IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 23, 2021, accepted December 9, 2021, date of publication December 13, 2021,
date of current version December 23, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3135315

Scaling UPF Instances in 5G/6G Core With Deep Reinforcement Learning

HAI T. NGUYEN[1], TIEN VAN DO[1,2], AND CSABA ROTTER[3]
[1]Department of Networked Systems and Services, Budapest University of Technology and Economics, 1111 Budapest, Hungary
[2]MTA-BME Information Systems Research Group, ELKH, 1117 Budapest, Hungary
[3]Nokia Bell Labs, 1140 Budapest, Hungary

Corresponding author: Tien Van Do (do@hit.bme.hu)

The work of Hai T. Nguyen was supported in part by the Hungarian Scientific Research Fund (OTKA) under Project K138208.

ABSTRACT In the 5G core and the upcoming 6G core, the User Plane Function (UPF) is responsible for the transportation of data from and to subscribers in Protocol Data Unit (PDU) sessions. The UPF is generally implemented in software and packed into either a virtual machine or container that can be launched as a UPF instance with a specific resource requirement in a cluster. To save resource consumption needed for UPF instances, the number of initiated UPF instances should depend on the number of PDU sessions required by customers, which is often controlled by a scaling algorithm. In this paper, we investigate the application of Deep Reinforcement Learning (DRL) for scaling UPF instances that are packed in the containers of the Kubernetes container-orchestration framework. We propose an approach with the formulation of a threshold-based reward function and adapt the proximal policy optimization (PPO) algorithm. Also, we apply a support vector machine (SVM) classifier to cope with a problem when the agent suggests an unwanted action due to the stochastic policy. Extensive numerical results show that our approach outperforms Kubernetes's built-in Horizontal Pod Autoscaler (HPA). DRL could save 2.7–3.8% of the average number of Pods, while SVM could achieve 0.7–4.5% saving compared to HPA.

INDEX TERMS 5G, 6G, core, PDU session, UPF, deep reinforcement learning, Kubernetes, proximal policy optimization.

I. INTRODUCTION

The fifth-generation (5G) networks and networks beyond 5G (e.g., the sixth-generation – 6G) will provide the service for customers in various vertical industries (vehicular communication, IoT, remote surgery, enhanced Mobile Broadband, Ultra-Reliable and Low Latency Communication, Massive Machine Type Communication, etc.) [1]–[8]. The transformation of network elements and network functions from dedicated and specialized hardware to software-based containers has been started [8]. The Third Generation Partnership Project (3GPP) has specified the framework of 5G core with many network function components based on the Service Based Architecture (SBA). These 5G core architectural elements are implemented in software and executed inside either virtual machines (VM) or containers in clouds' environment [9]. In the future, 6G networks will likely maintain the user plane functions of the 5G core as well [8]. It is worth emphasizing that virtual machines and containers hosting network functions (termed instances) are executed within cloud orchestration frameworks that manage and assign the cloud resource for instances. A Network Functions Virtualization Infrastructure (NFVI) Telco Taskforce (CloudiNfrastructure Telco Task Force, CNTT) has defined a reference model, a reference architecture [10] and reference implementations based on Openstack [11] and Kubernetes [12].

In operator environments, the instances of network functions should be orchestrated (launched and terminated) in response to the fluctuation of traffic demands from customers. For example, customers initiate requests for PDU sessions before data communications; the traffic volume of such requests may depend on the periods of a day. During peak traffic periods, more instances should be launched and utilized than regular periods. That is, operators should apply appropriate algorithms to control the resource usage of network functions.

In this paper, we investigate the application of Deep Reinforcement Learning (DRL) to the resource management

The associate editor coordinating the review of this manuscript and approving it for publication was Huaqing Li.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

TABLE 1. List of notations.

Notations for the environment	
D_{min}	minimum number of Pods in the system
D_{max}	maximum number of Pods in the system
L_{sess}	maximum number of PDU sessions in the system
t_{pend}	initialization time of a Pod
d_{busy}	number of busy Pods
B	maximum number of PDU sessions per Pod
$p_{b,th}$	blocking rate threshold
λ	arrival rate of the PDU sessions
μ	service rate of sessions
ΔT	time between two decisions
Notations for the observations	
d_{free}	number of idle Pods
d_{boot}	number of booting Pods
d_{on}	number of running (idle and busy) Pods
l_{sess}	number of PDU sessions
l_{free}	number of additional sessions the system could handle
p_b	probability of blocking
$\hat{\lambda}$	approximated arrival rate since previous decision
$\hat{p}_b$	measured blocking rate since previous decision
Notations for the MDP	
$\mathcal{S}$	set of states in the MDP
$\mathcal{A}$	set of actions in the MDP
p	transition probability in the MDP
T_i	the i-th decision time
r, r_i	immediate reward function, i-th reward at time T_i
κ	reward multiplier
γ	discounting factor
$\mathcal{P}$	probability distribution over a set
$s(t)$, s_i	state at time t, i-th state at time T_i
$a(t)$, a_i	action at time t, i-th action at time T_i
π, π^*	policy function, optimal policy
V	value function

(scaling) of User Plane Function (UPF) instances in the 5G/6G core. To our best knowledge, this is the first work for scaling UPF instances based on DRL. We assume that UPF instances are controlled by the Kubernetes container-orchestration framework, so we compare the DRL approach to Kubernetes's built-in Horizontal Pod Autoscaler (HPA) and found that DRL can perform better than the HPA. We find that the policy generated by the DRL method could make unwanted decisions occasionally. To remove the randomness from the policy, we apply a support vector machine (SVM) to classify actions based on a pre-trained DRL agent. As a result, we got a deterministic SVM-based policy at a slight performance degradation that could still perform just as well or even better than the HPA. Our contributions are as follows:

- We formulate the problem of scaling UPF instances in a Kubernetes-driven cloud. We apply model-free DRL to this problem and compare its performance to the Kubernetes HPA. Through simulations we show that DRL can outperform the HPA.
- After this, we show that in some cases, more particularly in case of sudden increases in traffic, the DRL agent may pick the unwanted action. This is due to the stochastic nature of the learned policy. We remedy this by generating a dataset of states with actions as labels and train a SVM to classify states and actions. We show that the performance of the SVM-based agent is slightly worse than the DRL agent when traffic change is slow. However, under sudden traffic change, the deterministic policy of the SVM agent does not take unwanted actions.

In Section II we review the related literature on autoscaling in clouds, reinforcement learning (RL) methods for scaling, and resource management in the 5G core. In Section III we describe the problem of scaling UPF instances in Kubernetes. In Section IV we formulate a Markov decision problem for scaling UPF instances and present the Deep Reinforcement approach. In Section V extensive numerical results are delivered. Finally in Section VI we draw our conclusions.

II. RELATED WORKS

About autoscaling in cloud environments, Lorido-Botran *et al.* [13] conducted a survey that classifies existing solution methods into five categories: threshold-based rules, control theory, reinforcement learning, queueing theory, and time series analysis. Zhang *et al.* [14] designed a threshold-based procedure to scale containers in a cloud platform and then measured the elasticity of their algorithm. In a hybrid approach, Gervásio *et al.* [15] combined an ensemble of prediction models with a dynamic threshold algorithm to scale virtual machines in an AWS cloud. Ullah *et al.* [16] used genetic algorithm and artificial neural networks to predict CPU usage in a cloud and then used a threshold-based rule.

Several works studied the performance of Kubernetes. Nguyen *et al.* [17] compared metrics server solutions and highlighted the effects of various configuration parameters under both resource metrics. Casalicchio [18] investigated the autoscaler in Kubernetes and showed that scaling based on absolute measures might be more effective than using relative measures.

Reinforcement learning has been used to tackle scaling and scheduling problems in clouds as well. Horovitz and Arian [19] used tabular Q-learning to autoscale in cloud environments. They focused on the reduction of the state and the action space as they did not use function approximation for the Q-values. Their experiments scaled web applications on Kubernetes. They also proposed the Q-threshold algorithm where Q-learning was used to control the parameters of a threshold rule. We found that Q-threshold has difficulty finding the optimal policy with our reward formulation. This is mainly because this algorithm cannot control the Kubernetes Pods directly which means actions do not have direct effect on rewards either. Shaw *et al.* [20] compared the Q-learning and the SARSA algorithms on virtual machine consolidation tasks. In these tasks the objective of the RL agent was to use live migration and place virtual machines on the approriate nodes to minimize resource usage. Garí *et al.* [21] conducted a survey of previous RL solutions for scaling and scheduling problems in the cloud. Rossi *et al.* [22] compared the Q-learning, the Dyna-Q, and a full backup model-based Q-learning to autoscale Docker Swarm containers horizontally and vertically. They measured the transition rate between different CPU utilization values to

estimate the model, however this method is difficult to scale as it needs to store the number of transitions between every state. Cardellini *et al.* [23] created a hierarchical control for data stream processing (DSP) systems. On the top-level they used token-bucket policy, on the bottom level they considered the threshold policy, Q-learning, and full backup model-based RL. Schuler *et al.* [24] used Q-learning to set concurrency limits in Knative, a serverless Kubernetes-based platform, for autoscaling.

Some works specifically focus on the 5G core. Járó *et al.* [25] discussed the evolution and virtualization of network services. They considered the availability, the dimensioning, and the operation of a Telecommunication Application Server. Tang *et al.* [26] proposed a linear regression-based traffic forecasting method for virtual network functions (VNFs). They also designed algorithms to deploy service function chains of VNFs according to the predicted traffic. Alawe *et al.* [27] used deep learning techniques to predict the 5G network traffic and scale AMF instances. Subramanya *et al.* [28] used multilayer perceptrons to predict the necessary number of UPFs in the system. Kumar *et al.* [29] devised a method for scaling up and scaling out UPF instances and deployed a 5G core network on the AWS Cloud. Rotter and Do [9] presented the queueing analysis for the scaling of 5G UPF instances based on threshold algorithms has been presented. Their queueing model provides a quick evaluation of scaling algorithms based on two thresholds.

From the related literature review, it is observed that scaling algorithms reported in most of the literature works so far control the number of VM, VNF, container instances with the use of some thresholds [14], [15], [18], [30]–[33]. However, to our best knowledge, no work exists on the application of artificial intelligence methods for the resource management of UPF instances.

III. THE OPERATION AND SCALING ISSUE OF 5G UPF INSTANCES

A. 5G UPF

The connection between the User Equipment (UE) and the Data Network (DN) in 5G requires the establishment of a PDU session. In this connection the UE first directly connects to a gNB in the Radio Access Network (RAN), and through the transport network reaches the 5G Core, which provides the end point to the DN (see Figure 1) [1]–[4], [34].

