A DEEP REINFORCEMENT LEARNING APPROACH FOR DATA MIGRATION IN MULTI-ACCESS EDGE COMPUTING

Fabrizio De Vita[1]*; Dario Bruneo*[1]*; Antonio Puliafito*[1]*; Giovanni Nardini*[2]*; Antonio Virdis*[2]*; Giovanni Stea*[2]

[1]University of Messina, Department of Engineering
{fdevita, dbruneo, apuliafito}@unime.it
[2]University of Pisa, Department of Information Engineering
{g.nardini}@ing.unipi.it, {antonio.virdis, giovanni.stea}@unipi.it

ABSTRACT

5G technology promises to improve the network performance by allowing users to seamlessly access distributed services in a powerful way. In this perspective, Multi-access Edge Computing (MEC) is a relevant paradigm that push data and computational resources nearby users with the final goal to reduce latencies and improve resource utilization. Such a scenario requires strong policies in order to react to the dynamics of the environment also taking into account multiple parameter settings. In this paper, we propose a deep reinforcement learning approach that is able to manage data migration in MEC scenarios by learning during the system evolution. We set up a simulation environment based on the OMNeT++/SimuLTE simulator integrated with the Keras machine learning framework. Preliminary results showing the feasibility of the proposed approach are discussed.

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

1. INTRODUCTION

Smart services represent the core element in a smart city environment; thanks to IoT diffusion together with the advancement of technology in terms of computation, nowadays users can access a large number of applications that usually communicate with the Cloud which provides support to them [1]. However, applications are becoming more and more resource demanding and they have reached a level where the cloud paradigm can no more guarantee low latencies especially when the distance between it and the user device is very large. In such a context, Multi-access Edge Computing (MEC) can address this problems by moving data and computational resources nearby the user [2]. 5G technology has the ambition to realize a framework where different technologies can cooperate to improve the overall performance, also leveraging context-related information and real-time awareness of state of the local network (e.g., congestion, types of services enabled). MEC is regarded as a key enabler for the 5G key performance indicators, such as low latency and bandwidth efficiency. In a 5G system, MEC is expected to interact with the rest of the network to improve traffic routing and policy control. The adoption of the MEC paradigm in the new 5G-enabled systems is a hot research topic with different solutions already presented in the literature. In this paper, we will focus on the data migration problem taking as reference a MEC system composed of a LTE network with some MEC servers attached to the eNodeb (eNB) base stations. Our intent, is to build a self-adaptive AI-powered algorithm capable to understand the system status and accordingly migrate users applications data with the final goal to improve the user Quality of Service (QoS). Thanks to machine learning capability to build complex mathematical models which allow a system to learn intricate relationships among a large number of parameters, we believe that such a technique could be an enabling technology for the future 5G systems where context aware network infrastructures must be able to make decisions in an autonomous way in order to improve their performance. In the literature, we found some papers that face this problem with different approaches. The authors in [3] adopts a traditional machine learning approach using LibSVM toolkit to have a forecast on users mobility and implement a proactive migration mechanism in order to minimize the downtime of the system. However, as the author remark, this kind of technique is more resource demanding if compared with discrete models like Markov Decision Processes (MDP). The approach described in [4], uses a multi-agent reinforcement learning scheme where agents compete among themselves in order to establish the best offload policy. In this paper, we present a deep reinforcement learning approach to deploy an optimal migration policy in order to improve user QoS. The main difference between our work and the others consist in the use of a deep learning technique instead of traditional machine learning algorithms or formalisms like MDPs. We believe that the use of deep learning can be a valid solution in order to build a system which is capable to adapt to changes by understanding the network state and performing actions autonomously. The paper contribution is twofold: we designed a deep RL algorithm that can be used as a general purpose and self-adaptable algorithm to manage complex MEC systems without needing an explicit knowledge of all the involved aspects; we realized a Deep RL environment by integrating SimuLTE and Keras that can be used as a *gym* where different RL approaches can be realized and tested in LTE/5G scenarios.

The paper is organized as follows. Section 2 introduces the 5G and MEC technologies focusing also on the main challenges which is necessary to address in this context. Section 3 contains the problem formulation and the details about the algorithm we designed to train the proposed deep RL agent. Section 4 provides a description of the simulation environment that we used to retrieve the data necessary for the system training. Section 5 describes the scenario we realized and provides the results we obtained by comparing the policy learned by the agent with respect to a simple policy. Finally Section 6 concludes the paper and gives some details about future developments.

