Using Deep Reinforcement Learning for Application Relocation in Multi-access Edge Computing

Fabrizio De Vita, Giovanni Nardini, Antonio Virdis, Dario Bruneo, Antonio Puliafito, Giovanni Stea

Abstract

Multi-access Edge Computing (MEC) brings data and computational resources near mobile users, with the ultimate goal of reducing latency, improving resource utilization and leveraging context- and radio-awareness. Relocation policies for applications in the MEC environment are necessary to guarantee its effectiveness and performance, and can use a multitude of different data (user position and direction, availability of MEC services and computation resources, etc.). In this paper, we advocate using deep reinforcement learning to relocate applications in MEC scenarios, by having MEC learn during the evolution of the sytem. We show the feasibility of this approach and highlight its benefits via simulation, also presenting an environment which can foster future research on this topic.

Index Terms

Multi-access Edge Computing, 4G/5G, LTE, Deep Reinforcement Learning, Data Migration, SimuLTE

I. Introduction

SMART services have become more and more demanding from a performance standpoint, requiring tight latencies that cloud-based communications are not able to meet, especially when cloud and user are far apart. In such a scenario, Multi-access Edge Computing (MEC), being standardized within the European Telecommunications Standards Institute (ETSI), is emerging as a viable alternative [1]. MEC allows cloud-computing capabilities to be placed at the edge of a mobile network, close to mobile users. This not only abates communication latency, but also allows smart services to leverage context information (e.g., user position, real-time awareness of the state of the radio access network, etc.), via secure and controlled interfaces, thus greatly increasing the effectiveness of the services themselves. Albeit first designed in the context of 4G, LTE-based mobile networks, MEC is expected to play a key role in the upcoming 5G ecosystem, where it can also be implemented as part of the Network Functions Virtualisation (NFV) architecture [2]. Its use-cases range from offloading user computations, to mobile big data analytics, connected cars, indoor positioning, etc..

A major issue with MEC is application relocation: to reap the benefits of low latency and context- and radio awareness, applications must follow their mobile users, relocating from a MEC server to another [3]. While the MEC standard includes *functions* to relocate applications seamlessly to the user, suitable *policies* are required to decide which applications should be relocated, when, and where. These decisions should take into account the availability of required services at the destination server, the computational load on the latter, the difference in communication latency, and the overhead and performance cost of relocation, with the goal of optimizing the Quality of Service (QoS) not just for a single user, but on a global scale. This is a challenging problem, further exacerbated by the heterogeneity of applications, which have different performance requirements and impose different storage, computation, and communication loads.

The relocation problem has been tackled in several works, using different approaches. Authors in [4] adopt a traditional machine-learning approach to forecast user mobility and implement a proactive relocation scheme in order to minimize the downtime of the system. The approach described in [5] uses a multi-agent reinforcement learning (RL) scheme, where agents compete among themselves to establish the best offloading policy. In this paper, we argue that deep Reinforcement Learning (deep RL), rather than traditional machine-learning algorithms or formalisms like Markov Decision Processes (MDP), should be used for application relocation in MEC. In fact, when the environment has a large number of states (e.g., 10^{20} or more), traditional techniques (such as MDP, heuristic methods, etc.), tend to be ineffective. Clustering has been proposed as a means to reduce the number of states, but such a technique introduces errors that may make a policy suboptimal or altogether ineffective. Specifically, deep RL is a novel technique which has been pioneered by DeepMind [6], providing a new way to represent complex environments, where the dimension of the state space and action space is very large; in particular, the state space dimension is strictly related to the number of parameters (or *features)* that define the state itself. For this reason, when we

F. De Vita, D. Bruneo and A. Puliafito are with the University of Messina (Italy), Department of Engineering
{fdevita, dbruneo, apuliafito}@unime.it
G. Nardini, A. Virdis and G. Stea are with the University of Pisa (Italy), Dipartimento di Ingegneria dell'Informazione
g.nardini@ing.unipi.it, {antonio.virdis, giovanni.stea}@unipi.it

work with states consisting of a large number of parameters, the state-space dimension tends to explode, making it impossible to use MDP or traditional machine-learning approaches [7].

In this paper, we design a self-adaptive AI-powered algorithm, which is capable of understanding the system status and accordingly relocates applications with the goal of improving user QoS. We test the above algorithm in a simple, but comprehensive case study, related to a social-media video streaming application, where the MEC and the surrounding environment (e.g., a 4G network including all the protocol stack as well as radio resource contention and communication latency) are modeled. Our results show that the AI engine we designed was able to understand its environment from the knowledge of network and system parameters, learning a complex policy which is very difficult to define using heuristic methods.

Some applications of deep RL in networking scenarios are already present in the literature. For example, the authors of [8] propose a deep RL approach for the deployment of an optimal offloading policy between mobile devices and cellular base stations. In [9], the authors implemented a deep RL framework that learns binary offloading decisions from its experience. Even though both the aforementioned works enforce deep RL techniques, they are mainly related to general mobile systems, not specifically addressing 5G/LTE features. Moreover, they only consider channel information as an input to the algorithms, whereas our approach is more general and allows us to manage several system status information and actions, thus implementing complex policies (e.g., relocating an application to a specific location).

The contribution of this paper is twofold: *i)* we designed a deep RL algorithm that is able to relocate applications in a MEC-enhanced LTE network without needing an explicit knowledge of all the involved aspects; *ii)* we realized a deep RL environment, by integrating OMNeT++/SimuLTE and Keras, that can be used as a *gym* to test different RL approaches in LTE scenarios.

