Demand-Driven Deep Reinforcement Learning for Scalable Fog and Service Placement

Hani Sami*, Azzam Mourad†, Hadi Otrok‡, Jamal Bentahar*
*Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada
†Department of Computer science and Mathematics, Lebanese American University, Beirut, Lebanon
‡Center of Cyber-Physical Systems (C2PS), Department of EECS, Khalifa University, Abu Dhabi, UAE

Abstract—The increasing number of Internet of Things (IoT) devices necessitates the need for a more substantial fog computing infrastructure to support the users' demand for services. In this context, the placement problem consists of selecting fog resources and mapping services to these resources. This problem is particularly challenging due to the dynamic changes in both users' demand and available fog resources. Existing solutions utilize on-demand fog formation and periodic container placement using heuristics due to the NP-hardness of the problem. Unfortunately, constant updates of services are time consuming in terms of environment setup, especially when required services and available fog nodes are changing. Therefore, due to the need for fast and proactive service updates to meet users' demand, and the complexity of the container placement problem, we propose in this paper a Deep Reinforcement Learning (DRL) solution, named Intelligent Fog and Service Placement (IFSP), to perform instantaneous placement decisions proactively. By proactively, we mean making placement decisions before demands occur. The DRL-based IFSP is developed through a scalable Markov Decision Process (MDP) design. To address the long learning time for DRL to converge, and the high volume of errors needed to explore, we also propose a novel end-to-end architecture utilizing a service scheduler and a bootstrapper. on the cloud. Our scheduler and bootstrapper perform offline learning on users' demand recorded in server logs. Through experiments and simulations performed on the NASA server logs and Google Cluster Trace datasets, we explore the ability of IFSP to perform efficient placement and overcome the above mentioned DRL limitations. We also show the ability of IFSP to adapt to changes in the environment and improve the Quality of Service (QoS) compared to state-of-the-art-heuristic and DRL solutions.

Index Terms—Fog Computing, Deep Reinforcement Learning (DRL), Internet of Things (IoT), On-Demand Fog Placement, Boostraper.

1 Introduction

1.1 Background

The rise in the number of connected things or IoT devices is accompanied by an increase in the demand for cloud services [1]. Due to transmission delay on the cloud and the possibility of network congestion, the fog computing concept was invented as a solution. Fog nodes extend the cloud by hosting services closer to IoT devices [2]. In this context, service providers are limited to hosting a limited number of services on fog nodes due to the small number of available resources [3] [4]. In existing fog solutions, services are pre-installed on fogs to serve only specific applications [5]. Thus, fog resources are not fully optimized to update or host new services. As a solution to this problem, we proposed in our previous work the formation of on-demand fog clusters by utilizing volunteering devices to host the fog services [6]. On-demand fog formation provides the flexibility to re-utilize resources by replacing idle or less requested services on the fly with new ones. The flexibility of dynamically pushing lightweight services on various types of operating systems is achieved using the Docker containerization technology [7]. In addition, orchestration technologies are applied on cluster heads like Kubernetes [8] for management purposes. Dynamic formation of on-demand fog clusters introduces the service placement problem. This problem is divided into fog selection and service assignment or placement. Fog selection entails choosing the best fog from a set of available ones, whereas service placement is the action of assigning services to selected fogs. In the sequel, we provide a use case to illustrate the main requirements of the service placement problem, and then we highlight the limitations of existing solutions in the literature.

1.2 Motivational Use Case

We consider the use case of a road with groups of autonomous vehicles performing self-driving, in addition to unmanned aerial vehicles (UAVs). Drivers and passengers are requesting different types of services using existing network protocols [9]. In particular, some drivers are requesting a service to retrieve real-time traffic information [10]. A smart vehicle is requesting services to collect more sensed data from vehicles around to improve its driving decisions [11]. Furthermore, the UAVs are requesting network management services [12], [13]. The passengers from their sides are interested in infotainment-related services such as video streaming and playing video games. Due to the limited computing resources on On-Boarding Units of the vehicles, on-demand fog computing is employed to improve the QoS experienced by the requesters. In this case, the fog computing cluster is initialized on volunteering edge servers, such as the Road Side Units (RSUs). In the on-demand fog context, services are not always lightweight. For example, downloading a guest operating system for hosting the traffic measurement service takes time. Besides, initializing the Kubernetes cluster and downloading the required modules are also time-consuming. In addition, the moving vehicles and UAVs can leave the range of the serving RSU, leading to networking delays and reachability issues. In order to overcome this problem, proactive placement is required.

Proactive placement can be performed by predicting the pattern of demands on each road by the RSUs. Furthermore, vehicles send different volumes of requests for services. Hence, a dynamic placement of services on the nearest RSU is required to meet the users' demand. Besides, the number of available resources running on the RSUs is changing due to other independent applications. Hence, the demands and available resources should be predicted to perform optimal placement.

1.3 Motivations and Challenges

As proved in [14], the placement problem is NP-hard because of the multiple contradicting objectives. In our previous work [14] and [15], we proposed a solution to the problem using a Memetic Algorithm (MA). Despite the heuristic solution's ability to produce feasible decisions, it still suffers from two main limitations:

1) The chances for heuristic algorithms to diverge increase as the input to the problem increases and, therefore, the number of possible solutions increases [14].
2) The heuristic-based solution does not provide an intelligent model capable of considering the change in demands for proactive decisions.

Alternatively, a breakthrough in Reinforcement Learning (RL) has been witnessed after introducing the Deep Reinforcement Learning (DRL) algorithms. The Deep Q-Network (DQN) algorithm is an efficient DRL solution that achieves high degree non-linear function approximation. Subsequently, DRL is becoming the core solution for several resource management and networking-related problems, compelling intelligent decisions which usually require human intervention. Thus, there is a potential in using DQN in the context of fogs selection, service placement, and demand analysis. However, using DRL as a solution still suffers from two main limitations, which affect applications that are time-sensitive and cannot tolerate faults. These two limitations are:

1) The RL exploration technique causes a high volume of errors at the first stages of learning. This behavior is not tolerable in our service placement problem because it directly affects the QoS experienced by the users.
2) The long learning time for DRL agents makes it inadequate for dynamically changing environments where an agent has to adapt to changes swiftly.

1.4 Proposed Approach and Contributions

To address the aforementioned problems related to heuristics and the use of DRL for placement, we present in this paper a new scheme utilizing the on-demand fog formation technologies. We present an end-to-end architecture relying on the cloud to perform offline learning. This is achieved by incorporating an Intelligent Fog Service Scheduler (IFSS) and an Intelligent Fog Service Placement (IFSP) on the cloud. IFSS is built using an R-Learning algorithm proposed in [16], which decides on the right time and location where an environment change happens. IFSS is then responsible for triggering IFSP, formulated using DQN, to build the intelligent service placement by receiving data from the cloud server logs. This process is called Bootstrapping. A mature or tuned IFSP model is then pushed to the target fog cluster. Our IFSP then executes online updates on the orchestrator to keep our agent up-to-date with the incoming demand of each service. Benefiting from the integrated load-balancing feature in the Kubernetes orchestration tool, the clusters can scale computing resources automatically when the volume of requests increases and more computation is required.

Building our IFSP requires defining the Markov Decision Process (MDP) components, which are the states, actions, and cost function. Given the change in the services to place, the available resources, the demand for services, and the need for handling large inputs, we present in this work a scalable MDP design for our problem. This design allows the agent to make proactive placement decisions, match the demand of users, and meet the fog environment's requirements. Using our MDP design, we exploit the use of DQN to build our IFSP agent. Using IFSP, we are able to demonstrate the high utility of the decision made, the power of improving intelligent decisions by adapting to unexpected changes in the environment, and the superiority over the state-of-the-art heuristic solutions. To the best of our knowledge, our work is the first to solve the service placement problem in the fog computing context using DRL.

The contributions of this work are as follows:

- We propose a complete end-to-end architecture benefiting from the on-demand formation approach and introducing the IFSS scheduler and IFSP bootstrapper. This solution avoids the exploration errors and long learning time required by DRL algorithms.
- We present a novel MDP design to build the IFSP agent that respects the change in demand and available resources and takes optimal placement decisions based on the fog environment requirements.
- We exploit the use of DQN to build the IFSP agent. Experiments conducted using real-life datasets demonstrate the ability of IFSP to make efficient and proactive decisions, adapt to changing demand and cost parameters in the environment, and outperform existing heuristic solutions.

The rest of this paper is organized as follows. Section 2 presents the problem statement. In Section 3, we discuss recent literature work related to our context and the use of AI. Section 4 introduces the proposed architecture enabling the use of DRL. An MDP formulation of our problem is introduced in Section 7. Section 6 is dedicated to demonstrate the implemented DQN algorithm utilizing the formulated MDP. In Section 7, we provide a series of experiments for validating the use of our solution. We finally conclude the paper with future directions in Section 8.