2. MULTI-ACCESS EDGE COMPUTING

Multi-access Edge Computing (MEC) is recently being standardized by the European Telecommunications Standards Institute (ETSI) [5]. It consists in placing nodes with computation capabilities, namely the MEC servers, close to the elements of the network edge like, e.g., base stations in a cellular environment. MEC differs from Fog Computing since it is able to interact with the network elements and can gather information on the network environment, such as resource utilization and users' location. This information can be obtained via a standard Application Programming Interface (API) and allows operators and/or third-party developers to offer new context-aware services and applications to the users. The latter can also experience lower latency and larger bandwidth with respect to a cloud-based application. Those services are created within a virtualized environment that allows to optimize the computational load of MEC servers and/or users' requirements. Moreover, MEC applications can also migrate between MEC servers to better accommodate mobile users, e.g. cellular users moving moving to another cell. Connected vehicles, video acceleration and augmented reality are some of the use cases identified by ETSI that can benefit from the introduction of MEC [6]. LTE is the cellular technology for which MEC was first proposed, and will be part (in its current state, or – more likely – leveraging an evolved physical and MAC layers) of the 5G ecosystem. In an LTE radio access network (see Fig. 1), eNBs create radio cells, to which User Equipments (UEs) attach. The entry/exit point for traffic in an LTE Evolved Packet System (EPS) network is the Packet Data Network Gateway (PGW). Traffic destined to the UE arrives at the PGW, where it is tunneled using the GPRS Tunneling Protocol (GTP). Tunneled traffic traverses the EPS network. The exit point of the GTP tunnel is on the serving eNB. At the eNB, the packet is extracted from the tunnel and transmitted over the air interface. A handover procedure is initiated by a UE to leave a cell and join another, typically when the signal strength of the new eNB exceeds the current one's. In this case, downstream traffic is steered from the PGW towards the new eNB, and the traffic still in transit at the handover may either be discarded or forwarded by the "old" eNB using the X2 interface, which connects eNBs in a peer-to-peer fashion. MEC servers can be deployed at any point in the EPS. However, it is foreseen that they will be either co-located with eNBs, in order to minimize the latency, or deployed close to them, so that a single MEC server will serve a geographic region area covering a limited number of contiguous cells. In these cases, functions are required to maintain service continuity and QoS for UEs using MEC services while in mobility. In fact, it may well happen that a UE moves too far away from its MEC server, in which case the communication latency increases beyond reasonable and context awareness becomes less effective. In such a scenario, it is preferable to move the server-side MEC application to a nearer MEC server. Besides providing functions to achieve this, policies are also required in order to understand when it is best to migrate a MEC application, and where to. These policies may take into account physical constraints (e.g., the availability of MEC services on particular servers), network-side QoS parameters (such as latency and communication bandwidth in the cell), server-side QoS parameters (such as available computation power and storage).

3. RL TECHNIQUES FOR MEC

During the last years, mobile traffic data is increasing and it is expected to raise in the next future. Such a trend, has lead to the start of a process which aims to improve the network infrastructure in order to address all the problems related to a such massive amount of data [7]. 5G system is expected to reach far better performance in terms of speed and energy efficiency if compared with the actual Long Term Evolution (LTE) and 3G technologies; 5G architecture takes advantage of Software Defined Networks (SDN) and Network Function Virtualization (NFV) in order to improve network management and dynamic resource allocation [8]. In such a context, where the network infrastructure is starting to become very complex, machine learning can be an useful technique to use in order to face the environment dynamics by finding an optimal strategy to improve the overall system performance.

3.1 Reference Scenario

The proposed reference scenario consists of a MEC-enabled LTE network where eNBs can be directly connected to MEC servers which contain application data, as depicted in Fig. 1. Such servers can be used as local repositories by serving those users attached to the corresponding eNB thus reducing the network latency as well as the amount of data traveling on the core network. One of the key issues of this schema is to find an optimal migration schema that opportunely moves data from one server to another in order to improve a specific objective function.