II. Multi-access Edge Computing

The core elements of the MEC architecture proposed by ETSI are Mobile Edge (ME) Hosts [10]: nodes with storage and computation capabilities placed close to radio base stations in order to provide low-latency and context-aware services to User Equipments (UEs), see Fig. 1(left). The ME Host provides an environment where ME Applications (MEApps) are run as instances of Virtual Machines (VMs) on top of a Virtualisation Infrastructure (VI) and can interact with other MEApps and services provided by the ME Platform (MEP) to accomplish their tasks, using the provided Application Programming Interface (API). The VI manages the computing and storage resources required to run software images of MEApps inside VM instances. The MEP handles the instantiation and termination of MEApps, provides services that can be exploited by MEApps, and applies traffic rules for the data plane among MEApps, services, and the underlying network. Relevant services are the Location Service (LS) and the Radio Network Information Service (RNIS). The LS provides information on the position of UEs within the network at different levels, e.g., geolocation or the indication of the radio base station serving a UE. The RNIS provides information about the radio network, such as the available bandwidth on the radio interface or UE channel conditions.

The overall view of the ME Hosts deployed in a MEC system is maintained by the ME Orchestrator (MEO). The MEO collects information about the topology of the system (i.e., location of ME Hosts and their connections), the amount of resources available on each ME Host and what services it provides. In particular, it receives requests of MEApp instantiations from the UEs and dispatches them to the ME Hosts based on the available resources and application-specific QoS constraints, e.g. a maximum latency.

A MEApp can be either dedicated to a specific UE or shared among several UEs, e.g., providing multicast/broadcast services. In the former case, the MEApp is instantiated and terminated on request of the UE. In the latter case, if the MEApp is already running on the ME Host, a UE request only subscribes that UE as a user of that MEApp. Moreover, a MEApp may or may not need to maintain a service state (or context) related to the UE(s) it is serving.

While ME Hosts form a logical topology, their physical location will determine the latency involved in host-to-user communications, and influence the network load and relocation policies. A possible deployment is to have ME Hosts co-located with radio base-stations (eNodeBs, in the LTE lexicon), as shown in Fig. 1(right). In that case, all the data-plane traffic involving MEApps only traverses the radio access network, and latencies are minimized. Maintaining this situation in mobility might require the MEC system to be able to relocate MEApps at every UE handover. Not doing it, in fact, creates additional inter-cell traffic. Moreover, problems arise when MEApps involve several users in different cells at the same time. As an alternative, ME Hosts can be deployed “close” to eNBs, so that a single ME Host will serve a geographic area covering some contiguous cells, trading some latency and added network traffic for a smaller MEApp relocation frequency.

The procedure for relocating a MEApp depends on the characteristics of the MEApp [11]. In case of a dedicated MEApp, the application instance is moved from the source ME Host to the target one. Otherwise, the UE data-plane traffic is only redirected towards the target ME Host. For stateful MEApps, also the UE context is transferred to the target ME Host. For simplicity, in the following we refer to *MEApp relocation* regardless of the MEApp type.

An example of MEApp relocation is shown in Fig. 1(right). When the UE changes its serving radio base station (1), the RNIS at the source ME Host collects this information (2). The ME Application Mobility Service (MAMS) running into the MEP of the source ME Host saves the current status of the MEApp to be relocated (for stateful MEApps only) and sends a relocation request to the MEO (3). The MEO identifies the potential target ME Host and verifies whether it has enough resources, supports the specific MEApp and can satisfy the QoS requirements. If so, the MEO triggers the instantiation of a new MEApp in the target ME Host (4) and the transfer of the UE context, if any (5). Finally, it terminates the MEApp at the source ME Host (6) and updates the traffic rules for the data plane. For shared MEApps, the target ME Host might already have a running instance of that MEApp, hence only the UE context need to be transferred (for stateful MEApps only) and the traffic rules are updated.

MEApp relocation can also occur as a result of system-wide procedures (e.g., affecting all the ME Hosts under the control of the same MEO), aiming at optimizing specific metrics, e.g., utilization of computing resources. This approach can take advantage of the complete knowledge of the system (e.g., available computation power and storage, service availability on ME Hosts, etc.). In this case, the relocations can be decided by a centralized entity, possibly the MEO itself, which also initiates the same relocation procedure shown in Fig. 1(right) from step 3.