2 Problem Statement

Users request a set of different services from the cloud using their smart devices. As shown in Fig. 1, these requests arrive from different locations with different volumes. Assuming that five services are initially running on the cloud $\{S1, S2, \ldots, S5\}$, each represented by a different color in the figure, the cloud can become overloaded with tons of requests. Subsequently, users start complaining because

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

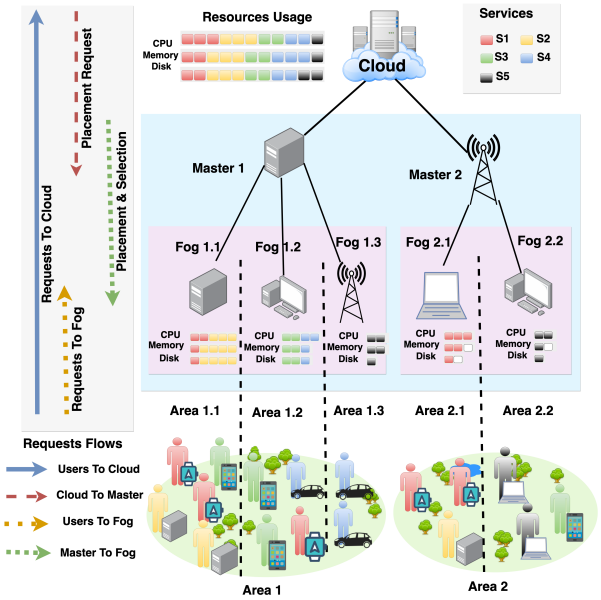


Fig. 1: An Illustration of the Service Placement Problem on Fog Clusters

of the embarrassing QoS caused by transmission delay or high load on the cloud servers. Thereafter, fogs are used to overcome these issues by running close to users with acceptable performance. As illustrated in the figure, the cloud makes scheduling decisions on a set of services that need to be placed near users. Following the approach in [16], the placement request is received by the master of the fog cluster. One of the master's tasks is to distribute these services on its fogs in a way that enhances the QoS level experienced by the users and preserves the fog requirements. For instance, these services should be as close as possible to users requesting them. Every fog has a coverage area users can reach. Therefore, fogs should serve users present within their range. In Fig. 1, Fog 1.1, serving Area 1.1, can only host $S1$ and $S2$ because of the limitation of its resources. In this subarea, a user is requesting service $S3$ (green); however, it's not hosted by Fog 1.1. Therefore, this user is still served by the cloud. Because of the fog servers resource limitation inside their Kuberenetes cluster, solving the problem of fog selection and service distribution is of immense importance. In addition, the demands of services coming from users to fogs change over time. For instance, in Area 1.2, Fog 1.2 is hosting $S3$ and $S4$. If the demands for $S1$ gets higher than $S4$, the fog has to switch these services to start serving $S1$ to achieve a better QoS. The demands are only one objective that the master has to satisfy when adjusting the placements. Other objectives have to be considered. For example, minimizing the distance between the serving fog and the user, maximizing the number of services pushed having a higher priority, and minimizing the number of hosts for lighter orchestration. These objectives are essential to consider when making selection and placement decisions.

The service placement and fog selection problem is NP-hard and requires at least a heuristic solution to solve it [14]. For instance, genetic algorithms use randomness to build populations, evaluate the solutions' fitness value, and evolve the pareto front. Thus, there is no single solution to the problem, especially when the input grows due to the hardness of reaching the optimal decision [14]. In other words, heuristic solutions are not always guaranteed to make acceptable decisions, affecting the quality of decisions made in critical situations. Furthermore, our problem requires studying users' demand of services so that services with high demand are prioritized for placement. Such a demand needs to be predicted in order for the model to react proactively by placing services before the demand occurs, and therefore eliminating the overhead of initializing fogs, migrating, and starting services. Proactive decision making is not feasible when using heuristics because of the lack of prediction model that can adapt to the stochastic change in demand for different services and from different locations.

In summary, the main limitations of using heuristics is the possibility of diverging from the optimal solution with an increase in the running time as the input size to the problem grows, and the lack of a prediction model that can study the change in the users demand for services. In order to overcome these limitations, we propose in the paper the use of DRL by modeling the selection and placement problem as MDP, providing a solution that guarantees scalable, proactive, and optimal decision making. The use of DRL is made feasible through a proposed architecture that introduced intelligent scheduling and bootstrapping to avoid (1) the long learning time, and (2) the high volume of mistakes at the exploration stage of learning.

3 Related Work

In this section, we highlight the most recent literature proposals which provide a solution to the placement problem using a heuristic. We then overview existing limitations in RL solutions for similar problems. Finally, we brief on the breakthrough of DRL in networking and resource management. Table 1 compares the features of existing solutions with IFSP.

3.1 Service Placement Using Heuristics

Existing efforts have proposed the use of heuristics as a solution for the fog's service placement problem. For instance, in our previous work [14], [15], we utilized the evolutionary MA to solve this problem in both static location and vehicular contexts. In [29], authors using mixed integer programming to solve the resource provisioning problem on MEC. Furthermore, researchers in [19] proposed to scale fog resources horizontally and vertically using reinforcement learning. In their work, service placement of scaled instances is similar to our problem, except that the authors in [19] addressed the placement for only one type of container and used linear integer programming to solve it. More generally, service placement applies in different contexts using different technologies where authors always use heuristics to reach near-optimal placement decisions. For instance, the authors in [21] use heuristics to solve Virtual Machine (VM) placement on the cloud, whereas in [23], [20], and [24], heuristic algorithms are used to place services on edge servers to support different types of users including vehicular systems.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

TABLE 1: Table of Comparison Between Latest Service Placement Solutions and IFSP

Solution	Dynamic Update (Services/cache)	Service Placement	Proactive	Bootstrapper	Scheduler	On-Demand	Scalable	Supports Mobility	Supports Multi-Services
IFSP	✓	✓	✓	✓	✓	✓	✓	-	✓
[17]	✓	✓	✓	-	-	✓	-	-	✓
[18]	✓	✓	-	-	-	-	✓	✓	✓
[14]	-	✓	-	-	-	✓	-	-	✓
[15]	-	✓	-	-	-	✓	-	✓	✓
[19], [20]	-	✓	-	-	-	-	-	-	-
[21], [22]	-	✓	-	-	-	-	-	-	✓
[23], [24]	-	✓	-	-	-	-	-	✓	✓
[25]	✓	-	-	-	-	-	-	-	-
[26]	-	✓	-	-	-	-	-	-	-
[27], [28]	✓	✓	-	-	-	-	✓	-	-

Service placement, that includes server selection, is also a well-known problem in the cloud environment. In such a context, services run inside VMs to be placed on cloud servers. The objectives for cloud-service placement problems are different than those of fog contexts, but the problem formulation and solution used can be mapped. The solution used for performing VM placement is through heuristics [22]. Thus, our solution can generalize to VM placement in the cloud environment. There are also other service management problems on the cloud layer including service composition and selection for energy minimization on datacenters. Solutions for these problems can be improved using DRL or by extending our solution to consider user requirements and energy activities within datacenters [30], [31].

3.2 Limitations of Existing MDP Designs

A common structure of papers that use reinforcement learning to solve their problem is by formulating it as Markov Decision Process (MDP). One of the main challenges for solving a problem using RL is to provide a scalable MDP design that guarantees convergence to ensure making the optimal decisions. Furthermore, the action space should be manageable by the computing machine memory when the input to the problem grows. Thus, a reduction in the size of the action space by design is highly desirable.

In [25], RL linear function approximation is used to decide on caching files on 5G base stations. The MDP design provided by the authors seems promising to be reformulated for solving our selection and placement problem. However, after following this formulation and increasing the input size, the action space can grow exponentially, which is not practical. The design proposed in [25] is adapted by other researchers performing similar tasks, such as [26]. In [26], distributed caching is applied in the wireless network. Nonetheless, their action space is a combination of sub-caching actions subject to infinite growth. Furthermore, different solutions that consider caching do not apply directly to the context of service placement where modifications of objectives, constraints, and MDP design are required. For instance, the state and action spaces must reflect the scaling operations, and the cost function should consider the cost of resources used when scaling. For simplicity, we rely on the Kubernetes master node to perform automatic scaling, which is part of its functionalities.

3.3 Deep Reinforcement Learning Advancement

After the proven success of RL in many fields, it has been adopted to solve challenging problems in the context of fog computing, such as network and resource management [32], [33]. In addition, the work in [17] proposes service migration at the edge to support cognitive computing. The MDP design suffers from scalability issues in terms of the action space, as well as a limitation of using a tabular Q-learning, making it hard to scale to large input spaces. Furthermore, the work does not study the change in available resources in order to perform proactive placement. Similarly, the proposal in [18] introduces an MDP design for container migration where the action space increases exponentially as the number of possible placements and size of the input increase. Besides, [17] and [18] do not consider the cost of training the RL agent, which is important in error sensitive environments. Researchers decide on the type of RL algorithm to use based on the dynamics of their environment and the formulated model. In the context of fog computing, authors in [27] use RL to provide a joint solution for caching, computation offloading, and radio resource allocation using the actor-critic approximation algorithm. Another work proposes a dynamic load balancing of loads on neighboring fogs using RL [34]. The agent runs on the SDN fog controller who decides to offload tasks between fogs and clouds based on the current demands/loads measured by the controller. The end goal is to minimize the execution time, where a model-free Q-learning algorithm is used.

Deep Q-Network (DQN) is a model-free off-policy RL algorithm. DQN is the first success to merge the concepts of RL and supervised Deep Learning (DL) in the video games [35], which is then proven to excel as a solution to several cloud and fog networking, caching, and resource management related problems [27], [28]. In this paper, we build our selection and placement agent, namely IFSP, using a DQN algorithm which surpasses the performance of other model-free RL algorithms for our problem. To the best of our knowledge, our work is the first to solve the fog selection and service placement problem using DRL.