The transport network may be wireless, wired, or optical connection [35] and the 5G core consists of a collection of various network functions implementing a Service Based Architecture (SBA). Such network functions are the Access and Mobility Management Function (AMF), which performs the authentication of UEs and controls the access of UE to the infrastructure; the Session Management Function (SMF), which helps the establishment and closing of PDU sessions and keeps track of the PDU session's state; and the User Plane Function (UPF) [1], [3]. Whereas the AMF and the

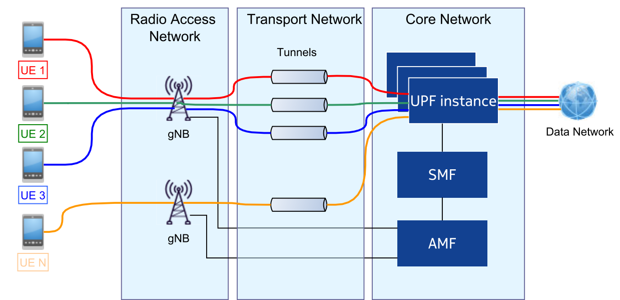


FIGURE 1. The role of the 5G UPF [9]. Each line represents a connection to a PDU session. An UPF instance may handle mutiple PDU sessions, the core network may contain multiple UPF instances.

SMF are part of the control plane, the UPF is responsible for the user plane functionality. UPF serves as a PDU session anchor (PSA) and provides a connection point for the access network to the 5G core. Additionally, a UPF also handles the inspection, routing, and forwarding of the packets and it can also handle the QoS, apply specific traffic rules, etc. [36]. The control and user plane separation (CUPS) guarantees that the individual components can scale independently and also allows the data processing to be placed closer to the edge of the network.

B. 5G UPF INSTANCES WITHIN THE K8S FRAMEWORK

Kubernetes is an open-source container-orchestration platform that manages containerized applications on a cloud-based infrastructure [37]. A Pod is the smallest deployable computing unit for a specific application in Kubernetes and may contain more containers. Machines on the cloud, either physical or virtual, are referred to as nodes with a specific set of resources (CPU, memory, disk, etc.). To deploy applications, the Kubernetes controller can configure Pods with a given resource requirement on the nodes and run one or more containers inside these Pods.

If the 5G core elements are packed into containers that are organized into Pods, the resource of each Pod and the execution of Pods are managed by Kubernetes. To establish UPF, it is a natural choice to map a UPF instance to a Pod. That is, a Pod runs a single container hosting a UPF software image. Figure 2 depicts an example cluster with multiple nodes, each having various Pods that host UPF software handling PDU sessions. In this paper, such a Pod is called a UPF Pod and UPF instance.

5G networks may serve various subscribers with different types of demands. Therefore, PDU sessions may have different requirements. To ease the operation, operators may apply a practical approach to limit the number of PDU session types. For each PDU session type a UPF instance type is created with identical resource requirement.

C. THE PROBLEM OF SCALING UPF PODS

The purpose of scaling UPF Pods is to save the resource consumption of the system. A scaling function changes (start

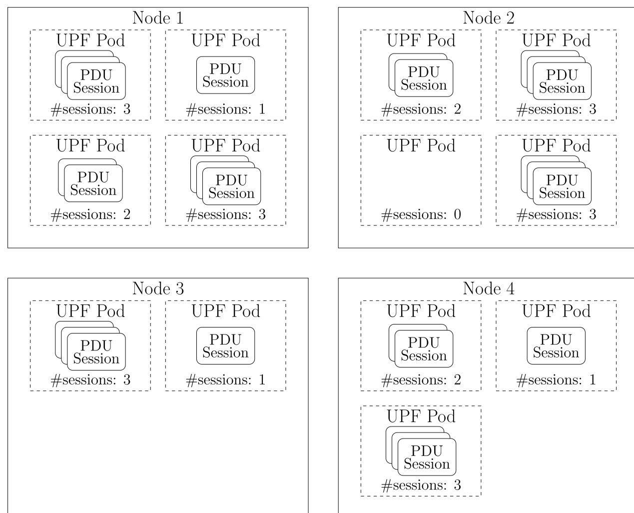


FIGURE 2. An example cluster with 4 nodes, hosting $d_{on} = 13$ UPF Pods respectively, handling a total of $l_{sess} = 27$ PDU sessions. Each Pod may handle multiple PDU sessions. It may also happen that a Pod does not handle any sessions and becomes idle in the node.

new Pods, or terminate existing ones) the number of UPF Pods depending on the number of PDU sessions required by UEs. On the one hand, if the number of UPF Pods is too low, the QoS degrades since we do not have enough UPF Pods to handle new incoming PDU sessions. On the other hand, if the number of UPF instances is too high and the load is low, a lot of reserved resource increases the operation cost. Therefore, a trade-off between the QoS and the operation cost is to be achieved.

For each type of PDU sessions, we assume that at least $D_{\min}$ Pods are initiated, $D_{\max}$ Pods can be started, each Pod simultaneously could handle maximum L_{sess} sessions. Each Pod takes t_{pend} time to boot, and their termination is instantaneous. Let $d_{\text{on}}(t)$ denote the number of running Pods in the system at time t. Therefore, $D_{\min} \leq d_{\text{on}}(t) \leq D_{\max}$ holds and the limit for the number of sessions in the system is $D_{\max} L_{\text{sess}}$. Let $l_{\text{sess}}(t)$ denote the number of sessions in the system at time t. Then we have $0 \leq l_{\text{sess}}(t) \leq D_{\max} L_{\text{sess}}$. Additionally, let us define a free slot as an available capacity for a session and denote their number with $l_{\text{free}}(t)$ at time t. Obviously, $l_{\text{free}}(t) + l_{\text{sess}}(t) = D_{\max} L_{\text{sess}}$.

A PDU session can only be created if there is free capacity in the cluster, that is $l_{\text{free}}(t) > 0$. In this case new PDU sessions are assigned to the appropriate UPF Pods by a load balancer. If $l_{\text{free}} = 0$ and there is no capacity left, the session and the UE's request is blocked. We denote the blocking rate, the probability of blocking a request, with p_b.

The list of basic notations is summarized in Table 1.

D. K8s HPA

Kubernetes autoscaler is responsible for the scaling functionality. Figure 3 shows the interactions between the autoscaler and other components. A metrics server monitors the resource usage of Pods and provides the autoscaling entity with statistics through the Metrics API. The autoscaler computes the necessary number of Pod replicas and may decide on a scaling action. The adjustment of the replica count can be done through the control interface.

The Horizontal Pod Autoscaler (HPA) is Kubernetes's default scaling algorithm. It uses the average CPU utilization, denoted by $\bar{\rho}$, as an observation to compute the necessary number of Pods, denoted by $d_{\text{desired}}(t)$ at time t. It has two configurable parameters: the target CPU utilization ρ_{target} and the tolerance ν. The equation used by the HPA is

$$d_{\text{desired}}(t) = \left\lceil d_{\text{on}}(t) \frac{\bar{\rho}(t)}{\rho_{\text{target}}} \right\rceil, \tag{1}$$

where $d_{\text{on}}(t)$ is the number of Pods at time t. The HPA then checks whether $d_{\text{desired}}(t)/d_{\text{on}}(t) \in [1 - \nu, 1 + \nu]$. If it is not, the HPA issues a scaling action to bring the replica count closer to the desired value. The above described procedure is executed periodically with ΔT interval. This time interval can be set through Kubernetes configurations.

IV. SCALING UPF PODS WITH DRL

The application of the built-in Kubernetes HPA needs the appropriate values of ρ_{target} and ν. A system operator may go through an arduous process of trials and errors to find the configuration that could minimize the Pod count while maintaining QoS levels. Instead, in this paper, we propose the application of Deep Reinforcement Learning (DRL) to set the Pod count dynamically depending on the traffic, without the assistance of an operator. The DRL agent observes the system and determines the correct action output through the continuous improvement of its policy. In what follows, we present our approach regarding the design of the DRL agent.

A. FORMULATION OF THE MARKOV DECISION PROBLEM

Before applying a reinforcement learning algorithm we need to formulate the problem as a Markov decision problem (MDP). This means we need to define the state space $\mathcal{S}$, the actions space $\mathcal{A}$, and the reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. A complete definition of the MDP would also require the state transition probability $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ and the discounting factor $\gamma \in [0, 1]$. Here p is a probability that the system enters a next state when an action happens at the current state, and such a transition results in real value reward r. To avoid the specification of the p transition function as in a model-based formulation (like in [22]), we decided to use a model-free RL method. Also, γ is implicitly contained in other hyperparameters as we will see later.

In the MDP framework an agent interacts with the environment described by the MDP. At the decision time t it observes the state $s(t) \in \mathcal{S}$ and following its policy $\pi : \mathcal{S} \to \mathcal{A}$ it makes an action $a(t) \in \mathcal{A}$. As a result the agent receives a reward $r(t)$ and at the next deicision time it can observe the next state.

Let us denote the i-th decision time with T_i $(i = 0, 1, \ldots)$. In our case the time between two decisions is ΔT, that is $T_{i+1} - T_i = \Delta T$ $(i = 0, 1, \ldots)$. Furthermore, we will also

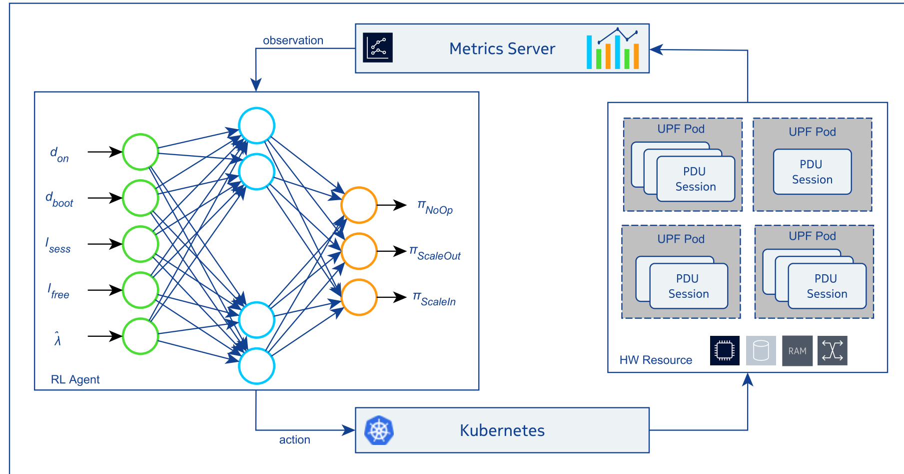


FIGURE 3. Autoscaler control loop. The metrics server collects statistics from the Pods, which are then sent to the autoscaler as an observation. The autoscaler may make a decision to scale and sends the number of replicas through the Scale interface.

denote the state, the action, and the reward at time T_i with a lower index i (e.g. $s(T_i) = s_i$).

