On real-time scheduling in Fog computing: A Reinforcement Learning algorithm with application to smart cities

Gabriele Proietti Mattia, Roberto Beraldi
Department of Computer, Control and Management Engineering "Antonio Ruberti",
Sapienza University of Rome,
Email: proiettimattia@diag.uniroma1.it, beraldi@diag.uniroma1.it

***Abstract*—Fog Computing is today a wide used paradigm that allows to distribute the computation in a geographic area. This not only makes possible to implement time-critical applications but opens the study to a series of solutions which permit to smartly organise the traffic among a set of Fog nodes, which constitute the core of the Fog Computing paradigm. A typical smart city setting is subject to a continuous change of traffic conditions, a node that was saturated can become almost completely unloaded and this creates the need of designing an algorithm which allows to meet the strict deadlines of the tasks but at the same time it can choose the best scheduling policy according to the current load situation that can vary at any time. In this paper, we use a Reinforcement Learning approach to design such an algorithm starting from the power-of-random choice paradigm, used as a baseline. By showing results from our delay-based simulator, we demonstrate how such distributed reinforcement learning approach is able to maximise the rate of the tasks executed within the deadline in a way that is equal to every node, both in a fixed load condition and in a real geographic scenario.**

***Index Terms*—fog computing, scheduling, real-time, reinforcement learning, smart cities**

I. Introduction

Fog Computing [1] is a well-known computing paradigm that is, not only, but usually chosen when the computation must be distributed in a geographic domain. This "must be" is generally given by the fact that the application has to be deployed as near as possible to users who have to use it. Indeed, in such situations, a cloud approach cannot be feasible, especially when the tasks that the application should carry out are strict in their deadlines, for instance, when we refer to a shared Virtual Reality (VR) [2] or Augmented Reality (AR) experiences. Distributing the load involves the setup of different computing nodes, called fog nodes, which, for example, can be spread across a city. Issues that arise within this setup essentially regard the fact that these nodes should be able to interact in some way in order to reach a common goal: each task requested to be executed by a user, to any fog node, has to be able to meet its deadline. This kind of interaction is needed not only because we want to create a sort of an ecosystem in which the application can live, so that it can be reachable in any fog node, but also because in such dynamic environments, for instance, some nodes can be overwhelmed by unpredictable traffic load, some instead can go down or others can become idle since no user is requesting them to execute tasks. Indeed, if we want them to be able to cooperate, a smart scheduling algorithm should be able to change its scheduling policy according to the current situation, by always having in mind the same previous goal. We suppose that each fog node receives requests to execute tasks from the clients and we need to make a scheduling decision on a per-task-request basis. This requires that, for making an optimal scheduling decision, we need to know the state of the other nodes. However, since this environment is completely distributed and decentralised, the only way for knowing the state (i.e. its current load level) of another node is to ask for it explicitly.

A well-known approach, that is proven to perform efficiently in this setting, is the power-of-random choice paradigm [3], where every scheduling decision, that is done on a per-task basis, is preceded by a random probing to another node, with the purpose to retrieve its current load. Once this information is retrieved, the task is scheduled internally or forwarded to the random-probed node. Executing a probing for each request is not always the best behaviour, indeed, adding a control threshold to decide when to trigger a new probing request [4] is shown to be an effective way to increase the performance over the standard approach. However, even this improved algorithm has limitations. For example, the scheduling policy (i.e. when to trigger the probing) is a fixed step function of the current load, e.g. the probing is performed only if the current workload exceeds the threshold. Moreover, it is also fixed over time, so it cannot react to load variation on the nodes, and finally, it does not take task heterogeneity into account. The purpose of this work is to overcome these limitations by designing a dynamic scheduling policy based on the Reinforcement Learning (RL) paradigm, where the probing decision is a function defined over the whole set of load states and the task performance requirements (expressed as deadlines).

We can summarise the main contributions of this work as follows.

- **Design of a decentralised RL-based algorithm** to be implemented in every fog node that is able to choose the best scheduling decision according to the current situation, which is a step forward the power-of-random choice approach, that allows for more **complex policies**

than simple and fixed threshold-based decisions, to be **dynamic over time** reacting to the current load situation of nodes and to deal with **different kinds of** time-constrained (i.e. with deadline requirements) **tasks**.
- **Study of a Geographic setting** which involves six fog nodes deployed in the city of New York and in which the algorithm can be deployed.
- **Simulation Results** on a delay-based simulator which prove the efficiency of the algorithm in a previously defined geographic environment compared to the classic power-of-choice strategy.