In particular, in this paper we are interested in the development of a machine learning algorithm based on reinforcement learning (RL) for the deployment of an optimal policy which establishes when is necessary to migrate data to another eNB depending on the user position and on the current state of the network in order to improve the user QoS. 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 many contexts where it is impossible to work with labeled data, RL is the only feasible solution to correctly train a system. In fact,

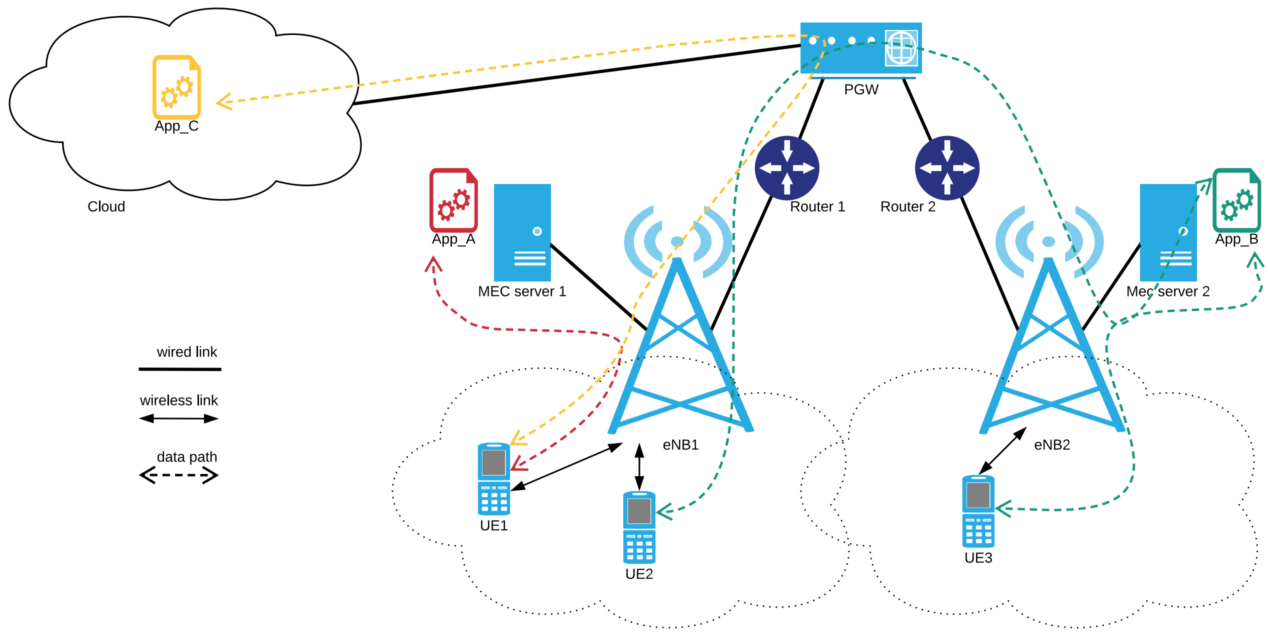

Figure 1 – The MEC-enabled LTE scenario.

its ability to learn through a trial and error process, which is very similar to the human being learning one, makes it the best choice to solve decision making problems. RL adopts, as a basic model, the MDP formalism which is a framework for modeling the decision making in stochastic environments. From a mathematical point of view, an MDP is defined as follows:

- set of states $\mathcal{S}$ that the environment can assume;
- set of actions $\mathcal{A}$ that the agent can perform;
- probability transition matrix P;
- reward function $R(s)$ $S : \rightarrow \mathbb{R}$
- discount factor $\gamma \in [0,1]$ which defines the importance we are giving to future rewards

In such a context, we define the figure of the RL agent whose objective is to find an optimal policy in order to maximize the reward function in each state of the environment. Using the MDP definition, the agent is able to run across the environment states several times and change the system policy to improve the reward accordingly. *Bellman equation* expresses the relationship between the utility of a state and its neighbors [9]:

$$U(s) = R(s) + \gamma \cdot max_{a \in \mathcal{A}} \sum_{s'} P(s'|s,a)U(s') \qquad (1)$$

where $U(s)$ is the immediate reward obtained in the state s assuming that the agent will choose the optimal action.
When the probability transition matrix is not known, a typical RL approach is the Q-learning [9], a *model free* technique which tries to learn the relationship between the execution of an action on a given state and the associated reward or utility through the concept of Q-value $Q(s,a)$ which returns the value of doing action a when the agent is in the state s:

$$Q(s,a) = Q(s,a) + \alpha(R(s + \gamma max_{a'}(Q(s',a') - Q(s,a)) \quad (2)$$

$$U(s) = max_a(Q(s,a)) \qquad (3)$$

where α is the learning rate and γ is the discount factor.
When the environment has a large number of states, it is impossible to use traditional Q-learning. In fact, the time to converge increases as the state space becomes larger, so if we consider a scenario with a huge number of states (e.g. 10^{20} or even more), it is impossible to think that the agent should visit all of them multiple times to learn a good policy [9]. One way to address this kind of problem is the quantization which consists in the process of grouping a set of states into a single one thus reducing the number of states which describe an environment. However, such a technique is not helpful especially in those cases where the state space reduction would result in a too large quantization error that could lead the agent to learn a wrong policy.
A very interesting alternative is provided by the use of a Deep Neural Network (DNN) to create a function approximator capable to predict the Q-values for a given state without explicitly using eqs. *((2))*-*((3))* and the corresponding MDP. Deep Reinforcement Learning is a technique which has been pioneered by DeepMind [10]. The idea comes from the necessity to find a new way to represent complex environments, where the dimension of the state space and action space is very large making impossible to solve them by using traditional approaches. The key idea at the base of the deep reinforcement learning consists in the use of two separated DNNs parameterized with θ and $\widehat{\theta}$ respectively: the *Main* DNN used to predict the Q values associated to a generic state and the *Target* DNN used to generate the target

Q values in the update rule which can be expressed as follows:

$$y_{j,a} = r_j + \gamma \max(\widehat{Q}(s_{j+1}, \widehat{\theta})) \tag{4}$$

where $y_{j,a}$ is the Q-value associated to the state j when the agent performs the action a,r_j is the reward associated to the state s_j, and $\widehat{Q}(s_{j+1}, \widehat{\theta})$ is the output layer of target DNN network which contains the Q-values estimation for the state s_{j+1}. By using two networks instead of just one, the obtained result is a more stable training which is not affected by learning loops due to the self-update network targets that sometimes can lead to oscillations or policy divergence.

3.2 Problem Formulation

In order to properly design a deep RL algorithm, it is first of all necessary to define the environment in which the RL agent will operate. In this subsection, we will formalize the MEC scenario we are interested in.

We start defining as $\mathcal{N}$ the number of eNBs present in the MEC-enabled LTE network. For the sake of simplicity and without loss of generality, let us assume that to each eNB is attached one and only one MEC server. Then, let us define the set, with cardinality $\mathcal{N} \in \mathbb{N}$, of the MEC servers attached to the eNBs as:

$$MEC = [MEC_1, MEC_2, MEC_3, ..., MEC_{\mathcal{N}}] \tag{5}$$

Users are free to move around the MEC environment and attach to different eNBs through seamless handover procedures. Moreover, they run several applications whose data is contained in the Cloud or inside one of the MEC servers. Regarding the applications that users can run and the number of devices attached to the eNBs, we can define the following sets:

$$Apps = [app_1, app2, app_3, ..., app_{\mathcal{M}}] \tag{6}$$

$$Devices = [UE_1, UE_2, UE_3, ..., UE_{\mathcal{K}}] \tag{7}$$

with $\mathcal{M}, \mathcal{K} \in \mathbb{N}$. With respect to the actions that the agent can perform in the environment, let us define the set of actions with cardinality $\mathcal{Z} \in \mathbb{N}$ as:

$$Actions = [a_1, a_2, a_3, ..., a_{\mathcal{Z}}] \tag{8}$$

where each element represents an app migration from the Cloud towards one of the servers defined in the MEC set, from one MEC server to the Cloud, or among MEC servers. Another important element that we have to introduce is the concept of *state* which is made of a t-uple containing all the information related to the users position and the app distribution over the network which is defined as follows:

$$s =< (s_1, s_2, s_3, ..., s_{\mathcal{T}}) > \tag{9}$$

Finally, with respect to the reward, we define it as a number $r \in \mathbb{R}$ that is computed as a combination of several network performance indexes.