The state s_i at time T_i should contain all the information necessary for an optimal scaling decision. In our case

$$s_i = \left\{ d_{\text{on}}(T_i), d_{\text{boot}}(T_i), l_{\text{sess}}(T_i), l_{\text{free}}(T_i), \hat{\lambda}(T_i) \right\}, \quad (2)$$

where $\hat{\lambda}$ is the measured arrival rate since the previous decision at time T_{i-1}.

The action space consists of three actions: `start` a new Pod; `terminate` an existing Pod; `no action`. The agent may only start new Pods if there is a capacity for it in the cluster, that is, $d_{\text{on}}(t) + d_{\text{boot}}(t) < D_{\text{max}}$, where $d_{\text{boot}}(t)$ is the number of Pods still booting at time t. These booting Pods exist because when we start a new Pod, it enters a pending phase while it starts up its necessary containers. We assume this phase lasts t_{pend} time. Also, the agent may only terminate Pods if $d_{\text{on}}(t) > D_{\text{min}}$. We assume this termination is graceful, which means that the Pod waits for all of its PDU sessions to close before shutting down. Obviously in this case the Pod is scheduled for termination and does not accept new PDU sessions.

The reward function is shown in (3).

$$r_i = \begin{cases} -\kappa \hat{p}_{b,i} & \text{if } \hat{p}_{b,i} > p_{b,th} \\ -d_{\text{on}}(T_i) & \text{if } \hat{p}_{b,i} \leq p_{b,th} \end{cases} \quad (3)$$

Here $\hat{p}_{b,i}$ is the measured blocking rate since the previous decision in the time interval $[T_{i-1}, T_i)$ and $p_{b,th}$ is the blocking rate threshold set by the QoS level that we should not exceed in the long term. The coefficient κ is a scalar that scales the blocking rate to numerically put it in range with the d_{on} value. The intuition behind this reward function is that if the measured blocking rate exceeds the threshold, we need to minimize the blocking rate; and if it does not exceed the threshold, we want to minimize the number of Pods.

For the list of notations used to describe the MDP see Table 1.

B. REINFORCEMENT LEARNING

Reinforcement learning (RL) is a method that applies an agent to interact with an environment. The agent observes the system states and rewards as results of subsequent actions. To apply a RL-based agent in the control loop illustrated in Figure 3, we propose an approach where a specific state contains the number of active and booting UPF Pods, the number of PDU sessions in the system, and an approximation of the arrival rate. These information about the states can be obtained either from monitoring the SMF and AMF functions of the 5G core, or from the SMF and AMF functions.

The RL agent uses the observations gathered between two scaling actions to update and improve its policy. This means that learning happens online during the operation of the cluster. Also, the neutral network in the RL agent can be pre-trained with the use of captured data and simulation as well.

The goal of RL is to find the policy π that maximizes tha value function $V^{\pi}(s)$, the long-term expected cumulated reward (4) starting from the state s. Note, that the optimal policy does not depend on the starting state.

$$V^{\pi}(s) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k r_{i+k} \middle| s_i = s \right] \quad (4)$$

In this paper we used proximal policy optimization (PPO) [38] as the RL algorithm with slight modifications, similar to our previous work [39]. The method is presented in Algorithms 1 and 2.

The PPO is an actor-critic algorithm [40]. It uses a parameterized policy $\pi(s, \boldsymbol{\theta})$ as an actor to select actions, where $\boldsymbol{\theta}$ is the parameter vector. The algorithm also approximates the value function with $V(s, \boldsymbol{\omega})$ parameterized with the $\boldsymbol{\omega}$ vector. This value function is used to calculate the advantage

$\hat{A}_{\mathrm{GAE}}$ using the generalized advantage estimator (GAE) [41]. If we consider a batch of advantages in a vector A_{GAE} of size $N_{\mathrm{batch}}+1$, the j-th element of the vector can be computed by

$$\hat{A}_{\mathrm{GAE},j} = \sum_{l=0}^{N_{\mathrm{batch}-j-1}} \lambda_{\mathrm{GAE}}^{l} \delta_{j+l} \quad j = 0, 1, \ldots, N_{\mathrm{batch}}. \tag{5}$$

Here λ_{GAE} is a weight hyperparameter which implicitly contains the discounting factor γ, and δ_j is the j-th element of the temporal difference error batch $\boldsymbol{\delta}$ and is calculated by

$$\delta_j = \mathrm{TD}_{\mathrm{target},j} - V(s_j, \boldsymbol{\omega}), \tag{6}$$

where the j-th temporal difference target is

$$\mathrm{TD}_{\mathrm{target},j} = r_j - \tilde{r} + V(s_j', \boldsymbol{\omega}). \tag{7}$$

Note, that we use bold to signify vector values and the lower index j to signify vector elements, that is, $\mathrm{TD}_{\mathrm{target},j}$, s_j, r_j, and s_j' are the j-th element of the batches TD_{target}, $\mathbf{s}$, $\mathbf{r}$, and $\mathbf{s}'$. In contrast to the original PPO algorithm, here in (7) we used the average reward scheme, where $\tilde{r}$ is the average reward that we keep track of through the soft update

$$\tilde{r} \leftarrow (1-\alpha_R)\tilde{r} + \alpha_R \sum_{j=0}^{N_{\mathrm{batch}}-1} r_j / N_{\mathrm{batch}}, \tag{8}$$

where α_R is the update rate hyperparameter.

The advantage and the TD_{target} are used to evaluate the policy. During its operation the algorithm tries to improve the policy by updating $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ repeatedly using

$$\begin{aligned} \boldsymbol{\omega} &\leftarrow \boldsymbol{\omega} + \alpha_{\omega} TD_{\mathrm{target}} \odot \nabla_{\omega} V(\mathbf{s}, \boldsymbol{\omega}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha_{\theta} \nabla_{\theta} \min\{\mathbf{r_t}(\boldsymbol{\theta}) \odot \hat{A}_{\mathrm{GAE}}, \\ &\quad \mathrm{clip}(\mathbf{r_t}(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon) \odot \hat{A}_{\mathrm{GAE}}\} \\ &\quad + \xi H(\pi(\cdot|\mathbf{s}, \boldsymbol{\theta})). \end{aligned} \tag{9}$$

In (9) the α_{ω} and α_{θ} are the learning rates of the gradient descent steps and ε is the clipping ratio of the PPO. The vector $\mathbf{r_t}(\boldsymbol{\theta})$ is the probability ratio and its j-th element can be computed by

$$r_t(\boldsymbol{\theta})_j = \pi(a_j|s_j, \boldsymbol{\theta}) / \pi_{\mathrm{old},j}, \tag{10}$$

where $\pi(a_j|s_j, \boldsymbol{\theta})$ and $\pi_{\mathrm{old,\,j}}$ are the probabilities of action a_j in state s_j. Note, that the difference between the two probabilities is that the former depends on $\boldsymbol{\theta}$ which can change throughout epochs during an update (as seen in Algorithm 1), whereas $\boldsymbol{\pi}_{\mathrm{old}}$, which is stored in the batch, represents the probability of action a_j when it was executed by the agent. This means that at the start of the update $\pi(a_j|s_j, \boldsymbol{\theta}) = \pi_{\mathrm{old},j}$, but after the first epoch $\boldsymbol{\theta}$ is changed by (9) and the equality does not hold anymore. The operation $\odot$ is the elementwise product. We also added entropy regularization

$$H(\pi(\cdot|\mathbf{s}, \boldsymbol{\theta})) = -\sum_{a' \in \mathcal{A}} \pi(a'|\mathbf{s}, \boldsymbol{\theta}) \log \pi(a'|\mathbf{s}, \boldsymbol{\theta}) \tag{11}$$

with a weight of ξ.

In Algorithm 1 we can see the Store and Update procedures of the algorithm used. The purpose of the Store function is to save the $(s_i, a_i, \pi(\cdot|s_i, \boldsymbol{\theta}), r_i, s_{i+1})$ trajectory samples into a batch for batch updates. Here $\pi(\cdot|s_i, \boldsymbol{\theta})$ denotes the vector of probabilities for all actions in state s_i.

Algorithm 1 Proximal Policy Optimization Update in the i-th Decision Epoch

1: **procedure** STORE(s_i, a_i, $\pi(\cdot|s_i, \boldsymbol{\theta})$, r_i, s_{i+1})
2: Append $(s_i, a_i, \pi(\cdot|s_i, \boldsymbol{\theta}), r_i, s_{i+1})$ to batch.
3: **end procedure**
4: **procedure** UPDATE( )
5: **if** size of the batch $< N_{\mathrm{batch}}$ **then return**
6: **end if**
7: Get batch $\mathbf{s}$, $\mathbf{a}$, $\boldsymbol{\pi}_{\mathrm{old}}$, $\mathbf{r}$, $\mathbf{s}'$ of size N_{batch}.
8: Update $\tilde{r}$ using (8).
9: **for** k epochs **do**
10: Compute $\hat{A}_{\mathrm{GAE}}$ and TD_{target} using (5–7).
11: Compute $\mathbf{r_t}(\boldsymbol{\theta})$ using (10).
12: Update $\boldsymbol{\omega}$ and $\boldsymbol{\theta}$ using (9).
13: **end for**
14: Clear batch storage.
15: **end procedure**

The Update procedure shows us the method used for improving the policy. It is only executed if the number of samples has reached N_{batch}. It approximates the mean reward $\tilde{r}$ and then it makes gradient descent steps k times. In each step it updates the policy π by updating $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$.