4 Proposed Architecture Realizing DRL

In this section, we elaborate on a proposed architecture tailored to enable the use of DRL to solve our service placement and fog selection problem. This architecture ensures a complete end-to-end intelligent and automated solution for service placement and fog selection to improve QoS. A presentation of the architecture and components interactions is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

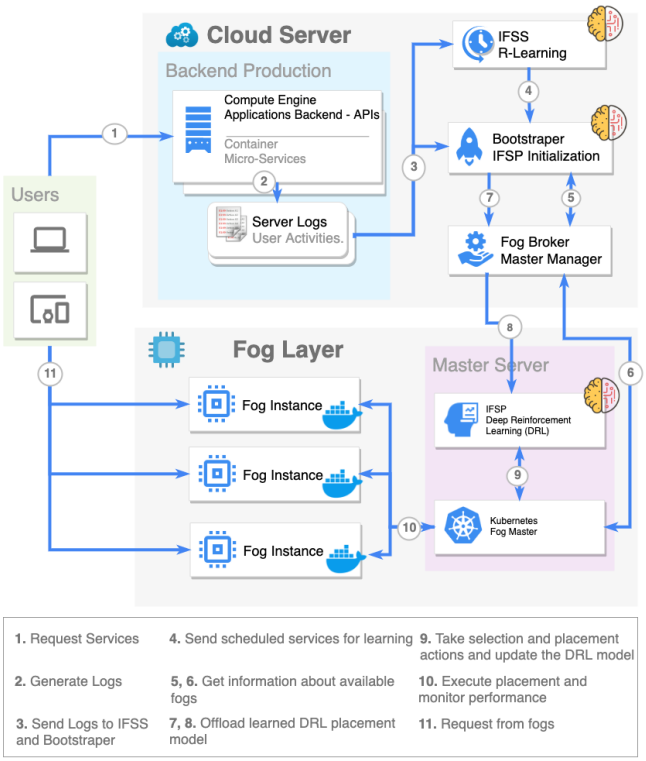


Fig. 2: Proposed Architecture Realizing DRL

depicted in Fig. 2. The two core layers of our architecture are the cloud and fog layers.

In this architecture, users can be any IoT devices. As shown in Fig. 2, users start by requesting services from the cloud. Applications on the cloud are hosted inside computing engines that are dedicated to hosting the back-end logic of the applications. Every application has a logging logic that keeps track of connected users, the source IP, and the requested APIs or services. These logs keep on updating, which means that new demands will always be taken into consideration while our models are learning. Our proposed architecture will then fully rely on these logs as a source of data to learn and make decisions. These data are then utilized by IFSS (Intelligent Fog Service Scheduler) and Bootsrapper, which are two intelligent components running two different reinforcement learning algorithms.

The IFSS agent runs an R-Learning algorithm that uses an MDP formulation for deciding about the best time and place for pushing services to fogs. IFSS was modeled and implemented in our previous work [16]. IFSS learns from the server logs the different demands of users divided by location and based on the time of the day. The decision of the IFSS scheduler is proactive, which means a decision for placing a service is taken before the actual demands occur. The bootstrapper then receives the scheduling decision by the IFSS agent, which contains the list of locations, each having a list of services to place for the given time. The Bootstrapper on the cloud runs a DRL algorithm that takes as input a state built using the services to be placed, the fogs available in the target location, and the changing demands of users over time retrieved from the server logs. The MDP for the agent is discussed in Section 4. In contrast, the DRL algorithm is presented in Section 7. The primary purpose of this model is to perform offline learning on all the demands captured in the log files using the DRL algorithm. This model is then considered as a bootstrapping for the IFSP (Intelligent Fog Service Placement) running on the master, which performs online learning. IFSP will receive a mature model and avoid the long learning time and errors the model yields at the first stages of learning.

The Fog Broker is responsible for managing all the communication between the cloud and the set of master nodes available anywhere. The Fog Broker is the gateway of the cloud to the Internet. It is horizontally scalable, keeps track of all available master nodes, and ensures reliable connections. The broker also reaches periodically to all the master nodes of the different clusters about their available fog nodes, the resources capacity, and the geographical locations of each. The Bootstrapper then utilizes this information as a requirement for the model to start offline learning. Once the Bootstrapper finishes modeling the environment and builds the DRL agent by achieving convergence, it forwards this model using the Fog Broker to the master node.

The fog layer consists of the fog clusters, each having a master node responsible for orchestrating fogs and managing the running services on each one. Each master node contains an IFSP and the Kubernetes required model for creating and maintaining a fog cluster which relies on the containerization technology. The master expects the broker's requests, which contain the new IFSP model to be adapted in its cluster. The received model is mature, and the master can rely on to make selection and placement decisions. Because user demands are stochastic, the master will also run an online update for each decision made. Using the Kubernetes model, the master has knowledge of the demands for services on each fog and how it changes over time. This information is fed for the IFSP for the online updates. Kubernetes is also used to take the actions generated by decisions from the IFSP including placing and updating containers on available fogs. Fog nodes will be able to run services that are of best use for users and therefore improving the QoS of the applications. Thus, users will migrate their requests from the cloud to available fogs.

5 IFSP Modeling

Following our proposed model, IFSP can be used in fog clusters with confidence that the deployed model online will act maturely and keep on improving itself as new demands are measured. In this section, we present a novel MDP model that is capable of taking the selection and placement decisions proactively, takes into account the changing demands per service, considers four contradicting objectives, and is scalable, which means it can model hundreds of fogs and containers as input.

5.1 Background

MDP is a mathematical framework for modeling sequential decision making in stochastic environments. An MDP is characterized by the following tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{C}, \gamma)$. $\mathcal{S}$ is the set of states that the agent can be at, where $\mathcal{S} = \{s_1, s_2, \dots\}$. $\mathcal{A} = \{a_1, a_2, \dots\}$ is the action space or the set of possible actions that can be taken by the agent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

at each step. $\mathcal{P}r$ is the probability transition matrix or the probability distribution over the successor states s' after taking an action in $\mathcal{A}$. In case $\mathcal{P}r$ is known, the agent then has a model of the environment. In most cases, $\mathcal{P}r$ is not given and the use of model-free RL is essential to obtain a model of the environment. $\mathcal{C}$ is the cost function designed to measure how well the agent is doing after taking an action from $\mathcal{A}$ at a state s and moving to a state s'. Finally, γ is a decimal value from $[0, 1]$. The value of γ is usually close to one. γ is used to help the model converge by discounting over the rewards of the next states. In other words, γ tells how much the agent cares about the reward of the future states. In order to model our problem as MDP, we need to define $\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{C}$, and γ.

5.2 States and Actions Modeling

The decision taken by the IFSP agent takes place at different time moments t, where $t = 1, 2, \ldots$. Let $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$ be the list of m fogs, where each fog has a list of available CPU, memory, disk, and geographical distance from the user. A fog F_i in the cluster can be represented as a vector $F_i = [F_{i_{cpu}}, F_{i_{mem}}, F_{i_{disk}}, F_{i_d}]$, where F_{i_d} is the mean of distances between the location of each user in a target area and the fog location. In other words, F_{i_d} measures the proximity of F_i to requesting users in the area. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be the set of n containers to place. Each service P_j has resources requirements and a priority value indicating that this service should be prioritized for placement to maintain a certain level of QoS. A service P_j has the following requirements for deployment $P_j = [P_{j_{cpu}}, P_{j_{mem}}, P_{j_{disk}}, P_{j_k}]$, where P_{j_k} is the priority of service P_j having a value of either zero for low priority or one for high priority.

Let $q(t)$ be the vector of normalized number of requests for every service in $\mathcal{P}$. An element in $q(t)$ for service P_j is denoted by $[q(t)]_{P_j}$ and is calculated as follows:

$$[q(t)]_{P_j} = \frac{\text{Number of requests for service } P_j \text{ at time } t}{\text{Total number of requests for all services in } \mathcal{P} \text{ at } t} \quad (1)$$

The placement decision is taken sequentially for each container per state at a time. The combined placement decision denoted by $k(t)$ at t is a binary matrix of size $m \times n$, where $k(t)_{ij} = 1$ means that P_j is placed on F_i, and 0 otherwise. Furthermore, a counter u is used to indicate the current container P_u the agent is taking the decision for, such that $u \in \{1, \ldots, n\}$. After making n decisions, the counter is reset to one. Henceforth, the state of our model is:

$$s(t, u) = [q(t), k(t), u] \quad (2)$$

Selecting an action from the set of possible actions $\mathcal{A}$ in our MDP allows the agent to take a placement decision for the current container P_u. The possibilities in $\mathcal{A}$ are (1) selecting a fog from $\mathcal{F}$; (2) selecting a container from $\mathcal{P}$; or (3) doing nothing. In case a fog is selected, the action performed is to place P_u on this fog. On the other hand, in case a container is selected, this container is removed from it's current running fog and replaced by P_u. Mathematically, $\mathcal{A} = \{0, f_1, f_2, \ldots, f_m, p_1, p_2, \ldots, p_n\}$, where zero means that container P_u is not assigned to any fog, f_i means the fog F_i is selected for placement, and p_j means an already placed service P_j is unplaced from its fog and replaced by the service P_u. We also denote by $a(t)$ a typical element of $\mathcal{A}$ at t. It is important to emphasize that the list of services for replacement in the action space are the ones that were placed in previous time-steps. This list of services can be extracted from $k(t)$. Therefore, there are some infeasible actions which are discarded from the actions list based on the current state. The main motivation behind considering one container to place at a time is to make the MDP design capable of handling large inputs, in contrast to [25] where the action space can grow exponentially. Moreover, placing one service at a time guarantees more availability as the other services will keep running.