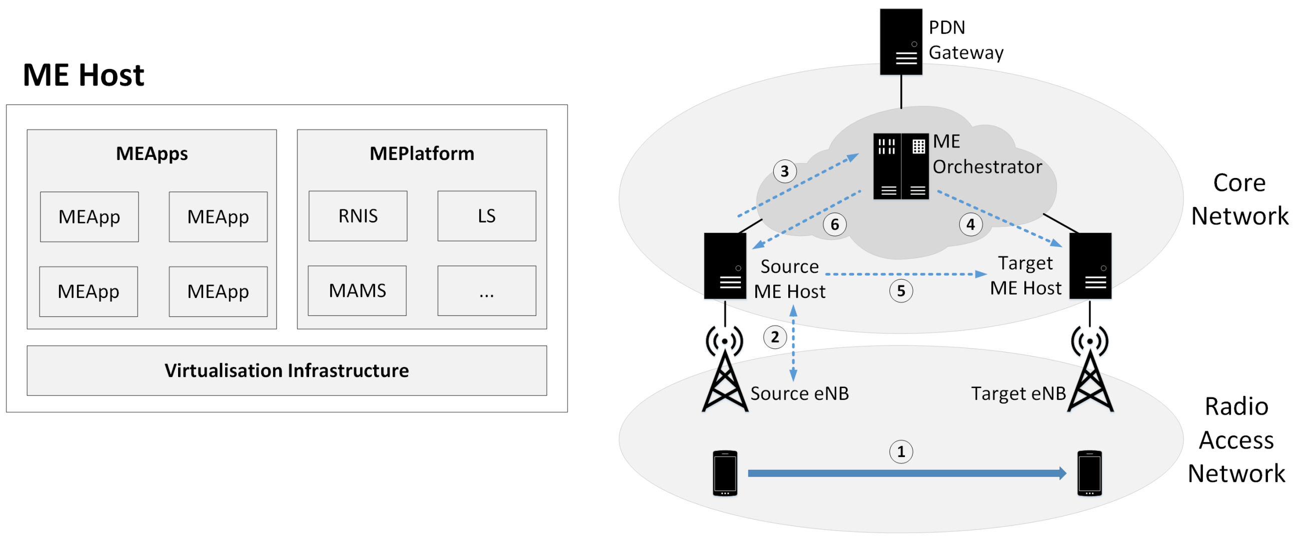


Fig. 1 – ME Host architecture (left) and relocation procedure (right).

III. Deep Reinforcement Learning for Application Relocation

In this section, we propose a framework for enabling the relocation of MEApps between ME Hosts, which leverages deep RL to optimize the system performance in terms of QoS.

A. *Reference Architecture*

The logical architecture of ML applications in 5G networks is discussed in [12]. A set of *sources* (e.g. UEs) provide relevant data to a *collector*, which feeds a *model* entity (the one running the ML algorithm). The output of the model represents the *policy* that needs to be enforced on *sink* entities, i.e. the action to be taken on the targets of the ML application. The logical architecture can be implemented according to a specific technology (e.g., MEC) by placing the above entities in the network according to the specific use case.

We consider the MEC-enhanced LTE-A network shown in Fig. 1(right), where ME Hosts are placed close to the eNBs. ME Hosts run the MEApps requested by UEs. However, the following architecture is independent of the underlying cellular technology, hence it can be implemented also in 5G networks. The core of the architecture is the deep RL engine located at the MEO (i.e., representing the model entity of the architecture in [12]). The deep RL engine periodically runs the learning algorithm to find a sub-optimal policy through a trial-and-error process. Then, given the set of MEApps whose locations need to be optimized, it applies that policy to produce the new configuration for the MEApps to be used. The decision is based on the current configuration of the MEApps and the knowledge of various information gathered by the MAMS located within the MEP of each ME Host and acting as the collector. The services of the MEP represent the sources of data for the ML application. In particular, the MAMS obtains from the RNIS the overall number of UEs attached to the eNB to which the ME Host is associated, and, for each MEApp running on the ME Host, it obtains the geolocation of the corresponding UE from the LS. Additional data from various sources can be considered, according to the requirements of the specific MEApps (e.g., end-to-end delay measurements from UEs) or the underlying network architecture (e.g., per-slice analytics from the Network Data Analytics Function - NWDAF - in 5G networks [13]). This information can be sent to the MEO using the standard API provided by the MEC framework. Once the algorithm terminates, the MEO initiates the relocation procedure described in Section II based on its results, instructing the MEP of the involved ME Hosts to take the necessary actions to move the MEApps, which are the sinks of the ML application.

B. Deep RL and Problem Formulation

RL is a machine-learning technique used to observe the dynamics of an environment, thus learning an optimal policy with respect to one or more performance indexes. In fact, its ability to learn through trial and error makes it a good choice to solve decision-making problems. RL adopts the MDP formalism, which is a framework for modeling decision making in stochastic environments. An MDP includes a set of *states* $\mathcal{S}$ that the environment can assume, a set of *actions* $\mathcal{A}$ that the agent can perform, a *reward function* $R(s) : \mathcal{S} \rightarrow \mathbb{R}$ that measures the reward associated to an action, and a *discount factor* $\gamma \in [0,1]$ stating the importance of future rewards. In such a context, the objective of the RL agent is to find an optimal policy that maximizes the reward while performing actions in the environment. The agent traverses the environment states several times and changes the system policy to favor the actions that maximize the reward. A typical RL approach is Q-learning [7], a *model-free* technique which tries to learn the relationship between actions in a given state and the associated rewards, by updating a Q-value $Q(s, a)$, which returns the utility of performing action a when the agent is in state s. However, when the number of states which describe an environment is too large, it is not possible to use traditional techniques. In such a context, deep RL is a valid solution to this problem, using Deep Neural Networks (DNNs) as a function approximator to predict the Q-values of a state. The key idea at the base of deep RL is using *two* separate DNNs: the *Main* DNN, used to predict the Q-values associated to states, and the *Target* DNN, used to generate the target Q-values needed to train the *Main* DNN. The Target DNN is necessary since the Main DNN cannot update itself effectively on its own. By using two networks instead of just one, with a different set of parameters, the result is a more stable training, which is not prone to oscillations or policy divergence.