Algorithm 2 RL Training Loop

1: Initialize system, and get initial state s_0.
2: Initialize learning parameters of Agent.
3: $i \leftarrow 0$
4: **for** N_{train} steps **do**
5: Get action from agent: $a_i, \leftarrow$ Sample $\pi(s_i, \boldsymbol{\theta})$.
6: Execute action a_i to scale the cluster.
7: Observe the new state s_{i+1} and performance measures after ΔT time.
8: Compute reward r_i from the measurements using (3).
9: AGENT.STORE(s_i, a_i, $\pi(\cdot|s_i, \boldsymbol{\theta})$, r_i, s_{i+1})
10: AGENT.UPDATE( )
11: $i \leftarrow i+1$
12: **end for**

The procedure used to train the RL agent can be seen in Algorithm 2. First, if the system is not initialized yet, we need to start it up. For the RL agent we need to set the hyperparameters and initialize the $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ vectors with random values and set the value of $\tilde{r}$ to 0.

After the initialization we run N_{train} steps and in each step we execute a scaling action a_i received from the agent based on the observed state s_i. We observe a new state and then store the observations using the Store procedure, then improve the agent's policy with the Update procedure.

C. NEURAL NETWORK APPROXIMATION

The 5-tuple that represents the state s_i of the system creates a 5-dimensional state space. Even though in practice the values

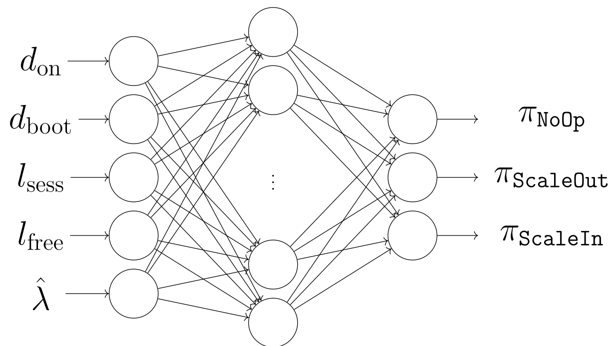


FIGURE 4. The neural network for policy π_θ. The network receives the state as an input and outputs the probabilities of each action for that given state. Connections represent the weights in θ. Nodes in the intermediate hidden layer represent the application of a non-linear activation function.

in the state are directly or indirectly bounded by the number of maximum Pods D_{max} and the arrival rate is also bounded by the maximum arrival rate $\hat{\lambda}_{max}$, the state space can grow so large that it would be impossible to fit the policy or the value function in a computer's memory. Therefore we used a neural network with one hidden layer of 50 hidden nodes to approximate the policy π and the value function V. This means that $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ represent the parameter set of these neural networks. Figure 4 shows us a neural network that accepts the state as the input and ouputs the probabilities ($\pi_{\texttt{NoOp}}$, $\pi_{\texttt{ScaleOut}}$, $\pi_{\texttt{ScaleIn}}$) of the possible actions. In the hidden layer, the rectified linear unit (ReLU) function is applied. For the policy π, we used the softmax function in the output layer. The parameters $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ were started with the Xavier initialization. For the update steps, we used the stochastic gradient descent method.

For numerical stability we normalized most of the input values into the range [0, 1]. This means, that we divided the d_{on} and d_{boot} values by D_{max} and also divided the l_{sess} and l_{free} values by L_{sess}. As for $\hat{\lambda}$, we do not have a maximum value for the arrival rate. Luckily for this normalization process we do not need to know this exact number, we only need that the order of magnitude of the *normalized* $\hat{\lambda}$ is close to the other input value's order of magnitude. We chose to divide $\hat{\lambda}$ by 500 assuming the maximum arrival rate is close to this value.

To find the best DRL agent we conducted a hyperparameter search during training. We identified the reward multiplier κ and the entropy regularization factor ξ as the hyperparameters the DRL agent was more sensitive to. We used grid search for $\kappa \in \{3, 5, 10, 13, 15, 20\}$ and $\xi \in \{0.01, 0.05\}$ to find the adequate hyperparameter values. We found the other hyperparameters to have less influence on the overall performance of the DRL agent. In these cases we used values that are often used in the literature, such as [38]. Table 2 shows us the hyperparameter values we used for the DRL agent. For the entropy parameter we chose $\xi = 0.01$ and for the reward multiplier we chose $\kappa = 13$. Note, that since we use GAE to estimate advantages, the discount factor γ is implicitly incorporated into the λ_{GAE} hyperparameter. Table 2 presents the list of hyperparameter values used for training the DRL agent.

TABLE 2. DRL hyperparameters.

neural network hidden layers	1
hidden layer node count	50
learning rate (α_θ, α_ω)	0.0001
epochs (k)	5
batch size (N_{batch})	32
reward averaging factor (α_R)	0.1
PPO clipping parameter (ϵ)	0.1
GAE parameter (λ_{GAE})	0.9
reward multiplier (κ)	3, 5, 10, 13, 15, 20
entropy regularization factor (ξ)	0.01, 0.05
random initialization	Xavier uniform
activation function	ReLU

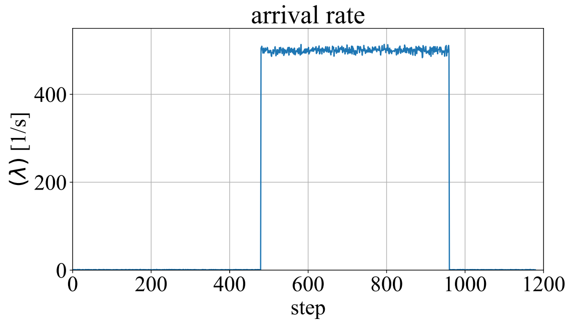
We used PyTorch 1.5.1 [42] to implement the DRL model and used an NVIDIA GeForce RTX 2070 (8GB) GPU for training.

D. DRL WITH CLASSIFICATION

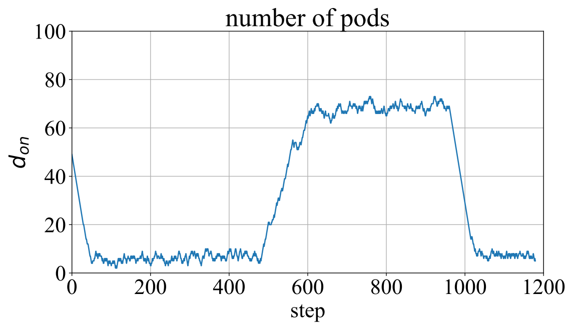
It is possible to use RL with non-stochastic policies that enforce actions with the probability equal to one for a specific observation. For example, the application of the deep Q-learning, also known as deep Q-networks (DQNs) [40], may result in a greedy policy, which is demonstrated in Section V. Moreover, we find that DQN leads to the over provisioning of the resource in our numerical study.

The PPO method learns and finds a stochastic policy where the action space has a probability distribution for a given state. The DRL agent takes action based on the learned distribution. In general, it is expected that the agent recommends the launch of new Pods when d_{on} is low and $\hat{\lambda}$ is high, and suggests the termination of Pods when d_{on} is high and $\hat{\lambda}$ is low. However, the agent may advise an unexpected action with low, but non-zero probability due to the nature of a stochastic policy. For example, when a sudden increase in traffic is detected as illustrated in Figure 5, the agent begins starting up Pods to lower the blocking rate. We would expect the agent to start new Pods until the blocking rate is below the threshold, however, from time to time the agent terminates a Pod incorrectly. If the algorithm could decrease the probability of the bad actions to 0 and increase the probability of the good action to 1 in every state, we would get a deterministic policy. However, this cannot happen, due to the entropy regularization which prevents the PPO algorithm from reducing the probability of an action to zero. This is a necessary measure to guarantee that all actions remain possible in all states so that the agent would have a possibility to explore the whole state space during training. The noisy behavior of the DRL agent can also be seen in Figure 6 that plots the action versus $\hat{\lambda}$ and d_{on}.

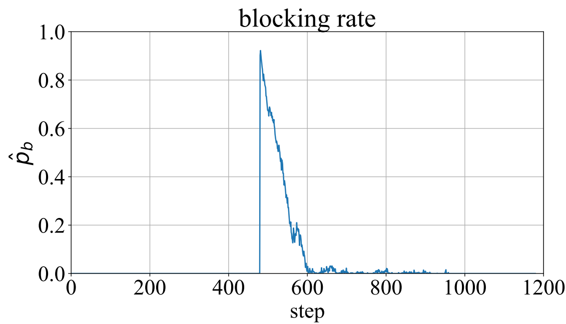
It is worth emphasizing that there may be outlier points in the dataset, e.g. where the arrival rate is very low and the Pod count is very high. If these points are labeled correctly, they do not influence the separating line. However, in case of mislabeling, these points can shift the decision boundary into an unwanted direction. Therefore we need a classifier to *clean* the dataset by removing these outlier points. We did this by



(a) At step 480 the arrival rate suddenly increases to $500\frac{1}{s}$.



(b) The DRL agent follows the traffic increase by starting new pods, but it sometimes terminates existing ones due to its stochastic policy.



(c) The blocking rate spikes along with the traffic increase but reduces quickly as new pods are started.

FIGURE 5. Trained DRL agent during sudden traffic increase. The system is initialized with 50 Pods and the DRL agent immediately starts removing unused ones. At step 480 the traffic suddenly increases and the agent reacts by increasing the Pod count. Due to the stochastic nature of the policy, a Pod may be terminated even when the blocking rate is above the threshold level.

considering every point an outlier for which

$$|d_{\text{on}}/D_{\max} - \hat{\lambda}/\hat{\lambda}_{\max}| > 0.4, \tag{12}$$

where $\hat{\lambda}_{\max}$ is the maximum of the measured arrival rate during the experiment. With this we removed every point that is not on the main diagonal strip of the scatter plot.

We apply the DRL agent to generate labels by taking a set of states and mapping actions to each state. The resulting dataset of size N_{data} was then used to create a linear support vector machine (SVM) classifier that maps actions to states.

The linear SVM is a machine learning model that can find the separating hyperplane in a dataset between two classes [43]. In our case, the set of actions $\mathcal{A}$ contains three types of actions which would require a multiclass classificator. We can circumvent this with the *one-versus-rest* strategy and build separate models for each action type. In this case we label the corrseponding action a with $+1$ if it belongs to the given action type, otherwise we label it with -1. We will denote the modified action label with $\tilde{a}$.

Using the set of states $\mathcal{S}$ as the feature set, for a state $s_i \in \mathcal{S}$ we are looking for the separating hyperplane $f(s_i) = \mathbf{w}^T s_i + w_0 = 0$, where $\mathbf{w}$ and w_0 are the parameters of the SVM. This would give us the classification rule

$$\tilde{a}_i = \text{sign}(\mathbf{w}^T s_i + w_0). \tag{13}$$

Note that multiple hyperplanes may be found. So we pick the one with the largest margin M, which is the distance between the hyperplane and the data point closest to the hyperplane.