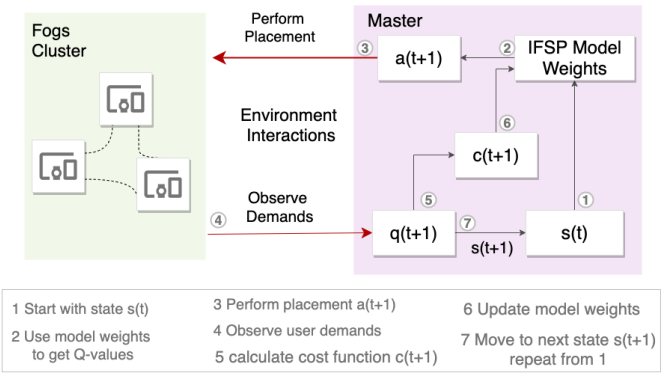


Fig. 3: The Evolution of the Main Quantities Used for Cost Calculation over Time

5.3 States Transition and Model Dynamics

Using Fig. 3, we elaborate in this section on the evolution of the key quantities used to evaluate the cost function and build the next states for the agent. In our formulation, every episode is divided into time-steps where the agent takes the placement decision for a single container. Consequently, during every episode, the agent takes separate placement decisions for each container in $\mathcal{P}$. The action taken at state $s(t, u)$ is $a(t + 1)$, which is a preparation for the coming request at state $s(t + 1, u^+)$ where $u^+ = u + 1$ if $u \leq n - 1$ and $u^+ = 1$ if $u = n$. Thus, u is incremented by 1 and reset to 1 in case $u = n$. After taking the action $a(t+1)$, $k(t+1)$ is extracted by updating $k(t)$ following the action taken. The agent then waits to observe $q(t+1)$ to be able to calculate the cost function $\mathcal{C}$ at the next state. The process of calculating $\mathcal{C}$ is elaborated in the next subsection. It is important to mention that the time difference between the time-steps is short so that the agent will be able to update the placement for all containers and learn different patterns of demands. The state space grows to cover all possible combinations of demands, resulting in more robust placement decisions.

Knowing that the loads for users' requests on services is unpredictable because of its stochastic nature, the design of our formulation helps the agent predict the common sequence of loads by considering the load of the next episode to calculate the cost of the combined decision for the current episode. Because of the dynamic change in demands, we use a model-free approach, which does not require an explicit modeling of the environment.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

In case the agent is at state $s(t,u)$ and takes action $a(t+1)$, the model has to represent the cost incurred by every previous state action in order to find the optimal policy for the new state at the next step. The calculation of the cost incurred and the formulation of the different objectives are presented in the forthcoming sub-section.

5.4 Cost Function

For an agent at a certain state, taking an action and moving to the next state is evaluated by considering four contradicting objectives. In this section, we elaborate on a mathematical formulation for each objective. The objectives are (1) minimizing unserved requests measured using unsatisfied demands per service; (2) minimizing the number of fogs selected for placement; (3) minimizing the number of not placed services with high priority; and (4) minimizing the distance of selected fogs from requesting users. A cost in our formulation is represented as $\mathcal{C}(s(t-1,u^-),a(t)|q(t))$, where $u^- = u-1$ if $u > 1$ and $u^- = n$ if $u = 1$. In the sequel, we present the mathematical formulation of the cost function which is a superposition of four sub-costs.

Throughout the cost function formulation, we denote by $g(t)$ a 1-dimensional binary list of size n, where $g_j(t) = 1$ means that the j^{th} service is placed on a fog in $\mathcal{F}$, and 0 otherwise. In addition, we denote by $r(t)$ a 1-dimensional binary list of size m, where $r_i(t) = 1$ indicates that the fog F_i is hosting at least one container from $\mathcal{P}$. $g(t)$ and $r(t)$ are extracted from $k(t)$ for each step in an episode. For each cost related to an objective, we assign a weight $\lambda_l \in [0,1]$ to it such that $\sum_{l=1}^{4} \lambda_l = 1$. These weights are adjustable depending on the service provider preferences, given that the higher the weight, the more the objective has impact.

For calculating the cost of $s(t-1,u^-)$, the first cost c_1 considers the cost of not placing requested services at the next state. The purpose of this cost is to motivate the agent to place services that will be demanded in the next time-step. Following this approach, the agent learns the pattern of how the demand of services changes over time. This objective ensures maximizing the number of satisfied requests served by the fog cluster; hence, leading to a lower response time, higher throughput and thus better overall QoS. This cost can be mathematically expressed as follows:

$$c_1 = \lambda_1(\mathbb{1} - g(t))^\intercal q(t) \tag{3}$$

Following Equation 3, $\mathbb{1} - g(t)$ results in a binary vector with 1 indicating that the container is not placed on any fog. Therefore, Equation 3 sums the loads ($q(t)$) for all unplaced services. This cost motivates the agent to satisfy as much demands as possible in order to minimize the cost.

The second cost, c_2, ensures that services with higher priority are considered in the placement decision in order to maintain an acceptable QoS level for all high priority services. High priority services do not necessarily have high demand. c_2 is then calculated using the below equation:

$$c_2 = \lambda_2 \mathcal{P}_k^\intercal [\mathbb{1} - g(t)] \tag{4}$$

where $\mathcal{P}_k$ is the vector of priority levels (0 or 1) for all services in $\mathcal{P}$. In Equation 4, we take services that are not placed by the decision at t, and some those with high priority. The aim for the agent is to minimize the total sum resulted by c_2.

In the third cost, c_3, we aim at minimizing the distance from selected fogs for placement and users requesting services to be placed. This preserves a very important feature for the fog layer, which is bringing services as close as possible to users. Respecting this objective leads to a lower response time experienced by users, therefore a better QoS. This cost is calculated using the following formula:

$$c_3 = \lambda_3 \mathcal{F}_d^\intercal N(t) \tag{5}$$

where $N(t)$ is a vector of size m indicating the total count of containers placed at each fog in $\mathcal{F}$ and $\mathcal{F}_d$ is the vector of mean distances of each fog in $\mathcal{F}$ to the users. Thus, c_3 computes the total sum of mean distances for all fogs used times the number of containers hosted on each. The end goal is to minimize this sum to ensure that running services on selected fogs are as close as possible to users.

The objective of the fourth cost, c_4, is to minimize the number of fogs used for a placement. This helps minimize the cluster complexity and the load on the orchestrator, which is responsible of managing all services running and the health of every fog. This objective leads to faster learning of changing fog resources and optimal placement, therefore improving the QoS experienced by the users. This cost is calculated as follows:

$$c_4 = \lambda_4 r(t)^\intercal \mathbb{1} \tag{6}$$

In Equation 6, we sum the number of fogs that are used for placement, by summing the 1's in $r(t)$.

Finally, the agent is allowed to make placement decisions that are not feasible, however, it's prompted to learn from it's mistakes through a punishment technique. For instance, an action is deemed infeasible if the agent overloads a fog by utilizing more than its available capacity. Thus, we calculate the CPU punishment score for the agent using the below equation:

$$p\_score_{cpu} = \sum_{i=1}^{m} \max(\sum_{j=1}^{n} (P_{j_{cpu}} k(t)_{ij}) - F_{i_{cpu}}, 0) \tag{7}$$

In Equation 7, $\sum_{j=1}^{n} P_{j_{cpu}} k(t)_{ij}$ is equal to the total CPU required by the containers and asked to be hosted on F_i. Our p_score_{cpu} calculates the excess of CPU utilization on F_i to be added to the total cost. Similar equations apply for calculating p_score_{mem} and p_score_{disk}, which are the punishment scores for memory and disk excess, by simply replacing the index cpu by mem and $disk$ respectively. Therefore, the punishment score is the sum of the three scores following Equation 8.

$$p\_score = p\_score_{cpu} + p\_score_{mem} + p\_score_{disk} \tag{8}$$

Following the calculation of the four sub-costs and the punishment score, our cost function for evaluating the agent action is expressed as follows:

$$\mathcal{C}(s(t-1,u^-),a(t)|q(t)) = \sum_{l=1}^{4} c_l + p\_score \tag{9}$$

This cost is a combination of different measures that are mainly used to evaluate the QoS level of the user in the fog

environment. In our case, minimizing the function $\mathcal{C}$ implies optimizing the QoS. Therefore, we use the cost function as a measurement of the QoS level experienced by the users in our experiments.

6 IFSP Using Deep Reinforcement Learning

The IFSP agent interacts with the environment for evaluating the placement action taken for each container. The agent executes actions for every state encountered and builds a strategy that adapts to the stochastic changing demands of users requesting services. The end goal of the agent is to learn the transition probability distribution from a state to all next states and find the optimal policy π^*, which takes as input a state and outputs the action that minimizes the future cost. In other words, π^* is a strategy or a set of actions the agent takes to minimize the cost. The future costs are discounted by γ, which controls the effect of future actions on past and current states and helps achieve the agent's mathematical convergence. By letting $\mathbb{C}(s(t,u),\pi)$ be the cost implied by choosing policy π from t that indicates the following actions $a(t^{'})$, such that $t \leq t^{'} \leq \mathcal{T}$ where $\mathcal{T}$ is the final time-step of the episodes, the future discounted cost is represented as:

$$\mathbb{C}(s(t,u),\pi) = \sum_{t^{'}=t}^{\mathcal{T}} \gamma^{t^{'}-t} \mathcal{C}(s(t^{'},u^{'}),a(t^{'})|q(t^{'})) \tag{10}$$

where $u^{'}$ is updated to u^{+} at each $t^{'}$. We denote by $Q^*(s,a)$ the optimal action value function which minimizes the average expected cost for any selected strategy. It can be expressed as:

$$Q^*(s,a) = \min_{\pi} \mathbb{E}[\mathbb{C}(s(t-1,u^{-}),\pi)|s(t-1,u^{-}) = s, a(t) = a, \pi] \tag{11}$$

The optimal Q-function selects the action of the next state that minimizes the action value function following the below equation:

$$Q^*(s,a) = \mathbb{E}_{s^{'}\sim\mathcal{E}}[\mathcal{C} + \gamma \min_{a^{'}} Q(s^{'},a^{'})] \tag{12}$$

where $\mathcal{C}$ is the immediate cost from Equation 9 and $\mathcal{E}$ is the state at $\mathcal{T}$. The basic form of RL is to find the optimal action value function using iterative updates following the Bellman equation. This update can be expressed as:

$$Q(s,a) := Q(s,a) + \alpha[\mathcal{C} + \gamma \min_{a^{'}} Q(s^{'},a^{'})] \tag{13}$$

where α is the learning rate. In Equation 13, the update of the Q-function happens following the Q-learning algorithm [36]. All Q-values are stored in a table structure containing the list of states and actions. An exploration-exploitation trade-off aids the agent into interacting with the environment by covering the maximum number of possibilities, observing the cost signal, and updating the Q-values using Equation 13.