Algorithm 1: *Deep RL*

```
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
6  set update step $U$
7  set waiting time $t$
8  set exploration rate $\epsilon$
9  set decay rate $d$
10 for episode = 1 to end:
11   observe current state $s_j$
12   $p$ = random([0,1])
13   if $\epsilon > p$:
14     action = random([1,$\mathcal{Z}$])
15   else:
16     action = argmax(Q($s_j$,$\theta$))
17   end if
18   execute the action
19   wait($x$ seconds)
20   observe the new state $s_{j+1}$
21   observe the reward $r$
22   store the t-uple ($s_j$,action,$s_{j+1}$,$r$) in $E$
23   sample a batch from $E$
24   y = Q($s_j$,$\theta$)
25   $y_{target} = \widehat{Q}(s_{j+1},\widehat{\theta})$
26   $y_{action}$ = r + $\gamma \cdot max(y_{target})$
27   execute one training step on main DNN network
28   every $U$ steps set $\widehat{\theta} = \theta$
29 end for
```

3.3 Proposed Algorithm

In this paragraph, we describe the proposed deep RL approach shown in Algorithm 1. Line 1 is dedicated to the set up of the replay memory E which is a data structure containing the experiences made by the agent. It plays a very important role since it stores all the data necessary for the DNN training. At the beginning, the two neural network weights are set to the same values (lines 2-3), then other parameters are set in lines 4-9. At each for-loop iteration, the agent observes the current state (line 11) and selects the action to perform depending on the exploration rate ϵ which establishes if the action has to be chosen randomly (line 12) from the action set we previously defined in Section 3.2 or if has to be returned by the main DNN (line 16). Then, after taking the action, the agent waits for x seconds (line 19) and observes again the state reached by the environment and the correspondent reward (lines 20,21). At this point the algorithm stores the experience made by the agent inside E (line 22). The core of the code is in lines 24-27. First of all, the main Q network predicts the Q-values for the given state s_j (line 24). In particular y is an array with a number of elements equal to the number of possible actions that can be executed. Then, target Q-values are evaluated through the target DNN network (line 25) and used in the *Bellman* update formula (line 26); to be more specific, only the Q-value related to the action sampled from the batch will be updated leaving the other values untouched. After the update, the main DNN network is trained by executing one training step on the cost function (line 27) and if U steps have been executed, the target DNN network weights are set equal to the main DNN network ones (line 28).

4. A SIMULATION ENVIRONMENT

Designing and testing deep RL algorithms requires the presence of an operative environment where status can be *sensed* and actions can be performed by receiving the corresponding rewards.

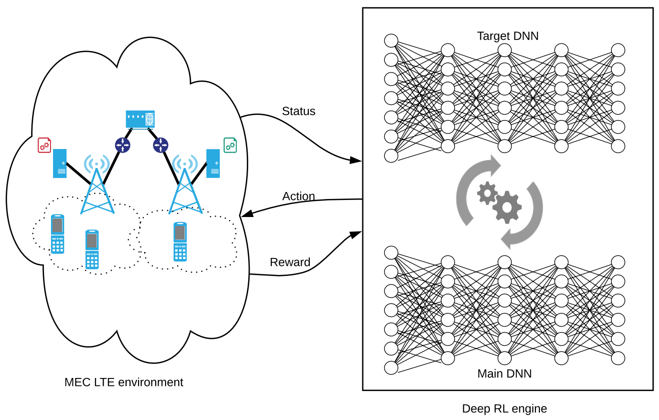

Figure 2 – Deep RL environment.

The general schema of a deep RL environment is depicted in Fig. 2 and consists of a *deep RL engine* where the algorithm (composed of the two DNNs) is running and an *environment* that in our case is composed of a LTE MEC with different MEC servers where different mobile users move while accessing some service. Due to the complexity of the LTE MEC environment and to the high number of actions that have to be tried by the deep RL algorithm before learning a good policy, we decided to simulate such an environment by using OMNeT++ [11]. The latter is a well-known event-driven simulator, written in C++, which allows us to model the network at a system level and a good level of detail, while scaling efficiently with the number of simulated nodes. Moreover, thanks to its large community of researchers and developers, OMNeT++ comes with a large set of pre-made and tested frameworks to simulate various portion of the network.