The rest of this paper is organised as follows. In Section II we present some related works, in Section III we define the system model, instead in Section IV we describe the reinforcement learning approach that we propose. Finally, we draw conclusions in Section VI.

II. Related Work

Among the different works in literature that try to solve the task scheduling problem in a distributed environment by using a reinforcement learning approach [5], one of the first approaches to this kind of problem, that is also called job-shop scheduling, is quite old [6]. Focusing on Fog Computing, instead, in [7], the authors present a Deep Reinforcement Learning approach, based on Q-Learning, in a MEC environment, for selecting the best edge server for offloading in order to minimise the energy consumption (also studied in [8]) while at the same time maximising the number of tasks that meet the deadline. In this work, two DNNs are used: one is kept fixed during an episode, while the other is updated and at the end of the episode they are swapped. Authors of [9] propose a scheduling scheme based again on two DNNs, but they are used for two different decisions, the first one is in charge of deciding if the task should be offloaded to the cloud, but if not, the second decision level chooses the best suitable Fog node to which schedule the task. The approach followed by [10], but in the context of MEC cells, is the one of defining the reward as the weighted sum of energy consumption, delay and cache fetching cost, then, a Deep Reinforcement Learning approach is followed but by using a Deep Deterministic Policy Gradient method which solves the problem of the state discretisation in the standard Q-Learning approach. Instead, [11] focuses on real-time task assignment but considers the evolution strategies approach instead of the backpropagation for updating the weight of the DNN. In [12], an approach based on a recurrent neural network (RNN) is proposed, [13] illustrates a solution that targets explicitly vehicular networks, and [14] instead to crowdsensing. In a broader sense of scheduling, other works are instead focused on resource allocation [15] but the task model does not fit the one that is studied in this paper. Finally, [16] uses the Asynchronous Advantage Actor-Critic (A3C) algorithm in an Edge-Cloud environment.

To summarise, the main points of novelty in our work lie in different facts. First of all, we do not rely on Deep Reinforcement Learning, which is not needed since we reduce as much as possible the state space. Other works that use the same approach like [17] and [18] use the Q-Table, but the first is focused on connected vehicles and the second on Edge computing and energy consumption. Instead, in this work, we focus on real-time tasks training the learner according to the hit of a task completion deadline.

III. System Model and Problem Definition

In order to reach our goal of adaptability and optimality, we frame our problem as a Markov Decision Process (MDP), which is solved using the Reinforcement Learning (RL) technique. We use a model-free approach with the advantage that it does not require the knowledge of the details of the underlying mathematical model, such as the state transition probabilities. Instead, it is enough to observe and interact with the environment. To proceed with our discussion, we need preliminarily to identify the two main entities of RL: the environment and the agent.

A. Environment

The environment is composed of a set of N communicating and nearby Fog nodes $\mathcal{F} = \{F_1, F_2, \ldots, F_N\}$ with the same computing power. Every Fog node serves an area from where users can require the execution of a task, and we define the total rate of the arriving requests to a node i to be λ_i req/s. One example of an application scenario for such tasks is Virtual Reality (VR), where a user needs to execute compute-intensive tasks like recognising and tracking objects or activities. Tasks have a deadline T associated with them, which represents the absolute time before which they must be processed, e.g. 10ms in a VR scenario [2]. They also have associated a payload of size b (e.g. an image, a set of video frames).

A Fog node F_i has a performance profile defined by its queue capacity K_i, representing the maximum number of pending tasks waiting to be processed, and the rate of execution of the tasks that is μ task/s that is supposed to be equal for each node. A fog node is capable of executing one task at a time, but since it has a queue of K_i this is exactly equal to saying that it can execute K_i tasks at a time in time-sharing with no queue, that is a mechanism closer to reality.

The total load to a specific node i is:

$$\rho_i = \frac{\lambda_i}{\mu} \tag{1}$$

Nodes can communicate with each other and each transmission requires a time interval d_t, determined by the data rate, r of the link connecting the two communicating nodes ($d_t = \frac{b}{r}$). Moreover, node A can probe another node B, meaning that A can ask B its current queue length to make a scheduling decision. The queue length of a node is usually referred to as its current "state".

B. Agent

Upon each task request arrival to a Fog node from a client, a scheduling decision must be taken. This is an online decision process carried out by an agent, running at each Fog node. Each agent has associated the same set of actions $\mathcal{A}$ that can

perform. The action $a \in \mathcal{A}$ to take is determined by a function $\pi(s)$, called *policy*, of the current observed state $s \in \mathcal{S}$.

$$\pi : S \rightarrow \mathcal{A} \tag{2}$$

1) Observed states: The agent running on node F_i can observe its current state, referring to the number of tasks in its execution queue Q_e at time t, i.e. $k_i^t \leq K_i$. Moreover, we consider the case in which tasks of multiple types can arrive in the node. For this reason, the final observed state by the agent is the aggregation of the number of tasks in the queue for each type and the type of the newly arrived task. This means that the decision about the action to choose is only made by observing the queue length of the current node.