However, the use of tabular RL is not practical in our problem, where we have a large state space. The state-space can grow with an increase in the number of containers and hosts to place. Thus, handling the whole table in memory, trying to cover all possible actions for every state, and updating the Q-values for all of them is computationally very expensive. Such an implementation is time consuming and makes any tabular RL agent diverge [37]. As a solution, learning the optimal Q-values can be retrieved from some adjustable weights denoted as θ. These weights get updated using gradient descent to update the weights downwards towards the direction of the gradient for minimizing the error of the calculated Q-values for every iteration. The common form of approximation is the linear function approximation which generalizes the environment through its weight, where the Q-function becomes close to the optimal Q^* having $Q^*(s,a) \approx Q(s,a,\theta)$.

Given the advantages of a linear approximation to overcome the tabular learning limitations, these models will not be able to generalize well when the model complexity and state spaces increase. Here comes the advantage of using non-linear approximations such as Deep Neural Network (DNN) to approximate the environment, giving the agent the power of Deep Learning (DL) to update its weights, where training can be customized [38]. The Deep Q-Network (DQN) algorithm has the advantage of merging the concepts of RL and DL [39]. Henceforth and after experimenting with the different linear approximation approaches for building our IFSP agent, including Temporal Difference TD(0) and TD(λ) [40], DQN outperforms the other approximation methods. Algorithm 1 provides a pseudo-code of our IFSP learning algorithm, which benefits from the advancement achieved in DQN.

DQN benefits from the DL power in the supervised learning paradigm of machine learning. This is made possible by introducing a replay buffer that performs mini-batch sampling and stores the weights in a target network. In the sequel, we go over our implementation of the DQN algorithm for building the IFSP agent.

As illustrated in Algorithm 1, we start by creating a multi-layer perceptron for the source model used for calculating the state action-value function Q using its weights θ. The input to the model is a transition sample, and the output is a single neuron with linear activation. A target multi-layer perceptron is created, which is a copy of the source model. We denote by θ^{-} the weights of the target model, which are a copy of θ in the initialization phase (line 1). We then initialize a replay buffer D of size $G = 1000$ which stores the transition containing the current state, the action taken, the cost retrieved, and the next state observed (line 2).

The learning starts by initializing a random state $s(t)$ at the beginning of every episode (line 5). X episodes are played for learning. X varies depending on the input size for the test case. Each episode is bounded by $\mathcal{T}$ learning steps. Every step starts by deciding on the action taken for the current state. We implement this decision by following the ϵ-greedy policy, which is essential for achieving a trade-off between exploration and exploitation. In ϵ-greedy, we set ϵ to be a variable that decays over time. For instance, $\epsilon = \frac{B1}{B2+Numberofiteration}$ decreases as the number of iteration increases, where $B1$ and $B2$ are two constants such that $B1 < B2$. We then generate a random value of w between zero and one. If $1-\epsilon > w$, we select an action randomly from the action space (lines 8-9). This is known as an exploration iteration for the agent. Otherwise, the action having the maximum Q-value in the source model is selected (lines 10-

Algorithm 1: IFSP Algorithm Using DQN

1 Build a Multi-Layer Perceptron as source model to calculate Q and randomly initialize its weights θ;
2 Build a target model for Q with weights θ^- which are a copy of θ;
3 Initialize replay buffer D to capacity G;
4 **while** *episode* X **do**
5 Initialize a random state $s(t,u)$;
6 Reset t;
7 **while** $t < \mathcal{T}$ **do**
  /* following ϵ-greedy policy */
8 **if** *Random Selection* **then**
9 select $a(t+1)$ randomly from feasible actions;
10 **else**
11 $a(t+1) = max_a Q(s(t,u), a, \theta)$;
12 **end**
13 Update $k(t+1)$, observe $q(t+1)$;
14 Calculate $\mathcal{C}(s(t,u), a(t+1)|q(t+1))$ using Equation 9;
15 Update u to u^+;
16 Build $s(t+1, u^+)$;
17 Store $[s(t,u), a(t+1), \mathcal{C}(s(t,u), a(t+1)|q(t+1)), s(t+1,u^+)]$ in D;
18 Select random mini-batch transition $(s_i, a_i, \mathcal{C}_i, s_{i+1}))$ of size Y from D;
19 **for** *j in length(mini-batch)* **do**
20 $y_i = \mathcal{C}_i + \gamma \min_{a'} Q(s_{i+1}, a', \theta^-)$;
21 **end**
22 Update θ using gradient descent towards minimizing the loss: $(y_i - Q(s_i, a_i, \theta))^2$ for every transition;
23 **if** *length(D)* $> G$ **then**
24 Pop out the oldest element in D;
25 **end**
26 Every Z steps, copy θ into θ^-;
27 Update the current state to $s(t+1, u^+)$;
28 Increment t;
29 **end**
30 Increment Episode;
31 **end**

11). This is known as the exploitation iteration.

After taking the action, the agent observes the service demands after the service placement is updated. This then allows the agent to calculate the cost $\mathcal{C}(s(t,u), a(t+1)|q(t+1))$ using Equation 9. After forming the next state $s(t+1, u^+)$, a transition is stored in the replay buffer (lines 13-17). Because updating the model online as data come causes instability, data are stored in the replay buffer. Samples from these data, of size $Y = 50$, are extracted randomly and uniformly to form the mini-batch dataset for the model to train and break the problem of correlation between sequences of actions (line 18). As mentioned previously, the source weights are stored in the target model. This is vital to improve the source model learning stability. The source model adjusts θ of Q-function by using the predicted Q-values of the target model as labels (lines 19-21). This, in turn, builds a supervised learning context with a fixed dataset and labels on which to train. In our implementation of our IFSP-based DQN, loss functions are inferred and calculated for every iteration using the mean squared error loss (line 22). This loss is back-propagated to the neurons using the gradient-descent towards minimizing the loss to get a better estimate of Q (line 22). To preserve the RL concept for allowing the model to keep on improving the Q-function as new data come, the replay buffer D keeps on updating slowly by removing the oldest transitions at every iteration when the buffer is full (lines 23-25). On the other hand, the weights for the target model θ^- keeps on updating after $Z = 500$ (line 26).

7 Experimental Study

In this section, we experiment with our proposed IFSP solution based on DQN by studying the following:

- The convergence behavior of IFSP for small, medium, and large scale clusters, where the objective is to minimize our cost function.
- The ability of the IFSP agent to adapt to changes in the environment, including unexpected changes in the users' demand patterns for requested services, and for unexpected changes in the cost function parameters.
- The ability of the IFSP bootstrapping technique on the cloud to avoid the high rate of exploration errors, make the learning faster, and scale for large inputs.
- The ability of IFSP to outperform existing heuristic-based approaches in (1) quality of the decision and (2) execution time to take the decision.

Our data utilized throughout the experiments are extracted from the Google Cluster Trace 2011-2 dataset (GCT) [41] and Nasa Server Logs (NSL) [42]. GCT provides real-life deployment scenarios of services on available servers. Thus, it provides a set of hosts with available resources and a set of services having resources requirements. In order to measure changing demands of requested services in real scenarios, we can utilize any logs present on any server, which point to the source IP of the user, the service requested, and the timestamp. These fields are enough for our IFSP agent to conduct the bootstrapping on the cloud and update itself when running on the orchestrator. In NSL, we considered source IP having the same subnet mask as a single geographical entity requesting services. Requesting a specific endpoint on the NASA web server is considered as calling a single service. Thus, for each service extracted from GCT, we assign a list of changing demands aggregated during a specified period of one hour.