To model our MEC-enabled scenario, we used and integrate two of these simulation frameworks: SimuLTE [12] and INET [1]. The former models two main aspects of an LTE network, i.e., the 4G-based radio-access network and the EPC. The latter, instead, implements all the relevant TCP/IP protocols and layers, application and mobility models. In Fig. 3 we represent the high-level architecture and layering of the main communicating nodes, where grayed elements are from SimuLTE, whereas withe ones are from INET. The UE and the eNB nodes are provided with an LTE NIC, which provides wireless connectivity through the radio-access network, and implements a model of the LTE protocol stack, i.e. with PHY, MAC, RLC and PDCP, layers. Each UE can be configured with multiple TCP- or UDP-based applications, which can communicate both with the internet or with the MEC server(s). Moreover, the UE includes a module to model the mobility of the node itself. In the scenarios simulated within this paper, we used a waypoint-based mobility model, wherein UEs move linearly and at constant speed between randomly generated waypoints. The eNB is connected with the UE through the LTE NIC on one side, with the EPC through the GTP layer on the other. The EPC has also a model of the PGW (not shown in Fig. 3), and of the MEC server, which includes a complete TCP/IP stack. EPC nodes can be configured to have various L1/L2 types, e.g. based on PPP, ethernet, etc. Moreover, the parameters of the physical connections among nodes, such as bandwidth, delay, etc., can be configured to model various degrees of congestion within the network.

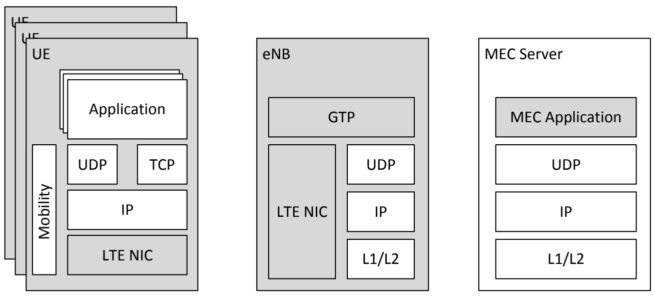

Figure 3 – High-level view of the simulator's architecture and layering. Grayed elements are from SimuLTE, white ones are from INET.

With respect to the deep RL engine, we used Keras [13] an open source library written in Python which runs on top of

[1] Available at "https://inet.omnetpp.org/", last accessed Jul 2018

machine learning frameworks like: TensorFlow, Theano and others. The main feature of this library is its simplicity, that allows to build complex neural network topologies with just a few lines of code in a *scikit-learn* fashion, but keeping at the same time the power of the neural network engine that runs underneath. Using Keras, we built a feedforward fully connected deep neural network composed of n hidden layers in between the input layer whose dimension is given by the cardinality $\|\mathcal{T}\|$ of the state t-uple, and the output layer whose dimension is given by the cardinality $\|\mathcal{Z}\|$ of the action set. In order to integrate the two systems which run respectively on Python and on C++ environments, we implemented a mechanism to let them communicate using text files. With reference to Fig. 2, the deep RL engine waits for the generation of the files containing the current state of the system, once it receives the data it generates as output a text file which contains the action to execute on the simulator. On the OMNeT++ side, we used an asynchronous timer which checks periodically for the action file availability, as soon as the file is available, the simulator is able to read the action code and change the server destination address for the *UEs* that are running the application indicated in the action thus emulating the data migration of the application from a server to another. After the action execution, the RL agent observes the reward obtained as the combination of several performance indexes provided by the OMNeT++ simulator by checking if the action performed has increased it or not. The reward in this sense is used by the agent as a feedback which helps it to understand if the action executed is a valid choice in that specific system state.

5. RESULTS

In this section, we present a preliminary scenario that we built to test the feasibility of the system where we only consider the presence of MEC servers without the possibility to use the Cloud. Fig.4 shows the structure of the network composed of three eNBs a set of devices with $\mathcal{K} = 9$, a set of MEC servers with $\mathcal{N} = 3$, a set of applications with $\mathcal{M} = 3$, and a set of actions with $\mathcal{Z} = \|\mathcal{N}\| \cdot \|\mathcal{M}\|$ where each action corresponds to the migration of an app taken from the *Apps* set to a server taken from the *MEC* set. The datarate connection provided by the cables which connect the eNBs is equal to 10 Gbps except for the ones that connect the routers to the PGW where the datarate is 3 Mbps to emulate a traffic congestion, thus creating a real scenario where we can test the performance of our algorithm. On the OMNeT++ side, it is possible to set several parameters for the simulation by using a configuration file called *omnetpp.ini*; since the number of parameters to set is very large, we synthesized them in a table.