Two cases are distinguished.

- If the data is separable, that is, a hyperplane exists that can separate the actions labeled with $+1$ from the actions labeled with -1, the optimization problem is

$$\begin{aligned} &\min_{\mathbf{w}, w_0} \quad \|\mathbf{w}\| \\ &\text{subject to} \quad \tilde{a}_i(\mathbf{w}^T s_i + w_0) \geq 1, \quad i = 1, 2, \ldots, N_{\text{data}}. \end{aligned} \tag{14}$$

- If the dataset contains overlaps and it is not separable we need to find the separating hyperplane that allows the least amount of points in the training set to be classified incorrectly. This can be achieved by introducing the slack variables ζ_i and modifying the optimization problem into

$$\begin{aligned} &\min_{\mathbf{w}, w_0} \quad \|\mathbf{w}\| \\ &\text{subject to} \quad \tilde{a}_i(\mathbf{w}^T s_i + w_0) \geq 1 - \zeta_i \\ &\qquad\qquad \zeta_i \geq 0, \quad \sum \zeta_i \leq C, \quad i=1, 2, \ldots, N_{\text{data}}, \end{aligned} \tag{15}$$

  where C, called the cost parameter, is a tuneable hyperparameter of the SVM. The smaller C is, the more points are allowed to be misclassified, resulting in a higher margin.

Algorithm 3 presents the training procedure of the SVM. The algorithm requires the parameters and the hyperparameters of the simulation environment and the DRL method. It returns the SVM model parameters $\mathbf{w}$ and w_0 and also returns the accuracy of the model on the test set which is a performance measure of the SVM.

After the initialization of the agent and the environment, the algorithm starts training the agent for N_{train} steps. This training loop is almost identical to the one in Algorithm 2. The difference is that in the last N_{data} steps the agent stores the states in a list $\mathcal{L}_{\text{states}}$ for the dataset later.

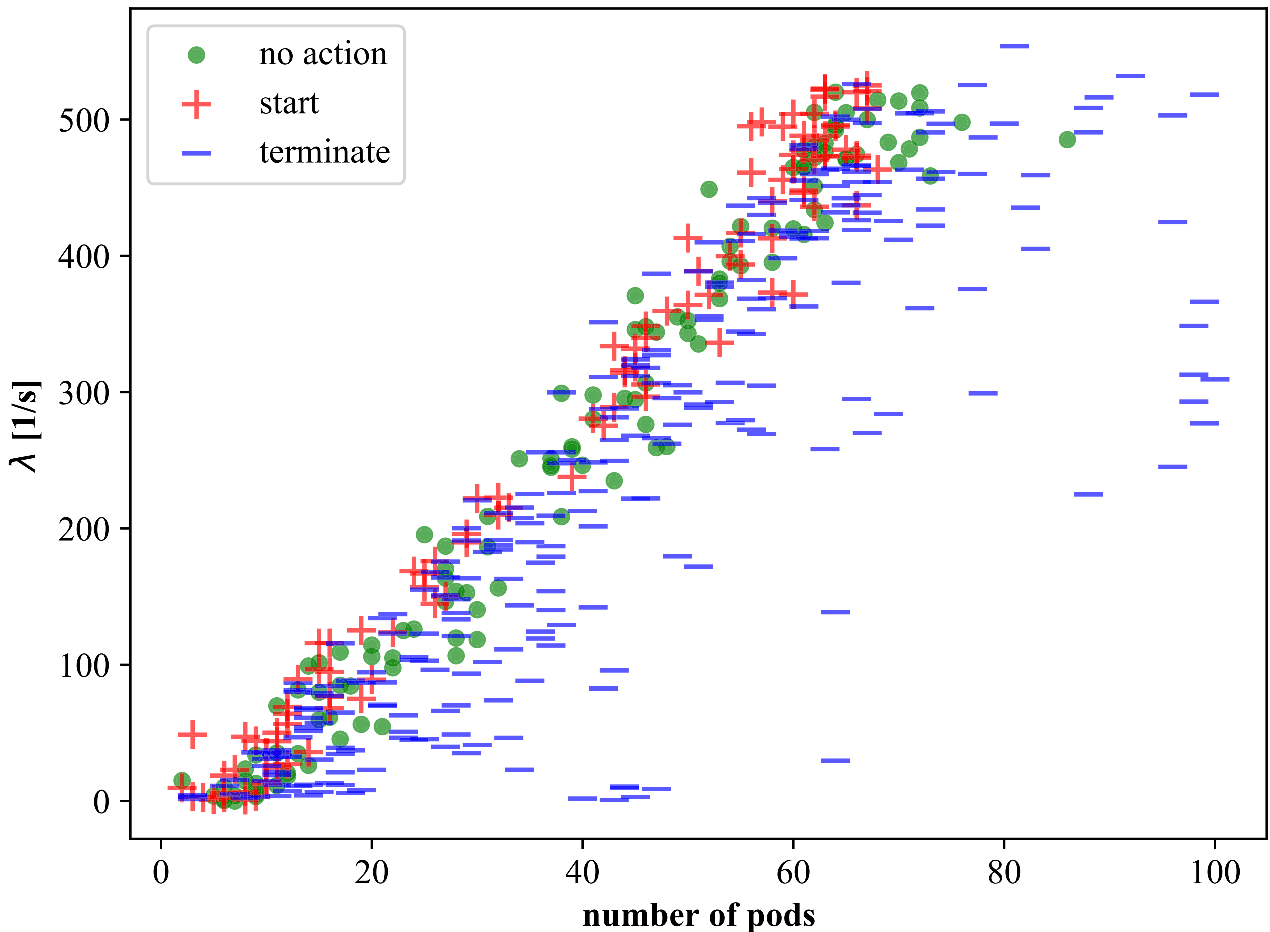


FIGURE 6. Selected actions at given $\hat{\lambda}$ and d_{on} values. We see more `terminate` actions when the arrival rate is low and the number of pods high and more `start` actions in the opposite case.

When the training of the DRL agent is finished, its policy is used to evaluate the states in $\mathcal{L}_{\text{states}}$. The resulting actions are then saved in the list $\mathcal{L}_{\text{acts}}$. These two lists together form the dataset we use to train the SVM. The dataset is cleaned by removing the outlier data points. Then it is split into training ($\mathcal{L}^{\text{train}}_{\text{states}}$, $\mathcal{L}^{\text{train}}_{\text{acts}}$) and test ($\mathcal{L}^{\text{test}}_{\text{states}}$, $\mathcal{L}^{\text{test}}_{\text{acts}}$) sets. The training set is used to train the SVM model, whereas the test set is used to determine the accuracy of the trained model. At the end of the procedure we get the SVM model parameters and the accuracy on the test set.

We run this algorithm multiple times to perform a grid search on the C hyperparameter of the SVM, which means each run we use a different C value. Finally, we pick the model with the highest accuracy on the test sets.

In order to asses the SVM classfier, we also experimented with another classification method, logistic regression which describes the log-odds of each class with a linear function. For more on this classifier, see [43]. In this case, Algorithm 3 can be modified by replacing the SVM model with a logistic regression model.

We used the scikit-learn 0.24.2 [44] library to implement the SVM and the logistic regression models. For the logistic regression, we used the default hyperparameters. For the SVM hyperparameter values see Section V-B. For the list of notations used by the algorithms see Table 3.

E. SYSTEM MODELING

We built a simulator program that emulates a multi-node cloud environment and implemented the DRL agent in Python with the help of pytorch. The simulator program contains a procedure that generates the arrival of a UE as a Poisson process with arrival rate $\lambda(t)$ at time t. Upon arrival a PDU session is initiated if there is available capacity among the pods. Otherwise the UE's request is blocked. The UPF handling the PDU session and its traffic is chosen at random. We assume the length of a session is random and distributed exponentially with rate μ.

Note that in practice we do not know the exact arrival rate function in advance. To show how the DRL algorithm can cope with this, we divided the DRL experiments

Algorithm 3 Training an SVM Classifier

Input: Environment and DRL parameters (see Table 1 and 3).

Output: SVM model ($\mathbf{w}$, w_0) and its accuracy

1: Initialize Pod count to D_{min}.
2: Initialize $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ of the Agent with random values.
3: $\mathcal{L}_{\text{states}} \leftarrow \{\emptyset\}$, $\mathcal{L}_{\text{acts}} \leftarrow \{\emptyset\}$
4: *// Training agent and collecting states.*
5: $i \leftarrow 0$
6: **for** N_{train} steps **do**
7: $a_i \leftarrow$ Get action from the agent in state s_i (based on the neural network).
8: r_i, $s_{i+1} \leftarrow$ Execute action a_i and get reward and the next state.
9: Store history and Update agent using the AGENT.STORE and AGENT.UPDATE procedures in Algorithm 1.
10: **if** $i > N_{\text{train}} - N_{\text{data}}$ **then**
11: Append state to $\mathcal{L}_{\text{states}}$.
12: **end if**
13: $i \leftarrow i + 1$
14: **end for**
15: $i \leftarrow 0$
16: **for** N_{data} steps **do**
17: Evaluate DRL agent on state $\mathcal{L}_{\text{states}}[i]$ to get action.
18: Append action to $\mathcal{L}_{\text{acts}}$.
19: $i \leftarrow i + 1$
20: **end for**
21: Remove outlier points according to (12).
22: Separate lists into train and test sets: $\mathcal{L}_{\text{states}} \rightarrow \mathcal{L}^{\text{train}}_{\text{states}}, \mathcal{L}^{\text{test}}_{\text{states}}$; $\mathcal{L}_{\text{acts}} \rightarrow \mathcal{L}^{\text{train}}_{\text{acts}}, \mathcal{L}^{\text{test}}_{\text{acts}}$.
23: $\mathbf{w}$, $w_0 \leftarrow$ Train SVM using $\mathcal{L}^{\text{train}}_{\text{states}}$ as features and $\mathcal{L}^{\text{train}}_{\text{acts}}$ as labels and run a grid search on hyperparameter C.
24: Get accuracy of the model using $\mathcal{L}^{\text{test}}_{\text{states}}$ and $\mathcal{L}^{\text{test}}_{\text{acts}}$.
25: **return** $\mathbf{w}$, w_0, accuracy

into two phases, a training phase and an evaluation phase. In each of these phases we used a different function for the arrival rate, λ_{train} and λ_{eval}, respectively. We can think of the training phase as a pre-training stage where we initialize the DRL agent and train it with a predefined arrival rate function. Whereas in the evaluation phase we apply the pre-trained agent on an environment with a new traffic model. So learning also happens in the evaluation phase, but the agent does not need to go through a cold start.