Our simulation was performed on a Core i7-8700 (12 CPUs), 32GB RAM, and a Graphic card for GPU computation of type NVIDIA Quadro P620. We implemented Algorithm 1 using Python and the Tensorflow library. Using Tensorflow, we built the source and target networks. Our networks have four layers of neurons with 32, 16, 8, and 1 neurons respectively. The activation function used on each layer is ReLU, except for the output layer, where linear activation is used to predict the Q-Value. The input of the source and target neural networks is a combination of the state and action taken at that state. The first layer has an input size of $n + (m \times n) + 2$, where n is the number

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

$\mathcal{F}$	$\mathcal{F}_{cpu}$	$\mathcal{F}_{mem}$	$\mathcal{F}_{disk}$	$\mathcal{F}_{d}$
F_1	0.5	0.24	0.4	500
F_2	0.25	0.4	0.4	50
F_3	1.0	1.0	1.0	20

TABLE 2: Fogs Configurations for Scenario 1 (s1)

$\mathcal{P}$	$\mathcal{P}_{cpu}$	$\mathcal{P}_{mem}$	$\mathcal{P}_{disk}$	$\mathcal{P}_{k}$
P_1	0.12	0.2	0.2	0
P_2	0.24	0.23	0.1	1
P_3	0.18	0.2	0.32	0
P_4	0.25	0.42	0.2	1
P_5	0.31	0.15	0.24	1
P_6	0.375	0.175	0.08	0

TABLE 3: Containers Configurations for Scenario 1 (s1)

of services, and m is the number of hosts as specified in Section . We also use the RMS optimizer with a learning rate of 0.001. Furthermore, we compare our approach with two heuristic-based solutions and a DRL service placement solution. These solutions are implemented from scratch based on the objectives specified by each paper. In some of our experiments, we rely on evaluating the cost function which is directly related to the QoS experienced by the users. Furthermore, for each experiment, we run a different number of iterations depending on the objectives. For instance, we run 10^3 iterations to study the convergence and stability of IFSP. Meanwhile, 30 iterations are enough to illustrate the bootstrapper advantage.

7.1 IFSP Convergence and Scalability

In order to study the performance of IFSP following our MDP design, we simulate two different sizes of clusters with a different number of fogs and services having varying demands. Similar sizes are expected to be used in real-life settings. In Scenario 1 (s1), we use 3 fogs and 6 containers which are shown in Tables I and II and extracted from the GCT dataset. The purpose of Tables I and II is to show a snapshot of the data we have. For Scenario 2 (s2), we also use the GCT dataset to simulate a larger cluster composed of 15 fogs and 40 containers for validating the scalability of our MDP design. For both scenarios, demands are assigned to each service randomly from the NSL dataset. To compare our solution with the optimal decision, we utilize a greedy search for small inputs and pass it the demands after their occurrence. This greedy search generates all possible solutions for a given input, and yields the best placement for it, considering the same weights and cost function.

We define the weights for the current simulation in both scenarios as $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 0.25$. Fig. 4 illustrates the convergence results towards minimizing the cost function while the number of iterations increases. The experiment was executed for 10^3 iterations in order to illustrate the convergence and stability of IFSP. The results shown in this figure are an average of 50 iterations for every observation. We can observe in (s1) that IFSP is capable of converging to make optimal decisions by approaching the optimal line. The optimal line is extracted using the

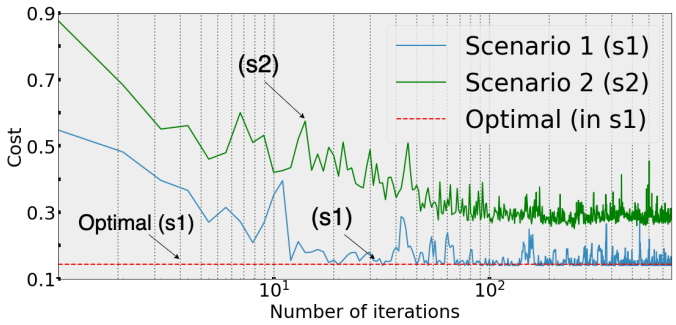


Fig. 4: Convergence Performance of IFSP for Small and Large Clusters

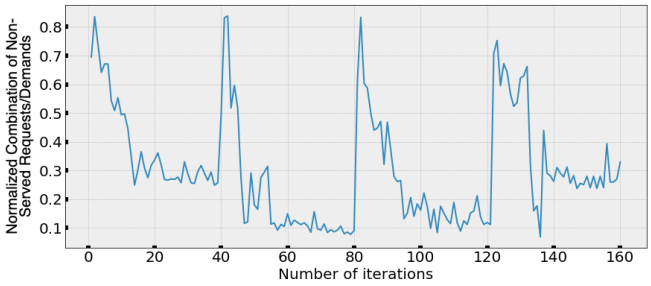


Fig. 5: IFSP Performance Evaluation while Changing Demands

greedy search method applied at each iteration for every encountered load. In (s2), the the IFSP agent also converges, validating its ability to handle large inputs. Due to the large input and high paste of changing demands, using greedy search to obtain the optimal decision is impossible. Therefore, we elaborate later in this section on the optimality of the decisions taken by comparing with a heuristic approach. We can also observe from the results that (s1) achieves faster convergence than (s2) due to the larger input experienced by the agent in (s2). Furthermore, it is important to mention that because all objective costs are given equal weights, the agent is expected to take longer because of a more complicated policy required to learn. In case one or two objectives are given a weight of zero, the convergence will be faster due to less complex policies to learn, which we demonstrate in the next subsection.

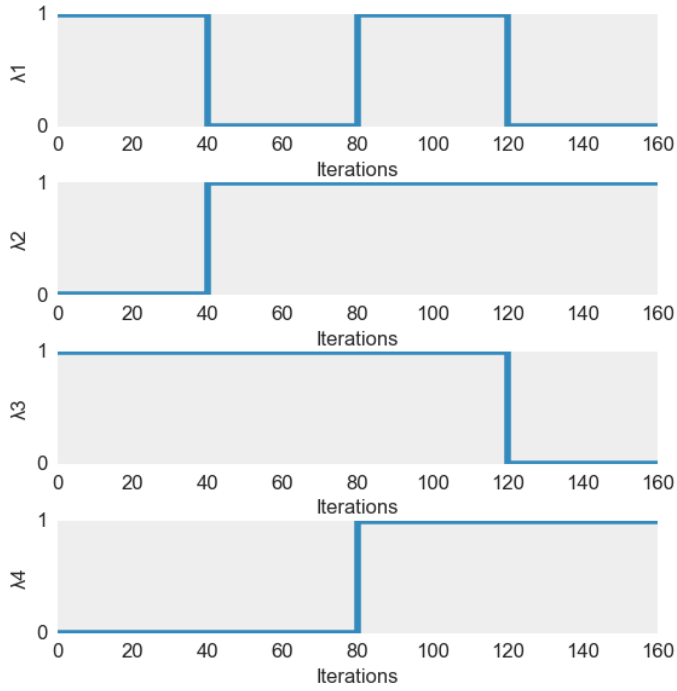
7.2 IFSP Adapting to Environment Changes

In order to elaborate further on the ability of IFSP to adapt to changing demands, we simulate Scenario 3 (s3) containing 8 fogs and 20 containers. We aim to study the combined and normalized demands of users that are not met after doing the IFSP placement for all services. In (s3), we aim to change the pattern of demands four times intentionally for each container to simulate how IFSP reacts to such a situation. The change in demands is provoked every 40 iterations, leading to 160 iterations during the evaluation. The results are shown in Fig. 5. As shown in the figure, IFSP is able to always converge into learning a pattern for the new incoming demands. We can see four jumps in the numerical cost for the four changes done to the demands because of the new states encountered. We are also able to notice through this simulation that the convergence incurred after

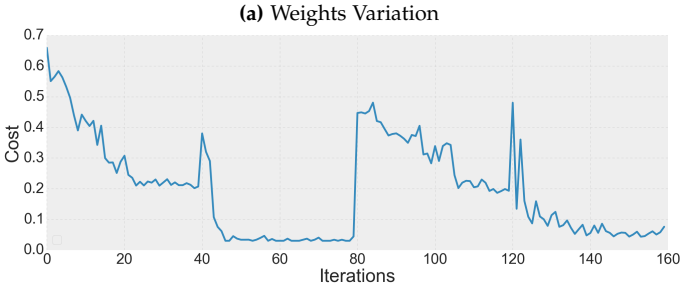
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

the second change in demands is slightly faster than the starting stages of learning. This reduction in convergence speed is due to the tuned model that we have from the first cycle of demands. As a conclusion from (s3), whenever the pattern of demands for a service changes over time, IFSP adapts to this change by learning new patterns and meeting new demands when possible. In some cases, the change in demands makes placing services impractical, and therefore IFSP refrains from placing it. This is illustrated in the first and fourth cycles of learning where the error is larger compared to the second and third cycles.
The non-served requests studied in this experiment are served by the cloud. A decrease in this amount implies that more requests are served by the fog, therefore the user experiences a lower response time and a better QoS. Therefore, we show through this experiment that IFSP is able to improve the QoS experienced by the users even-though new patterns of demand are encountered.

As mentioned in our cost function formulation, four weights are assigned to each objective cost, which can be adjusted based on the service provider's assessment of the environment's needs. Such changes in the cost function introduce changes to the cost calculated by the agent. In order to verify the robustness of the agent in such a situation, we simulate Scenario 4 (s4), which confirms the intelligence of IFSP for adapting to the changes in the cost function. In (s4), we copy the cluster configuration of (s3) and update the weights after hitting a predefined number of iterations by the agent. The strategy for updating the weights is illustrated in Fig. 6a. In this figure, a signal equal to one means that the weight is in use for the current cycle. If more than one weight is used, the total weight is divided equally among them. In Fig. 6b, we show the performance of IFSP while considering four consecutive changes in weights. Every 100 iterations in (s4) are averaged to obtain the results shown in the figure. The IFSP performance is measured using the normalized cost function. The cost function starts converging at the beginning until a change to the weights occurs. A change in the weights causes a peak in the cost, as shown in the results. After every change in weights, IFSP is able to converge again to optimal solutions.