Finally, the state derived as described and used for the learning process is not taken as is. Indeed the tiling technique [19] is used for representing it as a vector $v \in \mathbb{N}^8$.

2) Actions: We studied separately two sets of actions that the agent can perform. In the first case (i), the agent selects an action from the set $\mathcal{A} = \{0, 1\}$, where 0 means to execute the task locally, while 1 to probe another node at random and offloading the execution of the task to that node only if its queue length is lesser than the one of the current node (we call this strategy "probe-and-forward") otherwise the task is executed locally. In the second case (ii) instead, the agent of node i selects an action from $\mathcal{A}'_i = \mathcal{A} \cup \mathcal{F}_i$, where $\mathcal{F}_i$ contains additional direct forwarding actions that depends on the Fog node i. The action $f_j \in \mathcal{F}_i$ means to directly forward a task to the neighbour j without probing, obviously with $j \neq i$.

It is worth reminding that, in any case, when the task is scheduled to be executed locally, despite being forwarded from another node, it can be rejected if there is no room for being executed, i.e. the queue is full (at a certain time t, $k_i^t = K_i$).

C. Reward

The immediate reward given to a specific action is given by the fact that the task has been executed within the deadline or not. Therefore, the reward is a function of the state and the action performed given the state. For a given task j, the reward assigned to action a performed when the state was s is, given the total completion time W:

$$R_j(s, a) = \begin{cases} 1 & \text{if } W \leq T \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The agent cannot know the reward until a task has been completed its execution path, and in the meanwhile, other tasks may arrive and need to be scheduled. Moreover, even if two tasks are scheduled sequentially, the second can terminate before the first, altering the causal order for the decision path. To overcome this problem, the learning step is put on hold until a number equal to Z (the window size) of tasks have been completed. This is discussed in Section IV.

For measuring the performances of the algorithm, we use the reward rate, also called the in-deadline rate ι that is the average reward per second.

D. Delay model

Dealing with deadlines requires a fine-grained model of the delays. In our study, the environment is simulated as a network of N nodes in which every node is composed of three different internal queues:

- the execution queue (Q_e) represents the queue of tasks that have been scheduled to be run in the current node; a node can execute one task at the time and the total time that a task spends on this queue is d_e, but the actual execution time of a single task follows a Gaussian distribution;
- the transmission queue (Q_t) represents the queue of tasks that are in the transmission phase; the transmission can occur: (i) from client to node, (ii) from node to node, (iii) from node to client. The total time that a task spends on this queue is d_t, and it follows a Gaussian distribution with μ equal to d_t;
- the probing queue (Q_p) represents the queue of tasks for which there is a probing request to run; the total time that a task spends on this queue is d_p;

The flow according to which a task transits among the queues is represented in Figure 1.

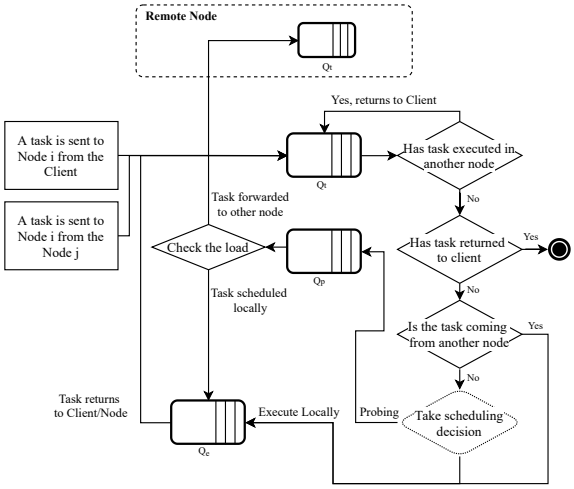

Fig. 1. The logic of the delay model.

The total time of a task to be executed, from the client perspective, is the summation of all the time spent in all the queues during its entire execution path, it is referred as W and it is measured in seconds.