Table 1 shows the main parameters we set for the simulation, we consider a total number of nine users who follow a random mobility motion pattern moving at speed equal to 1.5 mps which is a fairly good approximation for the human walking speed. With respect to the applications, we consider three constant bit rate applications (CBR) that can be run by only one MEC server per time. As already said at the beginning of this section, our goal is to test the feasibility of the technique we are proposing, for this reason, we decided to realize a scenario with manageable number of users and applications in order to keep the training time of the DNN not too high. In such a context, the state which defines the network can be expressed as follows:

Configuration Parameters	
$Number\ of\ users$	9
$User\ mobility$	$RandomWayPointMobility$
$User\ speed$	1.5 mps
$Number\ of\ applications$	3
$Application\ type$	$UDP\ ConstantBitRate$
$Packet\ size$	$1500B$
$Simulation\ Time$	$420\ seconds$

Table 1 – omnetpp.ini configuration.

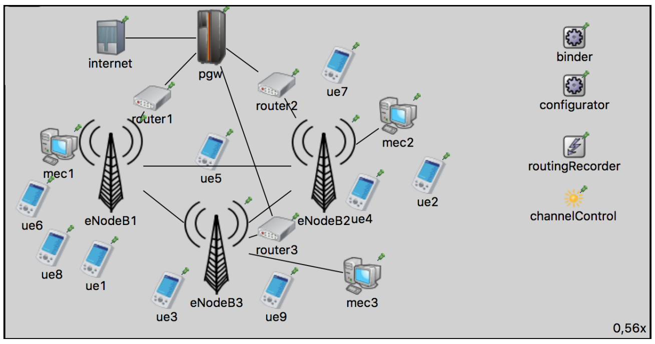

Figure 4 – OMNeT++/SimuLTE simulation scenario.

$$\begin{aligned} s =< (&UE_{eNB1}, UE_{eNB2}, UE_{eNB3},\\ &eNB^1_{app1}, eNB^1_{app2}, eNB^1_{app3},\\ &eNB^2_{app1}, eNB^2_{app2}, eNB^2_{app3},\\ &eNB^3_{app1}, eNB^3_{app2}, eNB^3_{app3},\\ &Mec^1_{app1}, Mec^1_{app2}, Mec^1_{app3},\\ &Mec^2_{app1}, Mec^2_{app2}, Mec^2_{app3},\\ &Mec^3_{app1}, Mec^3_{app2}, Mec^3_{app3}) > \end{aligned} \tag{10}$$

where:

- UE_{eNB_j} represents the number of devices connected to the j-th eNB;
- eNB^j_{appk} represents the number of devices which are running the k-th application in the j-th eNB;
- MEC^i_{appk} is a boolean flag that indicates if the i-th MEC server is running the k-th application.

With respect to the reward, we first defined as a QoS performance index the percentage of received data corresponding to the i-th application app_i as:

$$D^{app_i} = \frac{Received_{THR}}{\sum Sent_{THR} \cdot packetSize} \tag{11}$$

where the sum is extended to all the UEs that run the i-th application. In particular, we evaluated the average of the percentage of received data, for all the applications obtaining the final performance index value D. In order to observe the index after the action execution, we wait for ten seconds. Depending on the difference between the indexes evaluated before and after the execution on an action in the environment, we were able to define the reward as a number r which can assume the following values: [-1,-2/3,-1/3,0,1/3,2/3,1] where a value nearby one means that the action performed resulted in an improvement of system performance while a value nearby minus one means that the action performed resulted in a decrease of the system performance. For the sake of simplicity, we considered that UEs can only run one application per time. Our goal is to produce an optimal policy, which is able to address the problem related to the user's mobility inside the network in order to improve user QoS.

With respect to the DNN we designed, here we sum up the main parameters in the following table:

DNN parameters	
Number of hidden layers	3
Number of neurons	15
Input dimension	21
Output dimension	9
Learning rate	0.001
Activation function	*ReLU*
Update step	50
Batch size	32
Experience replay dimension	2000

Table 2 – Deep Neural Network parameters.