We trained the DRL agent on a sinusoidally varying arrival rate

$$\lambda_{\text{train}}(t) = 250 + 250 \sin\left(\frac{\pi}{6}t\right) \tag{16}$$

for N_{train} amount of simulation steps. With this function the agent can explore a wide range of traffic intensity. For evaluation we used an equation from [45] which was determined for mobile user traffic. We scaled it

TABLE 3. Notations used by the algorithms.

Notations for the HPA	
$\bar{\rho}$	mean utilization (e.g. CPU)
ρ_{target}	target utilization of the HPA algorithm
ν	tolerance of the HPA algorithm
d_{desired}	desired Pod count provided by the HPA
Notations for the DRL	
$\hat{\lambda}_{\text{max}}$	maximum measured arrival rate
λ_{train}, λ_{eval}	training and evaluation phase arrival rate functions
N_{train}, N_{eval}	number of training and evaluation steps
$\boldsymbol{\theta}$, $\boldsymbol{\omega}$	policy and value network parameter vector
N_{batch}	batch size
$\bar{r}$	mean reward
α_R	mean reward soft udpate rate
k	number of epochs in PPO
$\text{TD}_{\text{target}}$	temporal difference target
δ	TD-error
A, $\hat{A}$	advantage, estimate of the advantage
α_θ, α_ω	learning rate of the policy and the value network
ϵ	PPO clipping parameter
λ_{GAE}	GAE parameter
H	entropy regularization function
ξ	entropy coefficient
Notations for the SVM	
N_{data}	size of the dataset used for tarining the SVM
$\mathcal{L}^{\text{train/test}}_{\text{states/acts}}$	list containing the training/test set of states/actions
$\mathbf{w}$, w_0	SVM parameters
ζ_i	SVM slack variable
C	margin parameter of the SVM

to our use case to get

$$\begin{aligned}\lambda_{\text{eval}}(t) = 330.07620 + 171.10476 \sin\left(\frac{\pi}{12}t + 3.08\right) \\ + 100.19048 \sin\left(\frac{\pi}{6}t + 2.08\right) \\ + 31.77143 \sin\left(\frac{\pi}{4}t + 1.14\right)\end{aligned} \tag{17}$$

and ran N_{eval} amount of simulation steps with it. This scaling makes the peak traffic 500 PDU requests/s. To visualize λ_{train} and λ_{eval} in (16) and (17), we plotted Figures 7a and 7b that demonstrate the arrival rates measured ($\hat{\lambda}$) during the training phase and the evaluation phase for a 36 hour time period.

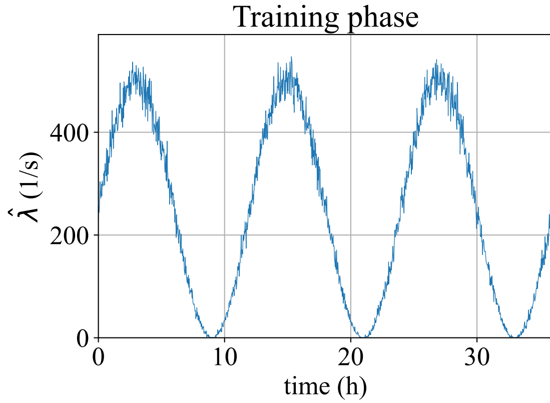
We set the blocking threshold $p_{b,th} = 0.01$ and ran the DRL algorithm under various t_{pend} values. For each t_{pend} value we ran 8 simulations and took the average of d_{on}, and $\hat{p}_b$ for the evaluations phase.

V. NUMERICAL RESULTS

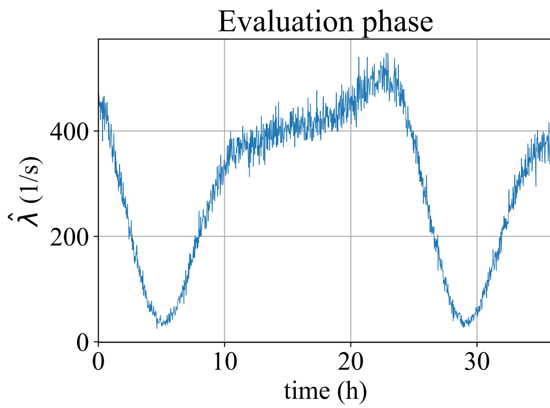
A. SCENARIOS

For the numerical evaluations, we assumed that

- UPF instances run in phyiscal servers [46] with the Intel Xeon 6238R 28 core 2.2 GHz processor and 4 × 64 GB RAM;
- each UPF session conveys video streaming data;
- eight cores on each server are allocated for OS and the container management system;
- Each UPF instance occupies one core and 2GB RAM and serve maximum 8 simultaneous video streams;
- booting time is not negligible and is fixed and identical for each UPF pod.

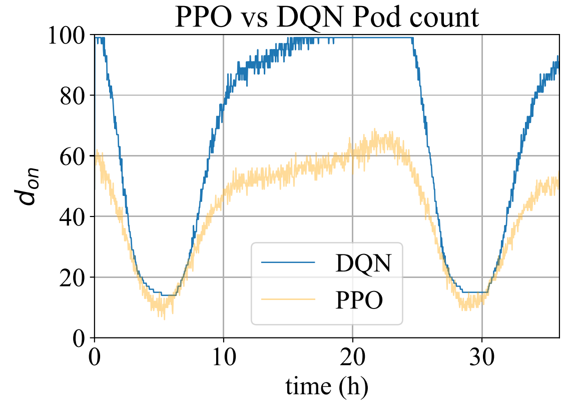


(a) The measured arrival rate $\hat{\lambda}$ during the training phase when traffic was generated according to λ_{train} in (16).

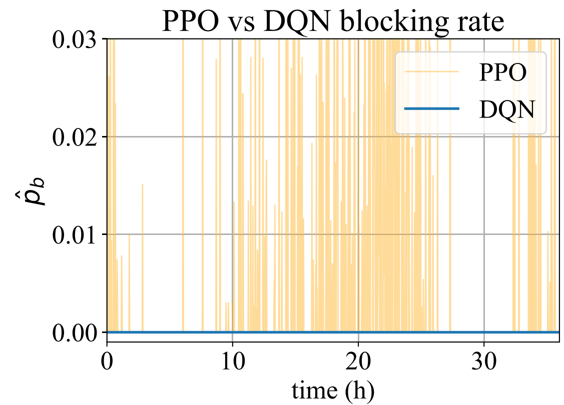


(b) The measured arrival rate $\hat{\lambda}$ during the evaluation phase when traffic was generated according to λ_{eval} in (17).

FIGURE 7. Arrival rates measured during the training and the evaluation phase for a 36 hour period.



(a) Number of Pods (d_{on}) during the evaluation phase, using DQN or PPO. We can see that the Pod count follows the traffic change showed in Figure 7b. However, the policy learned by the DQN does not perform as well as the PPO's policy as it starts way more Pods at higher traffic.



(b) Blocking rate ($\hat{p}_b$) during the evaluation phase, using DQN or PPO. The blocking rate occasionally breaks the $p_{b,th} = 0.01$ threshold with the PPO agent when traffic is high, but the long time average is still below the threshold. In the case of the DQN the blocking rate is zero.

FIGURE 8. Performance measures. DQN starts a lot more Pods during high traffic (the tendency is to use the maximum Pod count $D_{\text{max}} = 100$). Meanwhile the blocking rate with DQN is always zero which is a clear sign of overprovisioning.

TABLE 4. Cloud environment parameters.

Parameter	Value
the max. number of sessions per Pod (B)	8
service rate (μ)	1/s
minimum number of Pods (D_{min})	2
maximum number of Pods (D_{max})	100
initialization time (t_{pend})	0.25, 5, 10 s
time between decisions (ΔT)	1 s
blocking rate threshold ($p_{b,th}$)	0.01

Parameter values for the cluster used during the simulations can be found in Table 4.

Besides the PPO algorithm, we also experimented with deep Q-networks (DQNs) which use a deterministic, greedy policy in the evaluation phase. We found that DQN had difficulties learning the optimal policy. Figure 8 shows us a comparison between the evaluation of a trained DQN and a trained PPO agent. We can see that while the PPO could adapt well to the varying arrival rate in (17) and found a balance between the Pod count and the blocking rate, the DQN could not keep the Pod count as low and overprovisioned the Pods. We also have to note that the DQN seemed to be much more unstable as in many cases it failed to even learn a policy that would adapt to traffic changes, whereas the PPO algorithm could find a good policy throughout every run. For this reason we ruled out the use of DQN and focused on PPO instead.

Results for the PPO are included in Table 5. The values displayed are averaged through 8 runs. Each run took approximately 120–150 minutes. We can see that the DRL agent could keep the blocking rate $\hat{p}_b$ below the threshold 0.01 in each case.