E. Geographic Scenario

A peculiar characteristic of Fog computing is that nodes can be positioned in a geographic scenario [1]. In order to evaluate the adaptivity of our solution in a real environment, we used open data of New York city[1] to estimate the average daily traffic in specific points of the city. The data is referred to taxi trips of the year 2013, and for every trip, we considered the

[1]https://web.archive.org/web/20210424121526/https://chriswhong.com/open-data/foil_nyc_taxi/

start coordinates, the end coordinates, and the total trip time. Then we placed six Fog nodes, as in Figure 2, considering 1 km of radius for the service to be available. This is in line with the capability of an RRU to which has been attached a computing node. We estimated the taxi traffic by dividing the day into 15-minute time slots (for a total of 96 time slots) and counting the number of taxis within the area of the nearest Fog nodes during their trip. We used the first three months of data by averaging the daily traffic within each time slot for every Fog node.

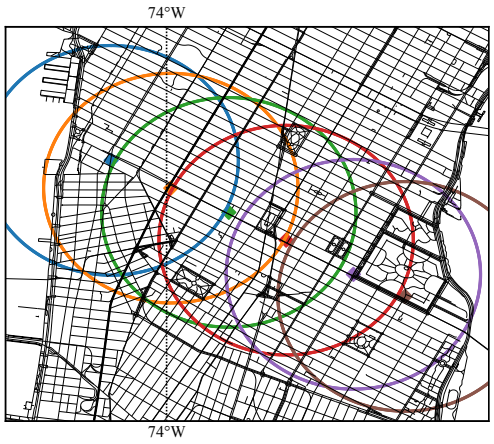

Fig. 2. Fog nodes position (diamond symbols) in New York city used in the experiments, from left to right Node 0 to Node 5. The radius of the circle for each node is 1 km.

By normalising in the range between 0 and 0.9 the final traffic distribution is represented in Figure 3. This range is given for simplicity and derives from a reasonable assumption that Fog nodes never saturate, leaving the opposite case as future work. Moreover, the final curves have been smoothed with Savitzky–Golay filter, using an order 4 polynomial and a window of size 17.

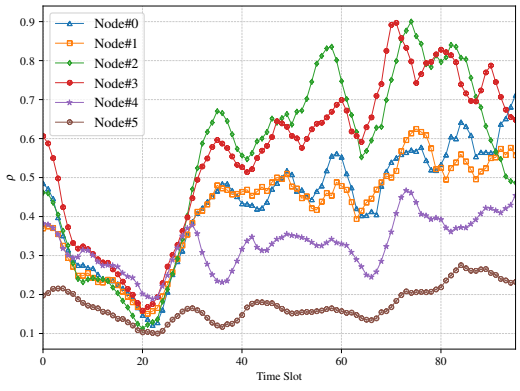

Fig. 3. The average distribution of the traffic during the day for the picked Fog nodes.

The final result of this study has been used in our simulation environment by setting the load to a specific Fog node i (ρ_i) to be equal to the value of the respective traffic curve (Figure 3) in that precise moment of the simulation.

IV. Online scheduling decisions with RL

The agent's final objective is to learn a scheduling policy π that maximizes the long-term reward. Since each decision must be taken online, we cannot envision episodes, but we treat the problem as a continuing learning task. It is not useful to discount future rewards in this specific setting, but it is better to consider the current average reward for taking the right direction. Given a state $s \in \mathcal{S}$, we perform the action $a \in \mathcal{A}$, we obtain the immediate reward r the next state is $s' \in \mathcal{S}$ then the optimal policy (that is the policy which maximizes the long-term reward) will result in the optimal q_* function defined as [19]:

$$q_*(s,a) = \sum_{r,s'} p(s', r|s, a) \left[r - \max_{\pi} r(\pi) + \max_{a'} q_*(s', a') \right] \tag{4}$$

Where $r(\pi)$ is a function which returns the average reward of the policy π. At certain time t and given a weight's vector $\vec{w}$ the differential form of the error, following the Sarsa approach for learning the policy, can be expressed as [19]:

$$\delta_t = R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \vec{w}_t) - \hat{q}(S_t, A_t, \vec{w}_t) \tag{5}$$

When used in practice, the $q(s, a, \vec{w})$ is approximated by using the linear combination of the coordinates given by the tiling technique, as described in Section III-B. The final used strategy for learning the policy is called Differential Semi-Gradient Sarsa (Algorithm 2). However, the reward is never immediate (after the learner's decision) in our setting because we know it only after a task has been executed or rejected. For this reason, we set a window size of Z tasks, and right after executing every task, we check if the window is reached and every task in the window has been executed or rejected.