With reference to Table 2, by doing multiple tests, we were able to establish that 3 hidden layers create a good topology which is able to properly fit the desired output. Moreover, we fixed 15 neurons for each layer which is a number that stays in between the input layer dimension and the output one. With respect to the activation function, we used the *Rectified Linear Unit* (ReLU) which resulted in a faster learning if compared with other functions like the sigmoid. Since our DNN has to predict the state Q-values which are values defined in the set of $\mathbb{R}$, the problem we tried to solve is a *regression*. For this reason, the cost function we used is the *Mean Squared Error* (MSE) which is typical for this kind of problems and defined as:

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - \widehat{y_i})^2 \qquad (12)$$

where y is the real output and $\widehat{y_i}$ is the output predicted by the DNN. Regarding the update step, the batch size, and the experience replay dimension, the values we set has been obtained empirically by trying different values.

In Fig.5 we show a comparison between the policy learned after training for 25000 seconds of simulation our Deep RL algorithm and a scenario without any policy where we simply distributed one application for each MEC server. For a fair comparison, we used the same random seed in order to maintain in both simulation the same user mobility pattern.

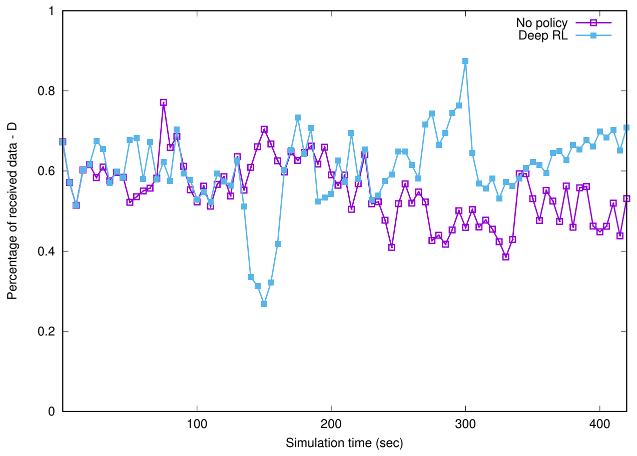

Figure 5 – Comparison between the performance obtained by the Deep RL policy and a scenario where the data migration is not enabled.

Plots show that the Deep RL algorithm is able to improve the overall system performance, in particular except for a little period between 100 and 200 seconds where the Deep RL algorithm encounters a little decrease (mainly due to the stochasticity of the environment), the results are in general good reaching an average of 0.60 which is better if compared with the no policy average equal to 0.54. As we are writing, we are trying to extend the training time with the aim to further improve the obtained results.

6. CONCLUSIONS

In this paper, we presented a deep reinforcement learning approach to address the problem related to the network environment dynamics. We designed a Deep RL algorithm and tested it in a real scenario demonstrating the feasibility of the technique. Future works will be devoted to implement a better integration with the OMNeT++ environment by using the Tensorflow C++ frontend, to compare with other solutions,to use more realistic traffic and mobility models, and to the investigation of new indexes with the aim to further improve the system performance.

REFERENCES

[1] D. Bruneo, S. Distefano, F. Longo, G. Merlino, and A. Puliafito, "I/Ocloud: Adding an IoT Dimension to Cloud Infrastructures," *Computer*, vol. 51, no. 1, pp. 57–65, January 2018.

[2] 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.

[3] 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.

[4] 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.

[5] ETSI, "Mobile edge computing (mec); framework and reference architecture," March 2016.

[6] ETSI, "Mobile edge computing (mec); service scenarios," Tech. Rep., November 2015.

[7] X. Xia, K. Xu, Y. Wang, and Y. Xu, "A 5g-enabling technology: Benefits, feasibility, and limitations of in-band full-duplex mmimo," *IEEE Vehicular Technology Magazine*, pp. 1–1, 2018.

[8] J. Li, Z. Zhao, and R. Li, "Machine learning-based ids for software-defined 5g network," *IET Networks*, vol. 7, no. 2, pp. 53–60, 2018.

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

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

[11] András Varga and Rudolf Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, ICST, Brussels, Belgium, Belgium, 2008, Simutools '08, pp. 60:1–60:10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[12] 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.

[13] François Chollet et al., "Keras," `https://keras.io`, 2015.