(a) Weights Variation



(b) Model Convergence

Fig. 6: IFSP Performance Evaluation while Changing Weights

7.3 Comparison with DRL and Heuristic Approaches

Following the large input in (s2), we study in this section the performance of our IFSP agent compared to existing heuristic-based solutions [14], [43]. Besides, we compare with a DRL solution for service placement at the edge [18] and show the importance of using the IFSP bootstrapper. The large input caused scalability issues by overloading the memory when generating the possible actions in [18], which is not the case when using IFSP. We also present the limitations of the existing heuristic solutions in terms of execution time when the input to the problem grows, thus requiring more iterations to try finding the near optimal solution. This in turn results in increasing the execution time to make the placement decision, henceforth delaying the update of services or ignoring them completely as demands can change more often. We are also able to highlight the importance of proactive placement, which is not possible when using a heuristic solution. In this context, we build Scenario 5 (s5), which is a copy of (s2) input. However, in (s5), the IFSP model has passed the bootstrapping on the cloud discussed in Section 4, hence the model is a continuation of the learning that happened in (s2). Knowing that heuristic approaches rely on randomness to generate solutions, the range of possibilities is considerable when the input size is large, making it hard for the algorithm to hit the optimal solution. For instance, in (s5), and following our MDP design, the agent has 825 possibilities of placement to be taken for each observed demand, which is an acceptable number for such a large input. If we consider the service placement formulation in [14] or in [43] for cloud/fog placement, the number of possibilities is the different binary combinations of a matrix of size 15×40. The implemented heuristic solution embeds a local search to minimize the chances of halting in local optimal, and speeds reaching a feasible solution. The same cost function formulated in Equation 9 is implemented for fitness evaluation in MA. We used 200 generations and 100 individuals per generation that evolves to find a feasible solution. More details about the implemented MA can be found in [14].

After completing the bootstrapping phase in (s5), we utilize the ready model to make decisions, benefiting from

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2021.3075988, IEEE Transactions on Services Computing

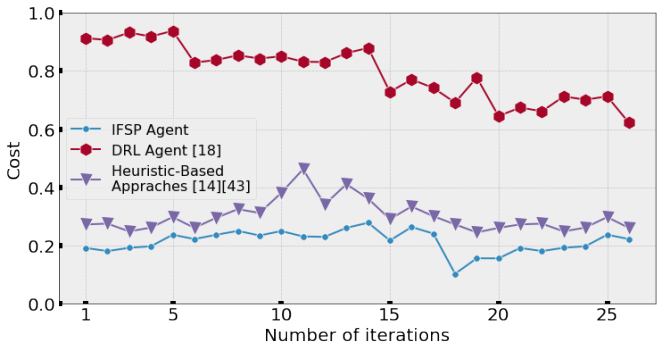


Fig. 7: Our IFSP Agent Performance v.s. Heuristic-Based Approaches In Scenario 5 (s5)

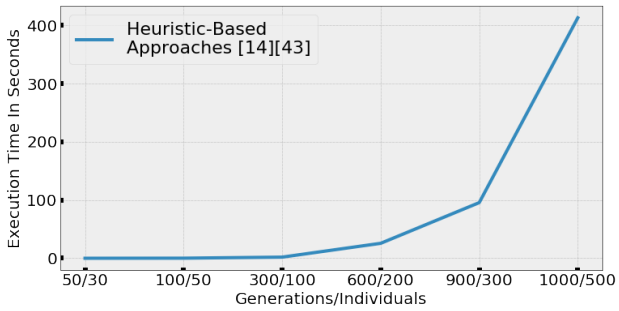
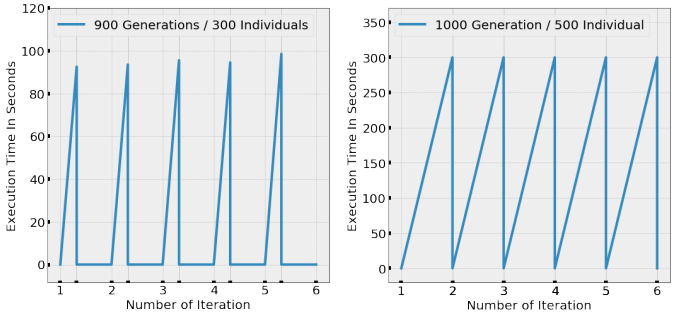


Fig. 8: Execution Time While Changing The Number of Generations and Individuals In Scenario 5 (s5)



(a) Execution Time With 900 Generations and 300 Individuals

(b) Execution Time With 1000 Generations and 500 Individuals

Fig. 9: The execution time of heuristic-based approaches [14], [43] using generations/individuals of 900/300 v.s. 1000/500 and changing the demands every 5 minutes for each iteration in scenario (s5)

the mature model. On the other hand, we run the DRL agent of [18] without bootstrapping, causing the agent to start learning from scratch on the current environment. Afterward, a snapshot of the decisions made by IFSP and the DRL agent of [18] is taken. The lists of fogs and services are passed to the heuristic solution to make the placement decisions. In the existing heuristic solutions [14] [43], the change in demands for services is not studied. In order to measure the heuristic decision's performance, we pass the input to MA 10 times and record the average cost of the placement decisions. A snapshot of the decisions made by IFSP compared to heuristic-based approaches following (s5) is illustrated in Fig. 7. The evaluation is performed for 30 iterations, which are enough to show the importance of using a bootstrapper compared to a traditional DRL solution. Besides, each environment state is passed to heuristic for evaluation, which is time-consuming. This also explains the choice of 30 iterations for evaluation.

Following the performance of IFSP compared to DRL and heuristic-based decisions, we can observe that the normalized costs produced by IFSP are always less than or equal to those produced by both solutions. The IFSP and heuristic-based results are not only for the snapshot of the sample, but also always valid after IFSP converges. From Fig. 7, we can observe the large difference in the cost results for all decisions made. In the case of DRL [18], the agent starts at the beginning by exploring the environment and taking random actions, which causes a high cost (i.e. low QoS) at the first stages of learning compared. In contrast, the IFSP solution has a pretrained model using the bootstrapper. This comparison highlights the importance of using a bootstrapper to overcome existing DRL limitations. Due to the limited capabilities of the heuristic solutions, proactive placement of services is not possible. Because we are executing the heuristic algorithm periodically, the placement of the current timestamp is outdated because it does not meet the actual demand. Moreover, due to the large input size, the execution time of the heuristic solution increases exponentially as illustrated in the next experiments. In this context, initializing the environment and migrating the containers consume more time, rendering the heuristic algorithms infeasible in time-sensitive applications. Noting that the results of Fig. 7 do not consider the time to setup the environment based on the new placement decision. On the other hand, IFSP is capable of taking decisions on the fly with a negligible processing time.

In heuristic-based algorithms, the processing time increases due to the increase in the input size. Every Pareto set, or list of best solutions is generated every time in heuristic by looping for specific number of generations and manipulating a set of individuals. The higher the numbers of generations and individuals are, the more likely the heuristic algorithm will generate better solutions. In (s5), because the input is large, we varied the numbers of generations/individuals and studied the run time for making the placement decision. The results of this experiment are revealed in Fig. 8. The execution time of the heuristic solution increases exponentially as the numbers of Generation/Individual increase. Because our IFSP solution is based on DRL, the execution time to get the placement decision is negligible because the agent takes a forward pass on the deep network for each action to select the best.

In order to elaborate further on the execution time drawback of using the heursitic solutions to solve our problem, we consider applying the heuristics to our placement environment by feeding it the changing demands in (s5). The time between one iteration and another when the demands change is 5 minutes. The purpose of the experiment is to show the time needed for heuristic-based solutions to update services in the environment. We also varied the numbers of generations/individuals for each trial. The results are shown in Fig. 9a for 900/300 and in Fig. 9b for 1000/500.

As shown in Fig. 9a, the heuristic solution takes around

1.5 minutes to generate every solution. This is shown in every peak in execution time at the beginning of the iteration. These peaks are the time taken to update services, whereas our IFSP agent updates the services proactively before demands occur. Increasing the number of generations/individuals to 1000/500, we observe in Fig. 9b that the execution time of the heuristic solution is not terminating at every iteration (never updating the services), because the time it takes to produce the solution is more than 5 minutes (the duration of observing new demands).

Therefore, heuristic-based solutions are not suitable for time-sensitive placement problems and can be replaced by our solution. IFSP is capable of producing more efficient results in minimal execution time. We also benefit from the IFSP Bootsrapper running on the cloud to prepare the orchestrator. Furthermore, because IFSP is (1) scalable, (2) capable of adapting to changes in the environment, and (3) making proactive decisions before demands occur, it can completely replace the state-of-the-art heuristic solutions.

8 Conclusion and Future Work

Fog and service placement is a challenging problem in demands-driven context entailing the need for effective decisions while adequately adapting to environmental changes. The use of heuristic solutions to perform the placement is not feasible due to the changing demands and the possibility of heuristics to diverge from optimal solutions. Empowered by the breakthroughs in the AI field, we exploit in the paper the use of DRL as an intelligent solution for fog selection and container placement. Despite the errors made by the agent at the exploration stage and the long time required to learn, we are able to build an IFSP agent based on DQN capable of making efficient decisions in no time. This is possible by incorporating an intelligent IFSS scheduler and a bootstrapper for preparing the IFSP model before being used. We then formulated an MDP design used for developing the IFSP agent based on the DQN algorithm. Our MDP formulation allows the agent to take proactive decisions, to study the change in user demands, and to consider fulfilling multiple objectives for serving the fog computing context. Through experimental studies, we used real-life datasets and demonstrated our IFSP agent's ability to generate efficient solutions for small and large cluster sizes. We were also able to validate the ability of IFSP to adapt to changes in the environment, including demand changes and preferences adjustments for calculating the cost function. In addition to these advancements, we were able to exhibit the power of our intelligent solution for generating better solutions compared to the state of the art heuristic solutions in large scale clusters.