The Algorithm 1 is run whenever a new task to be executed arrives. First of all, we append the task to the array of pending tasks ("TasksArray"), then we compute the state (as described in Section III), and we retrieve the best action to perform given the current $q(s, a, \vec{w})$. If the action is 0, the task is immediately executed locally. If it is 1, the node asks the state to a random node, and the task is forwarded only if the random node's state is better than the current one. These two actions are of $\mathcal{A}_1$, and in any other case, the task is directly forwarded to the chosen node ($\mathcal{A}_2$) unless the picked node is the current one, in that case, the function *forwardTo()* only executes the task locally.

Every time that a task completes its execution (that means that the result payload of the task is returned to the client), whether it is local or remote Algorithm 2 is executed. First of all, we record the task reward, and then we start to iterate over the array of pending tasks ("TasksArray") for checking if the first Z tasks of the array are finished. If this is not the case, the function returns, otherwise, we go on by retrieving the information about the first Z tasks by popping them from

Algorithm 1 Scheduling Decision

Require: Node, Task, TasksArray, $\vec{w}$, $\mathcal{A}$
  TasksArray.append(Task)
  $s \leftarrow$ aggregate(Node.getLoad(), Task.getType())
  $a \leftarrow \max_{a \in \mathcal{A}} q(s, a, \vec{w})$ with prob. $1 - \epsilon$ otherwise random($\mathcal{A}$)
  Task.saveStateAction(s, a)
  if a == 0 (*Execute Locally*) **then**
    Node.execute(Task)
  else if a == 1 (*Probe-and-Forward*) **then**
    RandomNode $\leftarrow$ pickRandom(Node.getNeighbors())
    if RandomNode.getLoad() $<$ Node.getLoad() **then**
      forwardTo(RandomNeighbor, Task)
    else
      Node.execute(Tasks)
    end if
  else
    Node $\leftarrow$ pickNodeFromAction(a)
    forwardTo(Node, Tasks)
  end if

the array. This information is used to train the weights vector $\vec{w}$ using the semi-gradient differential Sarsa algorithm.

Algorithm 2 Learning with Differential Semi-Gradient Sarsa

Require: Task, TasksArray, Z, $\vec{w}$, $\bar{R}$, α, β
  Task.setReward()
  i $\leftarrow$ 0
  for all j in TasksArray **do**
    if !j.isDone() **then**
      return
    end if
    if i == Z **then**
      break
    end if
    i $\leftarrow$ i + 1
  end for
  i $\leftarrow$ 0
  $j_0 \leftarrow$ TasksArray.pop(0)
  $s \leftarrow j_0$.getStateSnapshot()
  $a \leftarrow j_0$.getAction()
  $r \leftarrow j_0$.getReward()
  for i = 0; i $<$ Z; i++ **do**
    $j \leftarrow$ TasksArray.pop(0)
    $s' \leftarrow j$.getStateSnapshot()
    $a' \leftarrow j$.getAction()
    $\delta \leftarrow r$ - $\bar{R}$ + $q(s', a', \vec{w})$ - $q(s, a, \vec{w})$
    $\bar{R} \leftarrow \bar{R}$ + $\beta\delta$
    $\vec{w} \leftarrow \vec{w}$ + $\alpha\delta\nabla q(s, a, \vec{w})$
    $s \leftarrow s'$; $a \leftarrow a'$; $r \leftarrow j$.getReward()
  end for

V. Results

The results that we will present in this section follow the assumption that there are 6 Fog Nodes ($N = 6$), and for every fog node i, the maximum queue length is $K_i = 5$. Moreover, the arrival distribution of the tasks is a Poisson with mean μ_i. We suppose that nodes are connected with a link of 1Gbps, the payload of each task is 100kb, and the probing delay is 5ms. Q_t and Q_p are unlimited in size for each node. The rationale behind this number of nodes is that to avoid a high offloading delay among nodes, the cooperating nodes are physically close one with the other, and hence they amount to a few units. In other words, cooperation occurs only among nearby nodes.

In all of these experiments, the proposed RL algorithm is labelled as "Sarsa" while the power-of-choice one "Pwr2", that is specifically referring to the power-of *two* choices since only one node is probed random and therefore, the scheduling decision is between the current node (possible choice #1) and the probed one (possible choice #2).

A. Homogeneous Loads

The first approach that we followed is supposing the load is constant to every fog node. The learning agent AG, that we remind is present in every Fog node, can choose among the two actions in $\mathcal{A}$. Then, if the agent chooses to probe a random node, only if the queue length of the remote node is lesser than the current one, the job will be forwarded to it (we call this approach, "probe-and-forward") otherwise, the task will be executed locally. In this context, the policy π can be easily binary encoded $\pi = (a_0, a_1, \dots a_K)$, where $a_i \in \mathcal{A} = \{0, 1\}$ is the action executed when the state of the agent is i. For example, for $K = 5$ the policy 000111 means that the task is executed locally when the state is 0, 1, or 2 (action 0), and forwarded otherwise (action 1).