We assume that the MEC-enabled network includes a number of eNBs. For the sake of simplicity and without any loss of generality, we assume that each eNB has its own ME Host. UEs are free to move around the environment and attach to different eNBs through seamless handover procedures. UEs run applications which communicate with the MEApp contained inside one of the ME Hosts. With respect to the actions that the agent can perform in the environment, let us define a set of actions $A =< (a_1, a_2, a_3, ..., a_{\mathcal{Z}}) >$, each representing a MEApp relocation from a ME Host towards another one. States s_j, $1 \leq j \leq \mathcal{T}$, are tuples containing all the information related to the UE positions, the apps they are running, and the ME Hosts where they run:

$$\begin{aligned} s =< (&< UE^i > \;\; i = 1, ..., M, \\ &< UE^i_k > \;\; i = 1, ..., M, \;\; k = 1, ..., N, \\ &< ME\_Host^i_k > \;\; i = 1, ..., M, \;\; k = 1, ..., N) > \end{aligned} \tag{1}$$

where UE^i represents the number of UEs connected to the *i-th* eNB, UE^i_k is the number of UEs running the *k-th* application in the *i-th* eNB, and $ME\_Host^i_k$ is a value that is 1 if the *k-th* MEApp is in the *i-th* ME Host, 0 otherwise.

Finally, we denote the reward as a number $r \in \mathbb{R}$ that can be computed as a combination of several network performance indexes.

C. Proposed Algorithm

In this paragraph, we describe the proposed deep RL approach for the MEApp relocation shown in Algorithm 1.

Line 1 sets up the replay memory E, a data structure containing the experiences accumulated by the agent over time. Initially, the weights of the two neural networks, call them $\theta, \widehat{\theta}$, are set to the same value (lines 2-3). The discount factor (line 4) has already been described. The batch size (line 5) is the amount of experience that will be fetched from the replay memory. The update period (line 6) is the number of execution steps after which the weights of the two networks are synchronized. The exploration rate (line 7) measures the propensity for a random action over the "reasoned" one provided by the DNN. The exploration rate decays over time at a rate d (line 8), to favor convergence. At each for-loop iteration, the agent observes the current state (line 10) and selects the action to perform, depending on the exploration rate (lines 11-16). Then, the agent

Algorithm 1: *Deep RL for MEApp relocation in MEC*

```
1  initialize experience replay memory $E$ to {}
2  random initialize main DNN network weights $\theta$
3  set target DNN network weights $\widehat{\theta}$ equal to $\theta$
4  set discount factor $\gamma$
5  set batch size $B$
6  set update period $U$
7  set exploration rate $\epsilon$
8  set decay rate $d$
9  for episode = 1 to end:
10   observe state $s_j$ containing current UEs positions and applications distribution over the network ME Hosts
11   $p$ = random([0,1])
12   if $\epsilon > p$:
13     action = random([1,$\mathcal{Z}$])
14   else:
15     action = argmax(Q($s_j$,$\theta$))
16   end if
17   relocate the application to the destination ME Host as indicated by action
18   observe the new state $s_{j+1}$ which contains the UEs positions and the new application distribution after the relocation
     process
19   observe the reward $r$
20   store the tuple ($s_j$,$action$,$s_{j+1}$,$r$) in $E$
21   sample a batch from $E$
22   y = Q($s_j$,$\theta$)
23   $y_{target}$ = $\widehat{Q}(s_{j+1},\widehat{\theta})$
24   $y_{action}$ = r + $\gamma \cdot max(y_{target})$
25   execute one training step on main DNN network
26   every $U$ steps set $\widehat{\theta} = \theta$
27 end for
```

observes again the state reached by the environment and the correspondent reward (lines 18-19), and stores the experience gained by the agent inside E (line 20). The core of the algorithm lies in lines 22-26. First of all, the main DNN predicts the Q-values for the given state s_j (line 22). Then, target Q-values are evaluated through the target DNN network (line 23) using the update formula (line 24); more specifically only the Q-value related to the action sampled from the batch will be updated, leaving the other values unaltered. After the update, the main DNN network is trained by executing one training step (line 25), and if U steps have elapsed, the target DNN network weights are set to the main DNN network ones (line 26).

IV. A simulation environment

The general scheme of a deep RL environment is depicted in Fig. 2. The *deep RL engine* runs Algorithm 1. The *environment*, i.e., the part that provides input to the RL engine in the form of sensed states, is modified by the engine actions, and issues the corresponding rewards, consists of a MEC-enhanced LTE-A network with several ME Hosts where mobile users move while accessing services. The environment is simulated in detail using two frameworks of the popular OMNeT++ discrete-event ecosystem, namely SimuLTE [14] and INET[1]. The latter allows one to simulate application-level communication, using all the protocol layers of the TCP/IP suite, in an LTE-based cellular network with multiple nodes and mobile users. Fig. 3 represents the high-level architecture and layering of the main communicating nodes. The UE and the eNB nodes possess an LTE Network Interface Card (NIC), which provides wireless connectivity through the radio-access network, and implements a model of the LTE protocol stack, complete with PHY, MAC, RLC and PDCP, layers. Each UE can be configured with multiple TCP- or UDP-based applications, which can communicate with the Internet or with the ME Host(s). Moreover, UEs move according to a waypoint-based mobility model, wherein UEs move linearly and at constant speed between randomly generated waypoints. The eNB is connected to the UE via the LTE NIC on one side, and to the EPC through the GTP layer on the other. ME Hosts