As future work, we are working on advancing the scalability of the bootstrapping running on the cloud. Because the cloud has to orchestrate a large number of on-demand fog clusters, efficient federated learning is promising for achieving faster learning and responsiveness to changes in the fog clusters. Furthermore, caching the bootstrapped models is useful for efficient management of computing resources on the cloud. Thus, the development of an intelligent caching mechanism based on studying the frequency of demanding some IFSP model is another research direction.

References

[1] M. R. Palattella, M. Dohler, A. Grieco, G. Rizzo, J. Torsner, T. Engel, and L. Ladid, "Internet of things in the 5g era: Enablers, architecture, and business models," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 510–527, 2016.

[2] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in *Internet of everything*. Springer, 2018, pp. 103–130.

[3] A. V. Dastjerdi and R. Buyya, "Fog computing: Helping the internet of things realize its potential," *Computer*, vol. 49, no. 8, pp. 112–116, 2016.

[4] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency iot services in multi-access edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 668–682, 2019.

[5] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE access*, vol. 6, pp. 47 980–48 009, 2018.

[6] H. Sami and A. Mourad, "Towards dynamic on-demand fog computing formation based on containerization technology," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2018, pp. 960–965.

[7] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[8] D. Vohra, *Kubernetes microservices with Docker*. Apress, 2016.

[9] N. Moati, H. Otrok, A. Mourad, and J.-M. Robert, "Reputation-based cooperative detection model of selfish nodes in cluster-based qos-olsr protocol," *Wireless personal communications*, vol. 75, no. 3, pp. 1747–1768, 2014.

[10] S. A. Rahman, A. Mourad, M. El Barachi, and W. Al Orabi, "A novel on-demand vehicular sensing framework for traffic condition monitoring," *Vehicular Communications*, vol. 12, pp. 165–178, 2018.

[11] A. A. Abdallah, S. S. Saab, and Z. M. Kassas, "A machine learning approach for localization in cellular environments," in *2018 IEEE/ION Position, Location and Navigation Symposium (PLANS)*, 2018, pp. 1223–1227.

[12] W. Fawaz, R. Atallah, C. Assi, and M. Khabbaz, "Unmanned aerial vehicles as store-carry-forward nodes for vehicular networks," *IEEE Access*, vol. 5, pp. 23 710–23 718, 2017.

[13] W. Fawaz, "Effect of non-cooperative vehicles on path connectivity in vehicular networks: A theoretical analysis and uav-based remedy," *Vehicular Communications*, vol. 11, pp. 12–19, 2018.

[14] H. Sami and A. Mourad, "Dynamic on-demand fog formation offering on-the-fly iot service deployment," *IEEE Transactions on Network and Service Management*, 2020.

[15] H. Sami, A. Mourad, and W. El-Hajj, "Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 778–790, 2020.

[16] P. Farhat, H. Sami, and A. Mourad, "Reinforcement r-learning model for time scheduling of on-demand fog placement," *The Journal of Supercomputing*, vol. 76, no. 1, pp. 388–410, 2020.

[17] M. Chen, W. Li, G. Fortino, Y. Hao, L. Hu, and I. Humar, "A dynamic service migration mechanism in edge cognitive computing," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, pp. 1–15, 2019.

[18] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.

[19] F. Rossi, V. Cardellini, and F. L. Presti, "Elastic deployment of software containers in geo-distributed computing environments," in *Proc. of IEEE ISCC'19*, 2019.

[20] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards qos-aware fog service placement," in *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017, pp. 89–96.

[21] H. Goudarzi and M. Pedram, "Energy-efficient virtual machine replication and placement in a cloud computing system," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 750–757.

[22] N. M. Calcavecchia, O. Biran, E. Hadad, and Y. Moatti, "Vm placement strategies for cloud scenarios," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 852–859.

[23] L. Zhao and J. Liu, "Optimal placement of virtual machines for supporting multiple applications in mobile edge networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 7, pp. 6533–6545, 2018.

[24] A. Ceselli, M. Premoli, and S. Secci, "Mobile edge cloud network design optimization," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1818–1831, 2017.

[25] A. Sadeghi, F. Sheikholeslami, and G. B. Giannakis, "Optimal and scalable caching for 5g using reinforcement learning of space-time popularities," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 180–190, 2017.

[26] X. Lin, Y. Tang, X. Lei, J. Xia, Q. Zhou, H. Wu, and L. Fan, "Marl-based distributed cache placement for wireless networks," *IEEE Access*, vol. 7, pp. 62 606–62 615, 2019.

[27] Y. Wei, F. R. Yu, M. Song, and Z. Han, "Joint optimization of caching, computing, and radio resources for fog-enabled iot using natural actor–critic deep reinforcement learning," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2061–2073, 2018.

[28] Y. Sun, M. Peng, and S. Mao, "Deep reinforcement learning-based mode selection and resource management for green fog radio access networks," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1960–1971, 2018.

[29] N. Kherraf, H. A. Alameddine, S. Sharafeddine, C. M. Assi, and A. Ghrayeb, "Optimized provisioning of edge computing resources with heterogeneous workload in iot networks," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 459–474, 2019.

[30] T. Baker, B. Aldawsari, M. Asim, H. Tawfik, Z. Maamar, and R. Buyya, "Cloud-senergy: A bin-packing based multi-cloud service broker for energy efficient composition and execution of data-intensive applications," *Sustainable Computing: informatics and systems*, vol. 19, pp. 242–252, 2018.

[31] P. Kendrick, T. Baker, Z. Maamar, A. Hussain, R. Buyya, and D. Al-Jumeily, "An efficient multi-cloud service composition using a distributed multiagent-based, memory-driven approach," *IEEE Transactions on Sustainable Computing*, 2018.

[32] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[33] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Fscaler: Automatic resource scaling of containers in fog clusters using reinforcement learning," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2020, pp. 1824–1829.

[34] J.-y. Baek, G. Kaddoum, S. Garg, K. Kaur, and V. Gravel, "Managing fog networks using reinforcement learning based load balancing algorithm," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2019, pp. 1–7.

[35] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[36] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[37] X. Xu, L. Zuo, and Z. Huang, "Reinforcement learning algorithms with function approximation: Recent advances and applications," *Information Sciences*, vol. 261, pp. 1–31, 2014.

[38] S. S. Saab and D. Shen, "Multidimensional gains for stochastic approximation," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 5, pp. 1602–1615, 2019.

[39] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[40] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[41] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[42] Nasa dataset - two months of http logs from a busy www server. [Online]. Available: https://ita.ee.lbl.gov/html/contrib/NASA-HTTP

[43] F. López-Pires and B. Barán, "Many-objective virtual machine placement," *Journal of Grid Computing*, vol. 15, no. 2, pp. 161–176, 2017.

Hani Sami is currently pursuing his Ph.D. at Concordia University, Institute for information Systems Engineering (CIISE). He received his M.Sc. degree in Computer Science from the American University of Beirut and completed his B.S. and worked as Research Assistant at the Lebanese American University. The topics of his research are Fog Computing, Vehicular Fog Computing, and Reinforcement Learning. He is a reviewer of several prestigious conferences and journals.

Azzam Mourad received his M.Sc. in CS from Laval University, Canada (2003) and Ph.D. in ECE from Concordia University, Canada (2008). He is currently an associate professor of computer science with the Lebanese American University and an affiliate associate professor with the Software Engineering and IT Department, Ecole de Technologie Superieure (ETS), Montreal, Canada. He published more than 100 papers in international journal and conferences on Security, Network and Service Optimization and Management targeting IoT, Cloud/Fog/Edge Computing, Vehicular and Mobile Networks, and Federated Learning. He has served/serves as an associate editor for IEEE Transaction on Network and Service Management, IEEE Network, IEEE Open Journal of the Communications Society, IET Quantum Communication, and IEEE Communications Letters, the General Chair of IWCMC2020, the General Co-Chair of WiMob2016, and the Track Chair, a TPC member, and a reviewer for several prestigious journals and conferences. He is an IEEE senior member.

Hadi Otrok holds an associate professor position in the department of ECE at Khalifa University of Science and Technology, an affiliate associate professor in the Concordia Institute for Information Systems Engineering at Concordia University, Montreal, Canada, and an affiliate associate professor in the electrical department at Ecole de Technologie Superieure (ETS), Montreal, Canada. He received his Ph.D. in ECE from Concordia University. He is a senior member at IEEE, and associate editor at: Ad-hoc networks (Elsevier) and IEEE Networks. He served in the editorial board of IEEE communication letters. He co-chaired several committees at various IEEE conferences. His research interests include the domain of computer and network security, crowd sensing and sourcing, ad hoc networks, Reinforcement Learning, and Blockchain.

Jamal Bentahar received the Ph.D. degree in computer science and software engineering from Laval University, Canada, in 2005. He is a Professor with Concordia Institute for Information Systems Engineering, Concordia University, Canada. From 2005 to 2006, he was a Postdoctoral Fellow with Laval University, and then NSERC Postdoctoral Fellow at Simon Fraser University, Canada. He is an NSERC Co-Chair for Discovery Grant for Computer Science (2016-2018). His research interests include the areas of computational logics, model checking, multi-agent systems, service computing, game theory, and software engineering.