	4.32	1.25	1	3	4	5	17
	PID	4.06	1.09	1	3	4	5	10
Peak Latency [s]	DQN	0.0095	0.0025	0.0058	0.0088	0.0090	0.0093	0.0785
	THD	0.0153	0.007	0.0058	0.0099	0.0118	0.0195	0.1432
	PID	0.0198	0.0048	0.0033	0.0163	0.0194	0.0228	0.0689

TABLE III
SLO VIOLATIONS

Approach	%SLO Violations
DQN	0.69%
THD	12.20%
PID	16.57%

The behavior of each agent is shown in Fig. 5. On the left of Figs. 5a, 5b, and 5c, the testing workload trace and the number of created VNFs are shown. Every agent is tested using the same workload trace, which is different from the workload trace used during training or tuning. Generally speaking, the THD-based and the PID agents create more VNFs. Moreover, when facing the low workload valleys (around timesteps 30K and 125K), the PID and the THD-based decrease the number of VNFs to a number that cannot serve the incoming demand, increasing the peak latency and the overflow. To recover from that situation and clear the built-up overflow, both agents are forced to create more VNFs than needed. On the contrary, the DQN agent closely follows the workload trace, despite not having any information regarding it. Around the timesteps mentioned above, the DQN agent creates an amount of VNFs that can serve the incoming demand, keeping the peak latency and the overflow within desirable margins. Nonetheless, the DQN agent does not react well when facing the highest peaks of the workload (around timesteps 75K and 170K). In such situations and for short periods, the agent decreases the number of VNFs below the required, increasing the peak latency and the overflow momentarily.

The middle and right parts of Figs. 5a, 5b, and 5c show the peak latency and the CPU usage and overflow, respectively. All three agents can keep the peak latency within the desired tolerance range, with some temporary exceptions. Table II

gives a quantitative analysis of the behavior by showing the main statistical figures: mean, standard deviation, minimum, maximum, and the most representative quartiles of the peak latency and number of created VNFs. As can be seen, the DQN can maintain a more stable and lower number of created VNFs than the PID and the THD-based agents. However, this is more a secondary effect since all the agents are not designed to optimize the number of replicas. Regarding the peak latency, most of the time, all the agents can keep this metric under the upper bound (24ms). Nonetheless, as shown in Table III, the PID agent violates the upper bound 16.57% of the time while, the THD-based and the DQN are reducing the violations to 12.2% and 0.69%, respectively.

D. Discussion

This section shows the numerical evaluation of three types of agents, namely, DQN, PID, and THD-based. Although they are trained/tuned using different setups (cf. Section III), all these agents are designed with the same goal of keeping the peak latency at a target of 20ms with a tolerance of 20%. In this sense, we put them on the same table in the above experiment using the identical workload in a simulator engine to quantify their performance in a real system. Note that these results not only reveal the trend of these three agents but also set the foundation for future study on the impacts from different setups beyond instant peak latency, i.e., previous peak latency (used by PID agent) and CPU load (used by both DQN and THD-based agents).

Furthermore, choosing the applicable agent is a task beyond only performance evaluation. It also depends on both business- and operational-related conditions. On the one hand, a multi-tier Service Level Agreement (SLA) between stakeholders might show different amounts of marginal penalty among agreed objectives, e.g., a high penalty even when slightly violating the maximum service latency; therefore, the auto-scaler agent may have a higher chance to disregard the number of created VNFs. On the other hand, from the operational point of view, an operator might not have the required hardware to support ML solutions. Therefore, a DQN agent is ruled out due to its requirement to explore new actions to improve the reward, leading to unpredictable behavior on lower-end hardware.

VI. Conclusion

More and more, future networking systems are becoming autonomous by using intelligent agents to carry out several MANO operations. One of those operations is scaling. The auto-scaling problem determines how many replicas are needed to fulfill operational, economic, or business objectives. In this paper, we designed and evaluated three autonomous scaling agents using known techniques such as heuristics, classic control and RL. We compared the three agents in terms of the peak latency and the amount of created VNFs. Additionally, we discussed their advantages and disadvantages when considering their implementation.

Despite the promising results, several challenges are still open and can be addressed in future work. For instance, we will include vertical and horizontal scaling, where multiple servers need to be deployed.

Acknowledgment

This research was partially funded by Ctrl App, a research project of the Fund for Scientific Research Flanders (FWO) under grant agreement No. G055619N, and by the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 101017109 (DAEMON).

References

[1] ITU-R, "IMT Vision – Framework and overall objectives of the future development of IMT for 2020 and beyond," ITU, Recommendation, 2015. [Online]. Available: https://www.itu.int

[2] ETSI, "Open Source MANO (OSM)," accessed: 2021-08-23. [Online]. Available: https://www.etsi.org/technologies/open-source-mano

[3] ETSI, "Network Functions Virtualisation (NFV); Management and Orchestration," ETSI, Specification, 2014. [Online]. Available: https://www.etsi.org

[4] T. L. Duc *et al.*, "Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–39, 2019.

[5] T. Lorido-Botran *et al.*, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[6] T. Chen *et al.*, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *arXiv preprint arXiv:1609.03590*, 2016.

[7] J. Martín-Pérez *et al.*, "Dimensioning of v2x services in 5g networks through forecast-based scaling," *arXiv preprint arXiv:2105.12527*, 2021.

[8] S. Lange *et al.*, "Machine learning-based prediction of vnf deployment decisions in dynamic networks," in *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2019, pp. 1–6.

[9] S. Rahman *et al.*, "Auto-scaling vnfs using machine learning to improve qos and reduce cost," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[10] T. Subramanya and R. Riggio, "Machine learning-driven scaling and placement of virtual network functions at the network edges," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 414–422.

[11] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[12] D. Lee *et al.*, "Deep q-networks based auto-scaling for service function chaining," in *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE, 2020, pp. 1–9.

[13] C. H. T. Arteaga *et al.*, "An adaptive scaling mechanism for managing performance variations in network functions virtualization: A case study in an nfv-based epc," in *2017 13th International Conference on Network and Service Management (CNSM)*. IEEE, 2017, pp. 1–7.

[14] P. Gabriela *et al.*, "Machine learning-based auto-scaler for video conferencing systems," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 2021, pp. 142–150.

[15] T. Song *et al.*, "Performance evaluation of integrated smart energy solutions through large-scale simulations," in *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2011, pp. 37–42.

[16] A. Raffin *et al.*, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.