[1] Available at “https://inet.omnetpp.org/”, last accessed May 2019

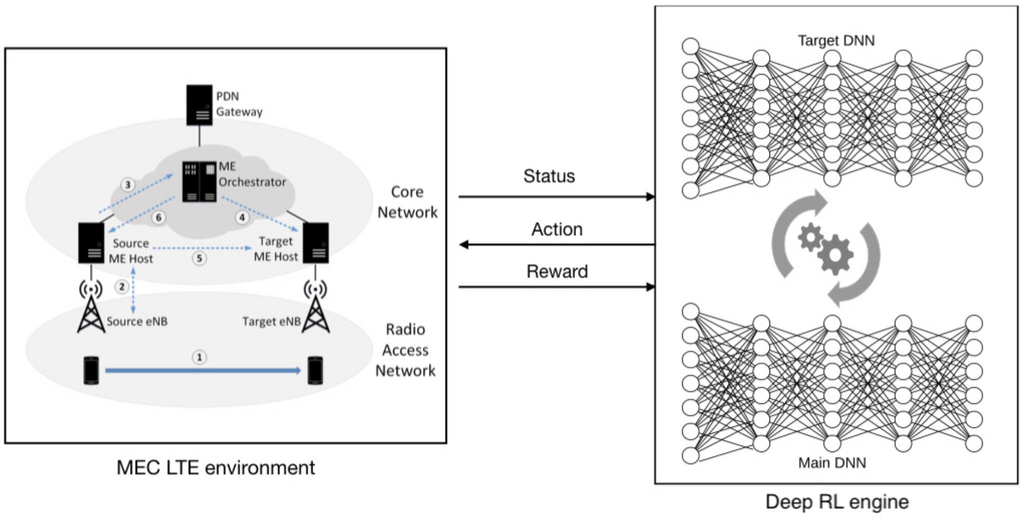


Fig. 2 – Deep RL environment.

are modeled in the latest release of SimuLTE [15]. They can be placed at arbitrary points of the EPC. In a ME Host, ME Apps can be instantiated on demand, by request of the UE, and can be associated with a resource demand (RAM, storage and CPU), so that resource constraints and admission control on a ME host can be enforced.

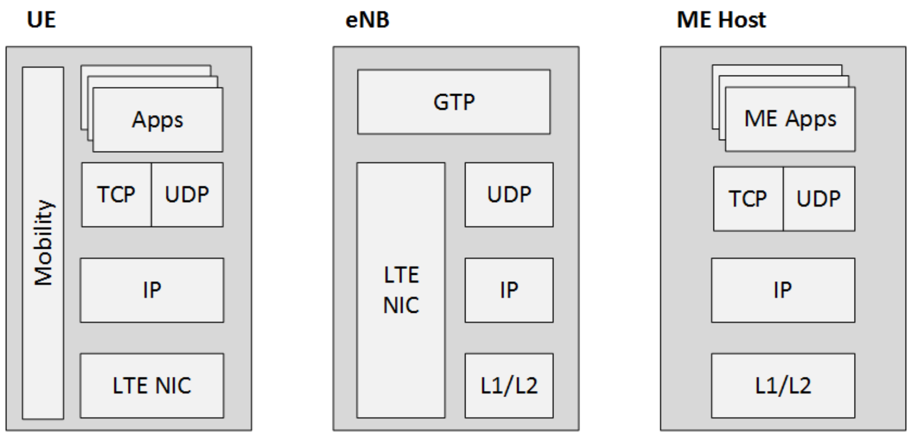


Fig. 3 – High-level view of the simulator main nodes and their layering.

For the deep RL engine, we used Keras[2], an open source library that allows one to build complex neural networks. Using Keras, we built a feed-forward fully connected DNN composed of n hidden layers in between the input layer, whose dimension is given by the cardinality $\|\mathcal{T}\|$ of the state tuple, and the output layer, whose dimension is given by the cardinality $\|\mathcal{Z}\|$ of the action set.

The deep RL engine implemented in Python is executed directly from the OMNeT++ simulator code using a syscall. When data containing the current state of the system is ready, the deep RL solver is called, and its output is a text file listing the action to be enforced in the simulator. On the OMNeT++ side, as soon as the file is available, the simulator is able to read the action code and change the ME Host destination address for the UEs that are running the application indicated in the action, thus emulating application relocation. After executing the action, the deep RL agent observes the reward, computed by combining several performance indexes provided by the OMNeT++ simulator, and checks whether or not its last action has increased it. The reward is thus used by the agent as a feedback to understand if the action executed is a valid choice for that specific system state.

[2] Available at "https://keras.io", last accessed May 2019

V. Results

In this section, we evaluate a proof-of-concept case study, where the use of deep RL can be beneficial. In an era where the use of social applications is increasing, we propose an example of urban scenario where a group of stationary users access a video streaming application and make comments in real time. Simultaneously, another group of users access a different client-server application while traversing a city highway.