For comparison with the learning algorithm, we tested every possible scheduling policy statically, without the learning framework but leaving all the infrastructure valid. This means that the environment, the set of actions and the reward function are the same, and we only consider the policy static. We set the mean task duration for every fog node i as $1/\mu_i = 0.023s$, the load as $\rho_i = 0.6$ and the deadline $T = 0.042s$. The bar plot in Figure 4 compares the reward of each possible policy, expressed as in the form of rate of in-deadline tasks ι. We can observe that the optimal policy is 001111 ($\iota = 0.8164$) that is the one of performing the random probing if the load is greater or equal to 2. This is a sort of cooperation threshold that perfectly matches the results of [4], although it considered no deadlines. However, this policy is very similar, in terms of performances, also to 001110 ($\iota = 0.8147$), 001100 ($\iota = 0.8145$) and 001101 ($\iota = 0.8114$) which respectively disable the cooperation when the load is equal to 5, 4 and 5, and only 4. This is justified by the fact that the rewards gained in such situations that are referring to load values greater than 4 are negligible with respect the other states.

In the same set-up, we used the learning approach described in Section IV. At the end of the simulation, the policy learned by the learner is the one in Figure 5, i.e. 001110. This is not the optimal policy, but this is again justified by the fact that the contribution of state 5 is so small that is not appreciable by the learner. For this reason, we defined decision confidence by just taking into account how many times the decision has been made in each possible state. Given that s_j is the number of tasks whose scheduling decision has been made when the state was j, and J the total number of jobs, the value in the cell i, j of the heat map is given by the following choice confidence function of the action in i when in state j:

$$C(i,j) = \begin{cases} \frac{s_j}{J} & \text{if } i = 0 \text{ (Non-Probe)} \\ -(1 - \frac{s_j}{J}) = \frac{s_j}{J} - 1 & \text{if } i = 1 \text{ (Probe)} \end{cases} \quad (6)$$

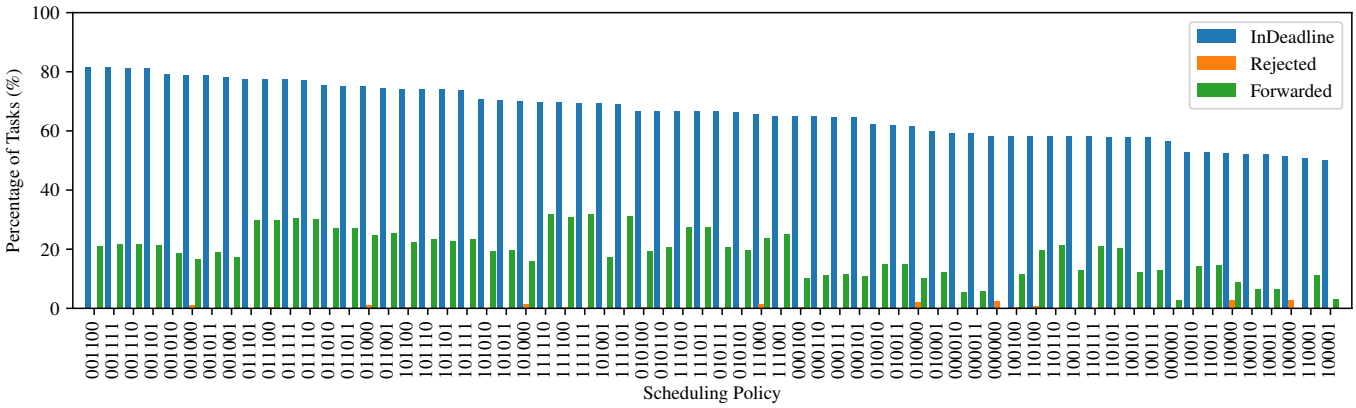

Fig. 4. Percentage of In-Deadline, Rejected and Forwarded tasks of all the possible policies with 6 nodes, a policy is encoded in binary where 1 means probing and 0 means executing locally. In this experiment $\rho = 0.6$, deadline is = 0.043s and job duration is 0.020s.

This formulation allows us to understand the confidence in the choice of the agent. In particular, from Figure 5 we can appreciate how the choices in the lower states are more confident since the states have been more frequent.