For the HPA algorithm, we assumed that handling a CPU session requires 100% of a CPU core. This means that the utilization of a UPF Pod is proportional to the number of PDU sessions it handles. We set the tolerance

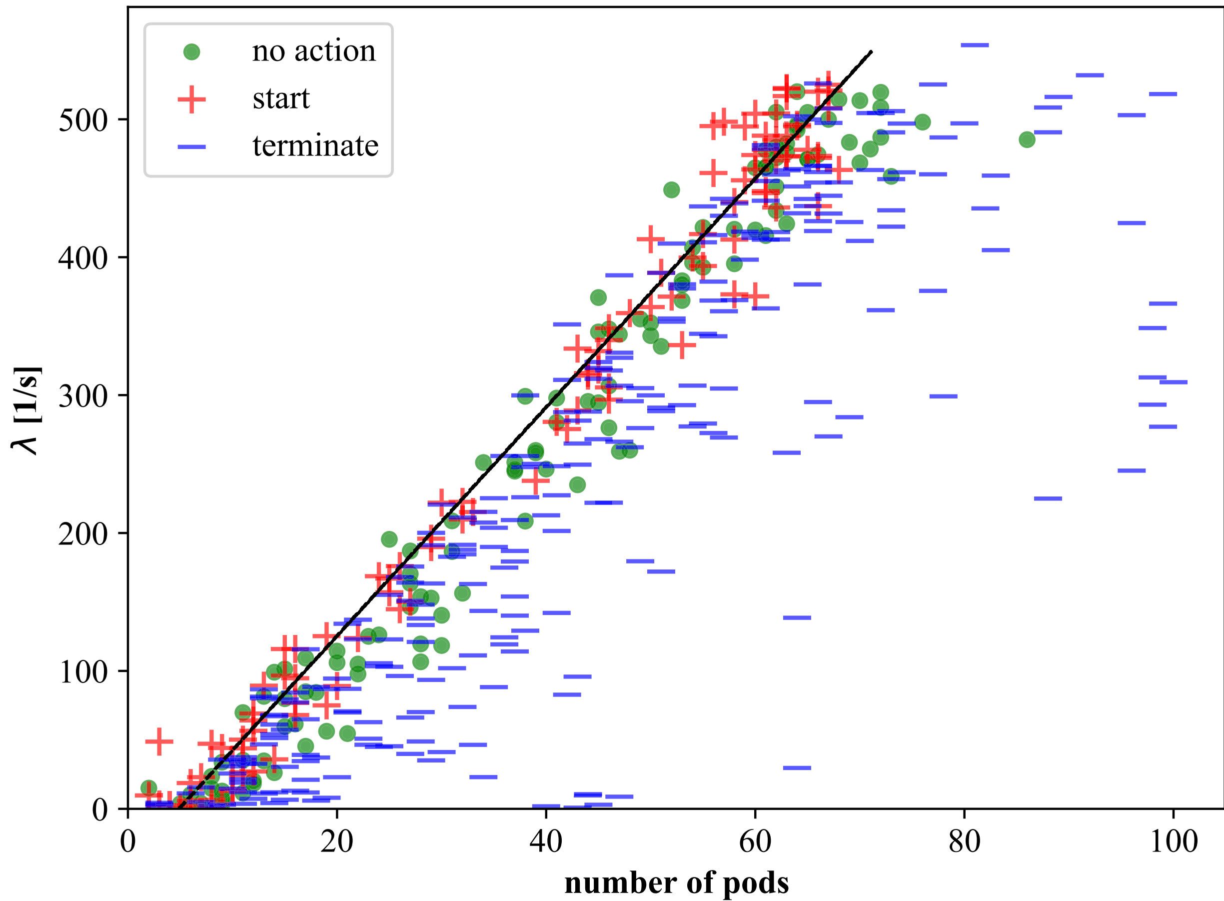


FIGURE 9. Decision boundary learned by the SVM classifier. The agent terminates a Pod if it is in a state below the boundary line and starts a new Pod if it is above.

$\nu = 0.025$ and searched for the parameter ρ_{target} that could still maintain the blocking rate below the threshold. Each run consisted of N_{eval} evaluation steps. The range of the search was $(6.0, 6.25, \ldots, 9.75)$ and the result was $\rho_{\text{target}} = 0.875$. We included the results in Table 5.

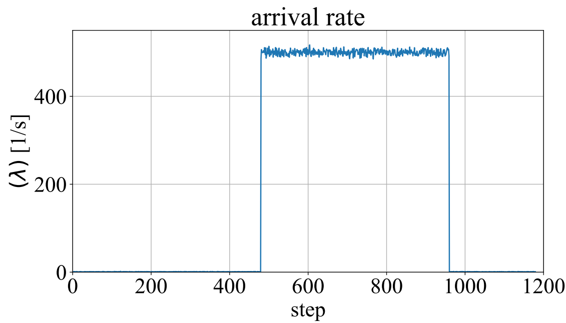
Comparing the results in Table 5 we can see that at lower t_{pend} values the DRL agent could maintain fewer UPF Pods and the d_{on} was reduced by 3.8%. At higher t_{pend} values this improvement percentage decreased but still, the DRL agent was more efficient in using the UPF Pods. The reason for the decrease is that when t_{pend} is high, it takes much longer for a Pod to start, which means that in order to keep $\hat{p}_b$ below the threshold, the DRL agent cannot terminate that many idle Pods.

B. IMPROVING PERFORMANCE WITH ACTION CLASSIFICATION

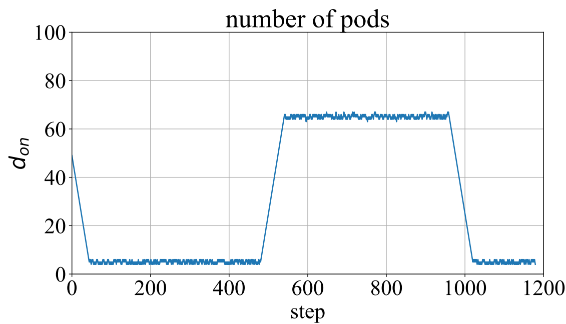
In a given state there is a small probability, that the DRL agent makes a bad decision. For example, Figures 5 and 6 show a scenario where the system is initialized with 50 Pods and the DRL agent immediately starts removing unused ones. At step 480 the traffic suddenly increases and the agent reacts by increasing the Pod count. Due to the stochastic nature of the policy, a Pod may be terminated even when the blocking rate is above the threshold level. Note that such actions could be beneficial during training because the agent should explore the state space. Therefore, we provide an approach to minimize such unwanted actions. We took the sample of states and used the actions as labels to create a dataset. Using this dataset we trained an SVM classifier with linear kernel. We also considered other kernel types such as polynomial or radial basis function kernels but found their training times significantly longer and also less accurate than the linear kernel. Algorithm 3 presents the procedure we used for training. We set the size of the dataset $N_{\text{data}} = 450000$ and used 80% of the data for training and 20% of the data for testing. For the hyperparameter search we considered $C \in \{0.1, 1, 10, 40, 100\}$.

TABLE 5. Mean number of Pods and average blocking rate with the DRL and the HPA algorithms. Percentages show improvement compared to the HPA algorithm.

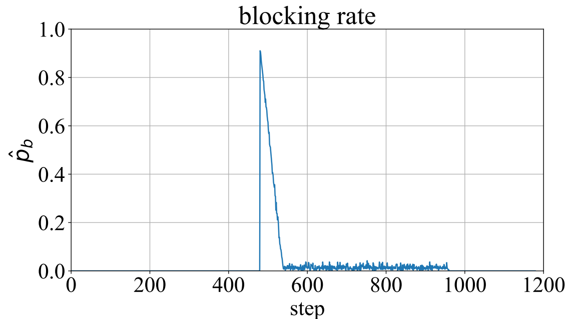
t_{pend}	d_{on} (DRL)	$\hat{p}_b$ (DRL)	d_{on} (HPA)	$\hat{p}_b$ (HPA)
.25 s	46.598 (3.8%)	0.009338	48.446	0.006300
5 s	46.908 (3.2%)	0.009125	48.452	0.007300
10 s	47.129 (2.7%)	0.008800	48.456	0.007425



(a) The mean arrival rate increases to $500\frac{1}{s}$ at timestep 480 and drops to $1\frac{1}{s}$ at timestep 960.



(b) d_{on} starts at 50 which drops due to unused Pods. Then it increases steadily when the traffic increases. During this period, Pods are not terminated unlike under the DRL agent due to the deterministic policy.



(c) The blocking rate starts dropping as the number of Pods increases.

FIGURE 10. **SVM trained agent during sudden increase in traffic. The agent uses a deterministic policy and does not terminate Pods, when they are needed.**

Besides the full state description, we also considered a reduced state description where $s(t) = \left\{d_{\text{on}}(t), \hat{\lambda}(t)\right\}$ to alleviate the curse of dimensionality during training of the SVM. With the full state space, training took approximately 12 minutes, whereas with the reduced state space, the training time was about 9 minutes.

Figure 9 shows the decision boundary learned by the SVM classifier and Figure 10 plots the behavior of the agent using the SVM classifier for the sudden increase of traffic. We can see that the agent behaves in a more consistent way than PPO and starts new Pods while the number of the Pods is not high enough to meet the blocking rate criterium.

TABLE 6. **Mean number of Pods and average blocking rate with the SVM classifier. Percentages show improvement (decrease of d_{on}) compared to the HPA algorithm at the same t_{pend} value.**

t_{pend}	d_{on}	$\hat{p}_b$
0.25	46.442775 (4.1%)	0.010400
5	48.425250 (0.1%)	0.003250
10	47.959200 (1.0%)	0.004013

TABLE 7. **Mean number of Pods and average blocking rate with the logistic regression classifier. Percentages show improvement (decrease of d_{on}) compared to the HPA algorithm at the same t_{pend} value.**

t_{pend}	d_{on}	$\hat{p}_b$
0.25	46.990392 (3.0%)	0.005427
5	48.493830 (−0.1%)	0.002906
10	48.213733 (0.5%)	0.003216

We ran Algorithm 3 for various t_{pend} values. Each experiment was carried out 8 times and we took the average of the mean number of Pods and the mean blocking rate throughout the runs. Table 6 shows us the results. We can see that by using the SVM classifier we could still keep the blocking rate below the $p_{b,th} = 0.01$ threshold. However, we get slightly higher mean pod counts than when using the DRL agent only.

To investigate the choice of a classification model, we conducted experiments with a logistic regression model as a classifier. In Table 7 results show that the logistic regression classifier could also keep $\hat{p}_b$ below the threshold $p_{b,th} = 0.01$. Incorporating the classifiers could save the resource usage (the mean number of Pods) compared to the HPA algorithm. Furthermore, the SVM model outperforms the logistic regression one when both the models are trained using the same generated dataset. The decrease in d_{on} was lower with the logistic regression at higher t_{pend}.

VI. DISCUSSIONS AND CONCLUSION

We have investigated the autoscaling of UPF Pods in a 5G core running inside the Kubernetes container-orchestration environment. An extensive numerical study shows that the application of deep Q-networks (DQNs) results in a greedy policy. Therefore, we proposed a DRL based on the PPO method to find a stochastic policy. We have shown that DRL can outperform the built-in HPA algorithm.

Note that the DRL agent may recommend an unexpected action with a tiny probability due to the nature of a stochastic policy. Such unexpected actions with the low probability value are due to outlier points in a dataset collected during the training, which drives the agent in an unwanted direction. Therefore, we need a classifier to *clean* the dataset by removing these outlier points. A study shows that the incorporation of the classifier could save the resource usage (the mean number of Pods) compared to the HPA algorithm. We have also investigated two classification models (the logistic regression model and SVM) and found that the SVM model outperforms the logistic regression one when both the models are trained using the same generated dataset. We have shown that our approach outperforms Kubernetes's built-in Horizontal Pod Autoscaler (HPA). DRL could save 2.7–3.8% of the average number of Pods, while SVM could achieve 0.7–4.5% saving

compared to HPA. It is worth emphasizing that training a classifier can create a deterministic policy that reacts better to sudden changes in traffic. In exchange, the performance degraded compared to the DRL agent when we evaluated it in an environment with slower traffic change. The degradation was slight, though, and the performance was still better than with the HPA most of the time. One major drawback, however is that Algorithm 3 cannot be run online. Therefore, it would be applied when the DRL policy is stable enough with available datasets. Otherwise, the DRL with PPO is suggested.