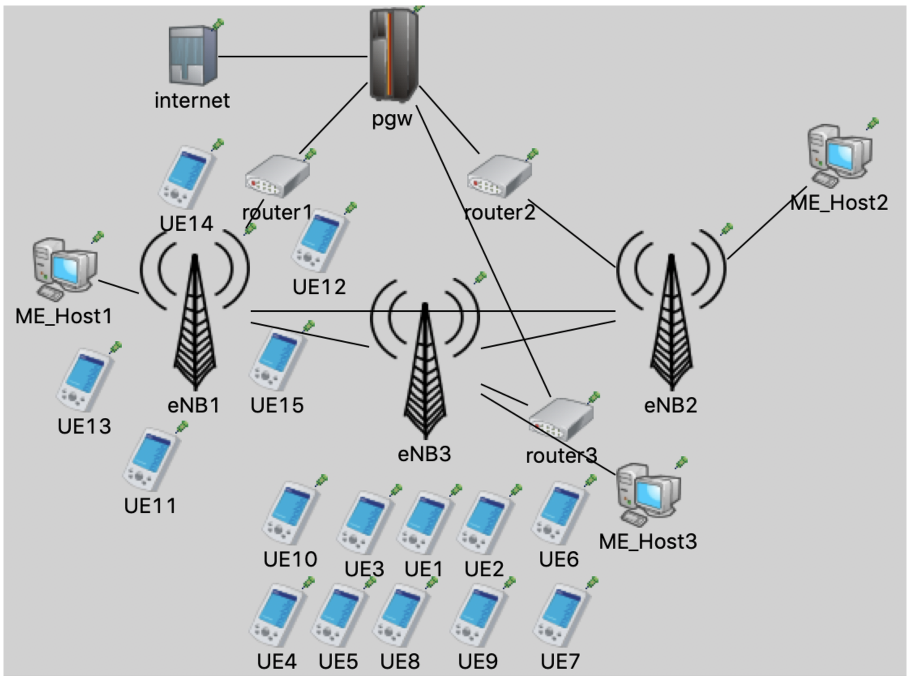


Fig. 4 – Simulated scenario on OMNeT++ environment.

Fig. 4 depicts the simulation scenario. In this case study, we deployed two different MEApps: the first one ($MEApp_A$) is always run by the ME_Host3, while the second one ($MEApp_B$) can be relocated to any ME Host in the network. Each UE sends a request every 50ms, and the response time at the ME Host is proportional to the number of UEs served by it. The scenario contains *three* eNBs in a straight line located 230m apart from each other. Moreover, we define two different groups of UEs: a first group of 10 UEs (numbered from 1 to 10) attached to eNB3, which are static and use the $MEApp_A$, and second group of 5 UEs (numbered from 11 to 15), initially attached to eNB1, that use the $MEApp_B$ while moving at a speed of about 50km/h, representative of a generic car speed, towards eNB3 and eNB2. These last UEs perform handovers based on signal strength. This allows us to compare our system with three straightforward heuristics i.e., a "no-relocation" policy, where the $MEApp_B$ always runs on the initial ME Host, a "load" policy where the $MEApp_B$ is relocated towards the least loaded ME Host, and a "majority" policy where the $MEApp_B$ is relocated towards the ME Host directly connected to the eNB that has the majority of UEs attached to it. We compare the application-level delay in the above policies, measured over 10-second intervals and resulting from the sum of two terms: a Round-Trip Time (RTT), covering the network traversal time of a packet, which is influenced (among other things) by the *relative position* of the UE and its serving ME Host, and a computation time at the ME Host, which instead depends on the load of the ME Host itself, i.e., on the number of UEs connected to it.

In order to test our system, we trained the deep RL agent and let it accumulate experience in the scenario we previously defined. Fig. 5 shows a performance comparison of the policy learned by our system and the above-mentioned three policies, together with a histogram which depicts how the deep RL agent learns as the number of training hours (i.e., the experience) increases, thus reducing the overall application-level delay. To make a fair comparison, we fixed both the starting position and the mobility pattern of the UEs for each experiment, so that each plot refers to exactly the same scenario.

Fig. 5a depicts a comparison between the policy learned by the system after $22.5h$ of training and a "no-relocation" policy, where the application always remains at the ME_Host2. In such a context, the deep RL agent has learned an effective policy, that allows the system to reduce the latency of the application for most of the simulated time. The intervals in which the two policies exhibit similar delays correspond to configurations where the "no-relocation" policy is optimal. One can observe that, at the beginning of the simulation, and between 600s and 1100s, our system drastically improves the performance of the application by reducing the delay by about 45%.

Fig. 5b compares the policy learned by our system to the "majority" policy. In this case, too, our solution outperforms the other. In fact, during intervals [100s; 200s], [500s; 600s], [1100s; 1200s], where the majority of the users are under eNB3, together with the ten stationary UEs connected to ME_Host3 running $MEApp_A$ (as shown in Fig. 4), our solution understands that it is preferable to relocate the $MEApp_B$ towards the closest ME Host *except* for the ME_Host3, which has already a load of ten users, thus reducing the application delay by about 20%.

Fig. 5c reports a comparison between the policy learned by our system and a "load" policy, which relocates the applications towards the least loaded ME Host, regardless of its distance to the UE. Due to the fact that ME_Host3 always accommodates ten stationary users, such a heuristic relocates the $MEApp_B$ between ME_Host1 and ME_Host2. Such a behaviour leads to a fluctuation of the delay, except for three cases, corresponding to the intervals mentioned above, when most of the users are attached to eNB3. In such a context, our policy outperforms the heuristic during almost the entire simulation, proving once again that the deep RL agent has understood the network dynamics and it is able to deploy a near-optimal policy.