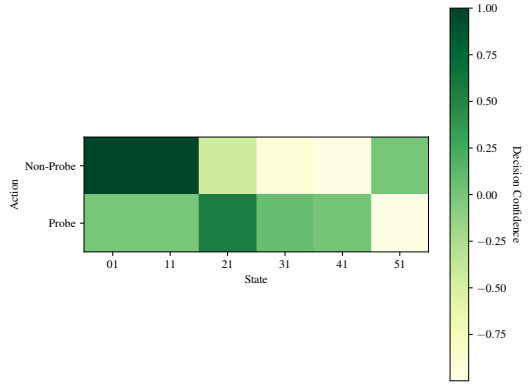

Fig. 5. The policy learned by the agent in the same setting of Figure 4

B. Heterogeneous Loads

During the previous experiments, we assumed only one kind of task, which has deadline 43ms and its average duration is 20ms. By using the same framework, we switched to two kinds of tasks, one (type 0) that is expected to run at 60fps, and therefore we supposed that it has a deadline of 16ms and mean duration 8ms ($\sigma = 0.4$ms) and one (type 1) that is expected to run at 30fps and therefore it has deadline 40ms and mean duration 20ms ($\sigma = 0.4$ms). Increasing the number of tasks does not lead to validity issues, instead, it leads to the increasing of the state space, but this will be further investigated in future works.

When dealing with heterogeneous loads, the best policy learned by the agent is still the one of power-of-choice with a threshold equal to two. We do not report the results in this paper, but we directly show we can achieve better performances. A possible way for achieving better results is to increase the agent's action space. For this reason, we introduced the set of actions $\mathcal{A}'_i$. Figure 6 shows the behaviour of the in-deadline rate when the loads are not balanced, as in the previous experiment, but stationary over time. The only difference here is that the agent can choose to directly forward tasks to a given node. After the first 1000s, which is the period in which the ϵ is greater than 0.1 (the lower limit), we can observe how ι is fixed over time, and it is equal to every node. This means that even the nodes with the policy 001111 could have a better reward. They are not selfish but voluntarily decrease their reward for making the others achieve the best reward. We believe that this behaviour is inherent to the distributed usage of the reinforcement learning approach since every node can understand the situation only from the reward, which declares the goodness of the chosen action. This means that by allowing the nodes to forward directly (set of actions $\mathcal{A}'_i$), we make them understand which is the best node to forward the jobs in a given time, and this enables the fact that acting like selfish will deteriorate its reward. This aspect will be further studied in future work.

C. Geographic Scenario

As described in Section III the open data for New York City has been used for generating the traffic to six fog nodes positioned as in Figure 2. Starting from this setting, by using the same assumptions of the previous experiment, we only enable the load to change according to the derived distribution for the open data (Figure 3). In Figure 7 we can observe that the behaviour that was shown in the fixed load case is again confirmed even if the load follows a variable distribution. Every node reaches the same level of the reward, even if following the policy 001111 (power-of-choice with threshold 2) could make a node reach a better reward, and this behaviour is invariant to the traffic variability.

VI. Conclusions and Future Work

In this paper, we addressed the problem of extending the power-of-choice distributed scheduling scheme with Reinforcement Learning to schedule real-time and deadline-constrained tasks efficiently. Starting from the simple approach

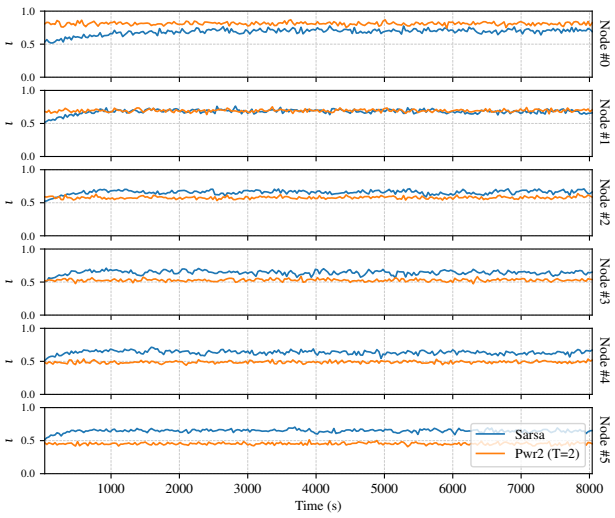

Fig. 6. Comparison between Sarsa and Pwr2: behaviour of the in-deadline rate ι for every node when load is fixed but heterogeneous