ACKNOWLEDGMENT

The authors appreciated the reviewers' constructive comments for improving the presentation.

REFERENCES

[1] *View on 5G Architecture*, Standard 22.891, Version 14.2.0, 5GPP, 5GPPP Architecture Working Group, Jul. 2018.
[2] *5G; Study on Scenarios and Requirements for Next Generation Access Technologies*, Standard (TS) 38.913, Version 16.0.0 Release 16, 3rd Generation Partnership Project (3GPP), Technical Specification, Jul. 2020.
[3] *Technical Specification Group Services and System Aspects;System architecture for the 5G System (5GS); Stage 2*, Standard (TS) 23.501, 3rd Generation Partnership Project (3GPP), Technical Specification, Version 16.7.0, Dec. 2020.
[4] D. Chandramouli, R. Liebhart, and J. Pirskanen, *5G for the Connected World*. Hoboken, NJ, USA: Wiley, 2019.
[5] A. Osseiran, J. F. Monserrat, and P. Marsch, *5G Mobile and Wireless Communications Technology*, 1st ed. Cambridge, U.K.: Cambridge Univ. Press, 2016.
[6] K. B. Letaief, W. Chen, Y. Shi, J. Zhang, and Y.-J.-A. Zhang, "The roadmap to 6G: AI empowered wireless networks," *IEEE Commun. Mag.*, vol. 57, no. 8, pp. 84–90, Aug. 2019.
[7] K. Samdanis and T. Taleb, "The road beyond 5G: A vision and insight of the key technologies," *IEEE Netw.*, vol. 34, no. 2, pp. 135–141, Mar./Apr. 2020.
[8] V. Ziegler, H. Viswanathan, H. Flinck, M. Hoffmann, V. Raisanen, and K. Hatonen, "6G architecture to connect the worlds," *IEEE Access*, vol. 8, pp. 173508–173520, 2020.
[9] C. Rotter and T. Van Do, "A queueing model for threshold-based scaling of UPF instances in 5G core," *IEEE Access*, vol. 9, pp. 81443–81453, 2021, doi: 10.1109/ACCESS.2021.3085955.
[10] CloudiNfrastructure Telco Task Force. (2021). *Anuket*. Accessed: Jun. 10, 2021. [Online]. Available: https://github.com/cntt-n/CNTT
[11] OpenStack Foundation and Community. (2021). *OpenStack: Open Source Cloud Computing Infrastructure*. Accessed: Jun. 10, 2021. [Online]. Available: https://www.openstack.org/
[12] Cloud Native Computing Foundation. (2021). *Kubernetes*. Accessed: Jun. 10, 2021. [Online]. Available: https://kubernetes.io/
[13] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, Dec. 2014.
[14] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Gener. Comput. Syst.*, vol. 98, pp. 672–681, Sep. 2019, doi: 10.1016/j.future.2018.09.009.
[15] I. Gervasio, K. Castro, and A. P. F. Araujo, "A hybrid automatic elasticity solution for the IaaS layer based on dynamic thresholds and time series," in *Proc. 15th Iberian Conf. Inf. Syst. Technol. (CISTI)*, Jun. 2020, pp. 1–6.
[16] Q. Z. Ullah, G. M. Khan, and S. Hassan, "Cloud infrastructure estimation and auto-scaling using recurrent Cartesian genetic programming-based ANN," *IEEE Access*, vol. 8, pp. 17965–17985, 2020.
[17] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in Kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, Aug. 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/16/4621
[18] E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," *Cluster Comput.*, vol. 22, no. 3, pp. 995–1006, Jan. 2019, doi: 10.1007/s10586-018-02890-1.
[19] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-learning," in *Proc. IEEE 6th Int. Conf. Future Internet Things Cloud (FiCloud)*, Aug. 2018, pp. 85–92.
[20] R. Shaw, E. Howley, and E. Barrett, "Applying reinforcement learning towards automating energy efficient virtual machine consolidation in cloud data centers," *Inf. Syst.*, Jan. 2021, Art. no. 101722. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030643792100003X
[21] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. G. Garino, "Reinforcement learning-based application autoscaling in the cloud: A survey," *Eng. Appl. Artif. Intell.*, vol. 102, Jun. 2021, Art. no. 104288. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197621001354
[22] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 329–338.
[23] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, Oct. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17326821
[24] L. Schuler, S. Jamil, and N. Kuhl, "AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *Proc. IEEE/ACM 21st Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2021, pp. 804–811, doi: 10.1109/CCGrid51090.2021.00098.
[25] G. Jaro, A. Hilt, L. Nagy, M. A. Tundik, and J. Varga, "Evolution towards telco-cloud: Reflections on dimensioning, availability and operability: (Invited paper)," in *Proc. 42nd Int. Conf. Telecommun. Signal Process. (TSP)*, Jul. 2019, pp. 1–8.
[26] H. Tang, D. Zhou, and D. Chen, "Dynamic network function instance scaling based on traffic forecasting and VNF placement in operator data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 3, pp. 530–543, 2019.
[27] I. Alawe, A. Ksentini, Y. Hadjadj-Aoul, and P. Bertin, "Improving traffic forecasting for 5G core network scalability: A machine learning approach," *IEEE/ACM Trans. Netw.*, vol. 32, no. 6, pp. 42–49, Nov./Dec. 2018.
[28] T. Subramanya, D. Harutyunyan, and R. Riggio, "Machine learning-driven service function chain placement and scaling in MEC-enabled 5G networks," *Comput. Netw.*, vol. 166, Jan. 2020, Art. no. 106980. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128619310254
[29] D. Kumar, S. Chakrabarti, A. S. Rajan, and J. Huang, "Scaling telecom core network functions in public cloud infrastructure," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2020, pp. 9–16.
[30] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Providing performance guarantees for cloud-deployed applications," *IEEE Trans. Cloud Comput.*, vol. 8, no. 1, pp. 269–281, Jan. 2020, doi: 10.1109/TCC.2017.2771402.
[31] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019, doi: 10.1093/comjnl/bxy043.
[32] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in Anna," *VLDB J.*, vol. 30, no. 1, pp. 25–43, Sep. 2020, doi: 10.1007/s00778-020-00632-7.
[33] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 217–228, doi: 10.1145/2950290.2950328.
[34] *5G; Technical Specifications and Technical Reports for a 5G Based 3GPP System*, (TS) 23.205, Version 16.0.0, 3GPP, 3rd Generation Partnership Project (3GPP), Technical Specification, Aug. 2020. [Online]. Available: https://www.3gpp.org/ftp//Specs/archive/23_series/23.501/23501-g70.zip
[35] *Study on Integrated Access and Backhaul*, Standard 38.874, Version 16.0.0, 3rd Generation Partnership Project (3GPP), Technical Specification Group Radio Access Network, Dec. 2018.
[36] S. Rommer, P. Hedman, M. Olsson, L. Frid, S. Sultana, and C. Mulligan, *5G Core Networks: Powering Digitalization*. New York, NY, USA: Academic, 2019.
[37] *Kubernetes Documentation*. Accessed: Aug. 6, 2021. [Online]. Available: https://kubernetes.io/docs/home/
[38] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[39] H. T. Nguyen, T. V. Do, and C. Rotter, "Optimizing the resource usage of actor-based systems," *J. Netw. Comput. Appl.*, vol. 190, Sep. 2021, Art. no. 103143. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521001594

[40] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[41] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *Proc. 4th Int. Conf. Learn. Represent., (ICLR)*, Y. Bengio and Y. LeCun, Eds. San Juan, Puerto Rico, May 2016, pp. 1–14.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and A. Desmaison, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-pe%rformance-deep-learning-library.pdf

[43] T. Hastie, R. Tibshirani, and J. H. Friedman, "The elements of statistical learning," in *Springer Series in Statistics*. New York, NY, USA: Springer, 2009.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, A. Müller, J. Nothman, G. Louppe, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2012.

[45] S. Wang, X. Zhang, J. Zhang, J. Feng, W. Wang, and K. Xin, "An approach for spatial-temporal traffic modeling in mobile cellular networks," in *Proc. 27th Int. Teletraffic Congr.*, Sep. 2015, pp. 203–209.

[46] Nokia. *AirFrame Open Edge Server*. Accessed: Mar. 26, 2020. [Online]. Available: https://www.nokia.com/networks/products/airframe-open-edge-server/

HAI T. NGUYEN received the B.Sc. and M.Sc. degrees in electrical engineering from the Budapest University of Technology and Economics, Budapest, Hungary, in 2014 and 2016, respectively. His research interests include machine learning and computer networks.

TIEN VAN DO received the M.Sc. and Ph.D. degrees in telecommunications engineering from the Technical University of Budapest, Hungary, in 1991 and 1996, respectively. He habilitated at the Budapest University of Technology and Economics (BME), and obtained the D.Sc. title from the Hungarian Academy of Sciences, in 2011. He is currently a Professor with the Department of Networked Systems and Services, Budapest University of Technology and Economics. He led various projects on network planning and software implementations that results are directly used for industry, such as ATM & IP network planning software for Hungarian Telekom, GGSN tester for Nokia, performance testing program for the performance testing of the NOKIA IMS product, automatic software testing framework for Nokia Siemens Networks. His research interests include queuing theory, telecommunication networks, cloud computing, performance evaluation and planning of ICT systems, and machine learning.

CSABA ROTTER received the M.Sc. degree in applied electronics from the Technical University of Oradea, in 1995, and the M.Sc. degree in IT management from Central European University, Budapest, in 2008. In 1999, he joined Nokia, later Nokia Research Center (currently Nokia Bell Labs), in 2008. He is heading the "Multi Cloud Orchestration" Research Department, Nokia Bell Labs. He is involved in cloud-related research topics targeting cloud-native network service operation challenges in a highly distributed multivendor environment. His particular interest is in developing solutions for the performance-related service level agreement of concurrent applications sharing the resources in distributed environments. His passion for automation started years before when he was responsible for test automation concept development in large telecommunication systems.