Finally, Fig. 5d shows the mean delays, computed by averaging the delays obtained during an entire simulation, evaluated at different training phases. As expected, at the beginning of the training process the actions are still strongly affected by the random exploration, which causes the system to make mistakes, leading to a noticeable increase of the delay. After $0.5h$ of training the deep RL agent has already learned a better policy, reducing the initial delay by about 55%. Once the agent has reached $5.5h$ of training time, it has learned a near-optimal policy, further reducing the delay to few percentage points of the one obtained at the end of the training process. In general, we can see that as the training progresses, the policy learned by the agent improves by reacting to the movements of the user in order to keep the QoS as high as possible, thus reducing the delay.

The results obtained demonstrate the feasibility of our technique and encourage us to investigate new techniques in order to further improve the performance of our system.

VI. Conclusions

In this paper, we presented a deep RL approach to find an optimal relocation policy in MEC scenarios. We designed a deep RL algorithm and we compared the policy learned against three heuristics, showing that the deep AI engine outperforms the other policies. Future works will be devoted to better integrate the algorithm with the OMNeT++ environment, to compare with other solutions, to use more realistic traffic and mobility models, and to investigating new indexes with the aim to further improve the system performance.

Acknowledgment

Work partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project at the University of Pisa (Departments of Excellence).

References

[1] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Data processing in cyber-physical-social systems through edge computing," *IEEE Access*, vol. 6, pp. 29822–29835, 2018.

[2] V. Sciancalepore, F. Giust, K. Samdanis, and Z. Yousaf, "A double-tier mec-nfv architecture: Design and optimisation," in *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, Oct 2016, pp. 1–6.

[3] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1657–1681, thirdquarter 2017.

[4] P. Bellavista, A. Zanni, and M. Solimando, "A migration-enhanced edge computing support for mobile devices in hostile environments," in *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, June 2017, pp. 957–962.

[5] M. G. R. Alam, Y. K. Tun, and C. S. Hong, "Multi-agent and reinforcement learning based code offloading in mobile fog," in *2016 International Conference on Information Networking (ICOIN)*, Jan 2016, pp. 285–290.

[6] Mnih Volodymyr *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[7] Stuart J. Russell and Peter Norvig, *Artificial Intelligence - A Modern Approach (3. internat. ed.)*, Pearson Education, 2010.

[8] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Performance optimization in mobile-edge computing via deep reinforcement learning," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, Aug 2018, pp. 1–6.

[9] Liang Huang, Suzhi Bi, and Ying-Jun Angela Zhang, "Deep reinforcement learning for online offloading in wireless powered mobile-edge computing networks," 2018.

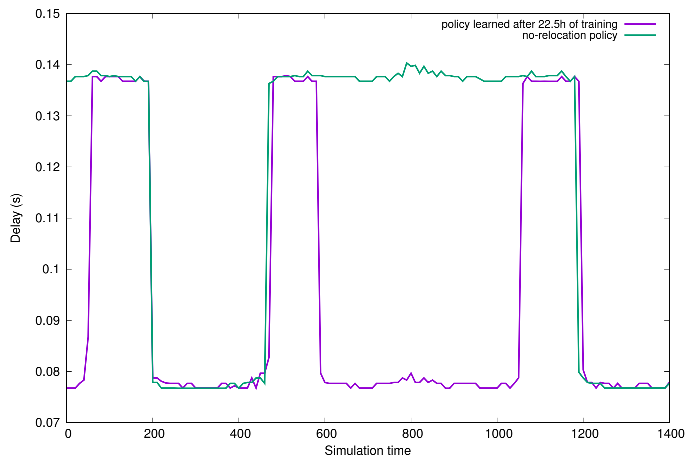
[10] ETSI GS MEC 003 V2.1.1, "Mobile edge computing (mec); framework and reference architecture," January 2019.

[11] ETSI GR MEC 018 V1.1.1, "Mobile edge computing (mec); end to end mobility aspects," October 2017.

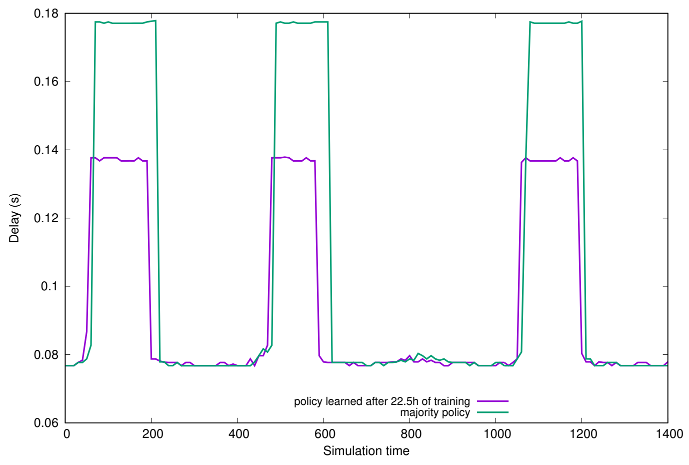
[12] ITU FG-ML5G, "Unified architecture for machine learning in 5g and future networks," January 2019.

[13] 3GPP TR 23.791 (v16.1.0), "Study of enablers for network automation for 5g (release 16)," March 2019.