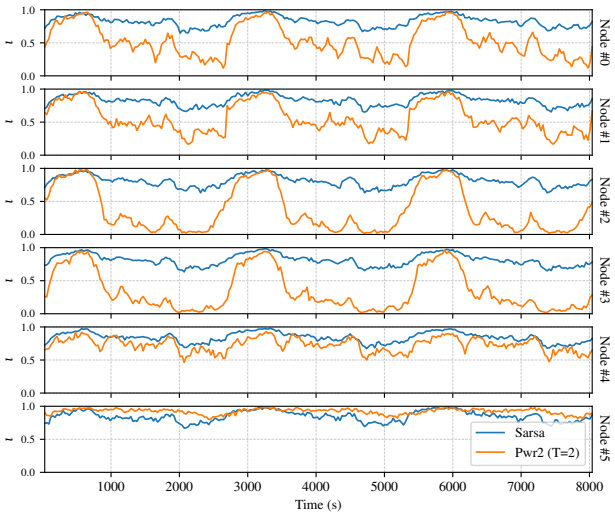

Fig. 7. Comparison between Sarsa and Pwr2: behaviour of the in-deadline rate ι for every node when the load is variable according to the geographic scenario, Figure 3

in which the agent learns a known policy, we arrived to provide simulation-based results that the approach works even if the load conditions are typical of an actual fog deployment in a smart city. We showed that a fully distributed scheduling approach based on reinforcement learning, in which every node is an agent and does not have any load information about the others, can maximise the performances of every single node by not behaving selfishly. However, other environmental characteristics can be studied in order to reveal the actual efficiency of the approach, for example, by introducing a variable communication delay between the nodes, considering that the nodes maintain a periodically updated value of the load of the others, or even increasing the complexity of the state in order to take into account other factors like CPU time and RAM consumption.

References

[1] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi, "Fog computing conceptual model," NIST, Tech. Rep., 2018.

[2] F. Hu, Y. Deng, W. Saad, M. Bennis, and A. H. Aghvami, "Cellular-connected wireless virtual reality: Requirements, challenges, and solutions," *IEEE Communications Magazine*, vol. 58, no. 5, pp. 105–111, 2020.

[3] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001.

[4] R. Beraldi and G. P. Mattia, "Power of random choices made efficient for fog computing," *IEEE Transactions on Cloud Computing*, 2020.

[5] A. I. Orhean, F. Pop, and I. Raicu, "New scheduling approach using reinforcement learning for heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 292–302, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301521

[6] M. E. Aydin and E. Öztemel, "Dynamic job-shop scheduling using reinforcement learning agents," *Robotics and Autonomous Systems*, vol. 33, no. 2-3, pp. 169–178, 2000.

[7] L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu, and L. Li, "Delay-aware and energy-efficient computation offloading in mobile edge computing using deep reinforcement learning," *IEEE Transactions on Cognitive Communications and Networking*, pp. 1–1, 2021.

[8] Q. Yang and P. Li, "Deep reinforcement learning based energy scheduling for edge computing," in *2020 IEEE International Conference on Smart Cloud (SmartCloud)*, 2020, pp. 175–180.

[9] M. K. Pandit, R. N. Mir, and M. A. Chishti, "Adaptive task scheduling in iot using reinforcement learning," *International Journal of Intelligent Computing and Cybernetics*, 2020.

[10] S. Nath and J. Wu, "Deep reinforcement learning for dynamic computation offloading and resource allocation in cache-assisted mobile edge computing systems," *Intelligent and Converged Networks*, vol. 1, no. 2, pp. 181–198, 2020.

[11] L. Mai, N.-N. Dao, and M. Park, "Real-time task assignment approach leveraging reinforcement learning with evolution strategies for long-term latency minimization in fog computing," *Sensors*, vol. 18, no. 9, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/9/2830

[12] S. Bian, X. Huang, Z. Shao, and Y. Yang, "Neural task scheduling with reinforcement learning for fog computing systems," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[13] J. Zhang, H. Guo, and J. Liu, "A reinforcement learning based task offloading scheme for vehicular edge computing network," in *Artificial Intelligence for Communications and Networks*, S. Han, L. Ye, and W. Meng, Eds. Cham: Springer International Publishing, 2019, pp. 438–449.

[14] H. Li, K. Ota, and M. Dong, "Deep reinforcement scheduling for mobile crowdsensing in fog computing," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, pp. 1–18, 2019.

[15] A. Mseddi, W. Jaafar, H. Elbiaze, and W. Ajib, "Intelligent resource allocation in dynamic fog computing environments," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, 2019, pp. 1–7.

[16] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2020.

[17] S. Park and Y. Yoo, "Real-time scheduling using reinforcement learning technique for the connected vehicles," in *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, 2018, pp. 1–5.

[18] T. Sen and H. Shen, "Machine learning based timeliness-guaranteed and energy-efficient task assignment in edge computing systems," in *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, 2019, pp. 1–10.

[19] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.