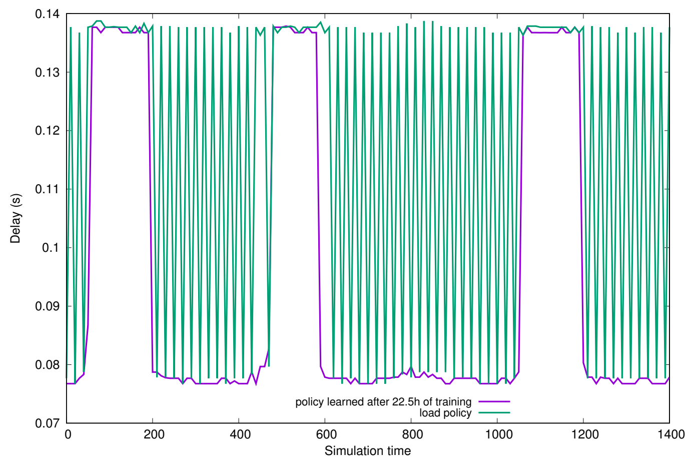
[14] A. Virdis, G. Stea, and G. Nardini, "Simulte - a modular system-level simulator for lte/lte-a networks based on omnet++," in *2014 International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, Aug. 2014, vol. 00, pp. 59–70.



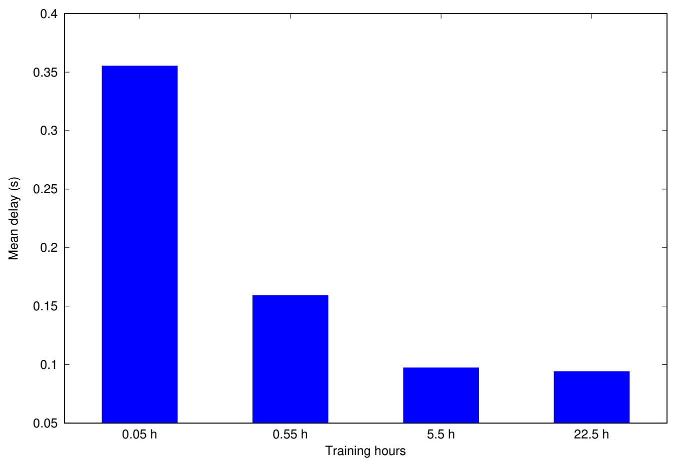
(a) Deep RL agent Vs. “no-relocation” policy



(b) Deep RL agent Vs. “majority” policy



(c) Deep RL agent Vs. “load” policy



(d) Mean delay reached by the deep RL agent at different training phases

Fig. 5 – Policy comparisons and impact of the training time.

[15] G. Stea G. Nardini, A. Virdis and A. Buono, “Simulte-mec: Extending simulte for multi-access edge computing,” in *Proceedings of 5th International OMNeT++ Community Summit*, 2018, vol. 56, pp. 35–42.

Fabrizio De Vita is a PhD student at the University of Messina, Italy. The research activity of Fabrizio De Vita is focused on the Internet of Things, embedded systems, and Machine Learning techniques for Cyber-Physical Systems with applications in Smart Environments and Smart Industry contexts.

Giovanni Nardini is a post-doc researcher at the Department of Information Engineering of the University of Pisa, where he obtained his PhD in Information Engineering in 2017 and his MSc in Computer System Engineering in 2013. His research interests are resource allocation in cellular networks, multi-access edge computing and network simulation. In these fields, he has coauthored five patents and more than 20 peer-reviewed papers. He has been involved in EU-funded and industrial research projects.

Antonio Virdis is assistant professor at the University of Pisa, where he obtained his MSc degree in Computer System Engineering in 2011, and his PhD in Information Engineering in 2015. His research interests include Quality of Service, scheduling and resource allocation in wireless networks, network simulation and performance evaluation. He has been involved in national, EU-funded and industry-funded research projects. He coauthored six patents and 30 peer-reviewed papers in the field of cellular network modeling and algorithms.

Dario Bruneo is an Associate Professor of Computer Engineering at the University of Messina, Italy. The research activity of Dario Bruneo has been focused on the study of distributed systems with particular regards to the management of advanced service provisioning, to the system modeling and performance evaluation. His current research topics include Internet of Things (and its application in Smart City scenarios), performance and reliability of complex systems, Machine Learning techniques for Cyber-Physical Systems. He has published over 100 papers in international journal and conferences and he is co-editor of the book "Quantitative Assessments of Distributed Systems - Methodologies and Techniques" - Scrivener/Wiley.

Antonio Puliafito is a full professor of computer engineering at the University of Messina, Italy. His interests include distributed systems, networking, IoT and Cloud computing. He is acting as an expert in ICT for the European Commission since 1998. Till Sept 2018, he acted as the President of the Centre on Information Technologies at University of Messina. He participated in several European projects such as Reservoir, Vision, CloudWave and Beacon. He has contributed in the development of several tools such as WebSPN, ArgoPerformance, GS3 and Stack4Things. He is member of the management board of the National Center of Informatics in Italy (CINI) and the director of the CINI Italian Lab on "Smart Cities & Communities". He is in charge of the #SmartME crowdfunding initiative, to develop a smart city infrastructure in the city of Messina. He is author and co-author of more than 400 scientific papers.

Giovanni Stea received his Ph.D. degree in 2003 from the University of Pisa, Italy, where he is currently associate professor in the Department of Information Engineering. He has coauthored more than 100 peer-reviewed papers and 16 patents. He has been involved in national and European research projects, and he has led joint research projects with industrial partners. He served as a member of the technical and/or organizing committees for several international conferences, including SIGCOMM, WoWMoM and VALUETOOLS. He also serves on the editorial boards of Springer Wireless Networks. His current research interests include Quality of Service and resource allocation in networks, performance evaluation through simulation and analytical techniques, and multi-access edge computing.