A Double-decision Reinforcement Learning based Algorithm for Online Scheduling in Edge and Fog Computing

Ahmed Fayez Moustafa Tayel[0009−0007−0333−3918], Gabriele Proietti Mattia[0000−0003−4551−7567], and Roberto Beraldi[0000−0002−9731−6321]

Department of Computer, Control and Management Engineering "Antonio Ruberti",
Sapienza University of Rome, Rome, Italy
tayel.1972085@studenti.uniroma1.it,
{proiettimattia,beraldi}@diag.uniroma1.it

Abstract. Fog and Edge Computing are two paradigms specifically suitable for real-time and time-critical applications, which are usually distributed among a set of nodes that constitutes the core idea of both Fog and Edge Computing. Since nodes are heterogeneous and subject to different traffic patterns, distributed scheduling algorithms are in charge of making each request meet the specified deadline. In this paper, we exploit the approach of Reinforcement Learning based decision-making for designing a cooperative and decentralized task online scheduling approach which is composed of two RL-based decisions. One for selecting the node to which to offload the traffic and one for accepting or not the incoming offloading request. The experiments that we conducted on a cluster of Raspberry Pi 4 show that introducing a second RL decision increases the rate of tasks executed within the deadline of 4% as it introduces more flexibility during the decision-making process, consequently enabling better scheduling decisions.

Keywords: Fog Computing · Online Scheduling · Distributed Scheduling · Reinforcement Learning.

1 Introduction

Fog Computing is a paradigm that links edge devices to cloud computing data centers by offering processing, storage, and networking resources [14]. It might be viewed as an addition to cloud computing rather than as a substitute, and its main objective is to support services and applications that are not supported by cloud computing [6], such as those which need predictable and low latency, geographically distributed, require quick responses from mobile devices, and require large-scale distributed control systems. The three primary layers of an IoT environment (Edge Processing, Fog Computing, and Cloud Computing) can be considered a hierarchical arrangement of network resources, computing, and storage.

Recently Fog Computing applications drastically increased due to the unique solution it provides with time-critical applications [12], as over the past few years, cloud computing and native cloud services were used for deploying applications due to the fast deployment methods, and the provided scalability options by the cloud providers, in addition to the ready maintained packages for configuring the infrastructure according to every application-specific needs. One of the important flaws to be considered in this approach is that the application is deployed in one of the cloud provider's centers which are placed evenly all over the globe to serve applications from anywhere in the world, and it does not behave very well with the time-critical applications due to network (and geographic) latency. Use cases like real-time face detection applications, machine learning-based applications, or general applications which are time critical [10], [5], [7] and need some computational effort would not get much use of the basic cloud services, this is when fog computing comes into action. Fog computing would be an optimal solution for applications and use cases which are time critical, need some computational power to be executed successfully and most importantly could be executed in a distributed fashion by dividing and delegating the tasks to other geographically nearby servers [23], [2], [9] to execute the function in the shortest time possible, this has multiple benefits, the work could be split among different servers which speed up the execution process, also the tasks allocations process [21] could be scheduled in an optimized way to serve the overall execution of the task with respect to multiple things, the servers geographical locations, the complexity of the tasks, and the servers capability in terms of memory and computational power. Many scheduling algorithms were used to solve this problem, and most focused on using Reinforcement Learning approaches [15] which, in this particular context, allows for adapting the scheduling policy in a very dynamic context that can regard Edge or Fog Computing, where nodes can unpredictably go down or saturate due to high traffic.

This work focuses on designing an improved approach [16] for a cooperative, decentralized, and online scheduling of tasks among a set of nodes using Reinforcement Learning. We suppose that clients request tasks to be executed on the nodes, and each task has defined a specific deadline that makes the task usable by the clients. Each node can be seen as a worker and a scheduler that takes decisions according to the RL model. The model is used to make two kinds of decisions. Figure 1 shows our double-step decision model during the cooperation. Firstly, when (1) Node A (which we call the "originator node") receives a task to be executed by the client, the node decides whether to execute, reject or forward it to another neighbor node, suppose Node B (which we call the "delegated node"). When the task is forwarded, even node Node B decides whether to execute it locally or reject it. Both decisions are made with an online learning RL model, and we suppose a task cannot be forwarded more than once since every forwarding step adds network delay to tasks that already have a deadline.

The rest of the paper is organized as follows. In Section 2, we present some related work, then in Section 3, we show the model of the system, in Section 4, we
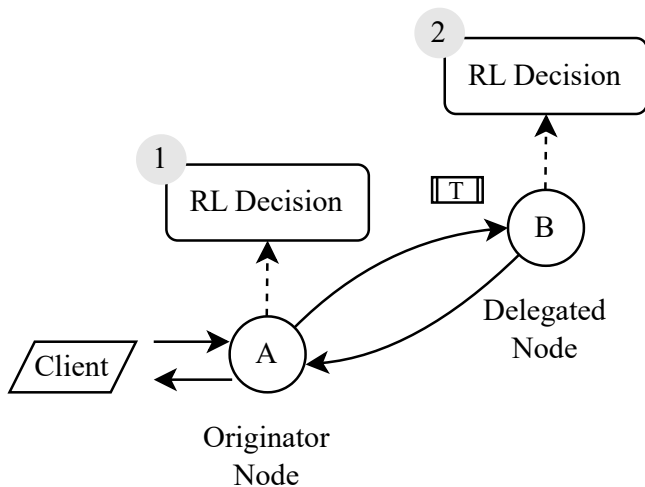
Fig. 1: The proposed double-decision scheme for cooperative and decentralized scheduling. Node A receives a task to be executed and decides whether to forward it to another node; when the task is forwarded, even Node B makes the decision whether to accept or reject it.

present the double-step RL-based algorithm for online scheduling and in Section 5 we show the experimental results of the proposed approach. Finally, we draw the conclusions in Section 6.

2   Related Work

Dynamic scheduling solutions address scheduling issues the most when the scheduler lacks precise knowledge of the jobs. Starting from the classic Job-shop scheduling problem [3], lots of techniques and algorithms were introduced to solve this problem.

Various sets of heuristic algorithms are used for the job scheduling process, popular algorithms include the genetic algorithm, ant colony optimization, bee life [18] and symbiotic organization used to optimize the scheduling process [13]. However these categories belong to static scheduling techniques which need all the details about the task to be known beforehand, and this is in general not optimal for online and real-time scheduling. In some studies multi-agent-based models were introduced to evaluate task scheduling based on a priority rule, the work introduced by Hosseinioun et al. [8] has the objective of minimizing energy consumption, as it's a major factor when working in a fog environment, and the approach was pivoted around dynamic voltage and frequency scaling (DVFS) methodology [4] which does not care much about the tasks' deadline as much as the energy consumption. Wang and Chen [22] also contributed to optimizing resource allocation but without experimenting with the model in a real-world

application. There was also a significant contribution to using neural networks with the task scheduling problem [11] [1]. Witanto et al. [24] proposed neural network based on adaptive selection of VM consolidation algorithms which selects the most adequate algorithm based on a specified priority, also CNNs started getting involved in task scheduling but with very limited use cases compared to other popular neural networks models [20]. the main characteristic of NN approaches is that it optimizes for energy consumption but it lacks speed due to the high inference latency when dealing with task deadline-critical applications.

The main focus of this work is to improve the algorithm introduced in [16], which implemented a decentralized distributed system for efficient job scheduling. It used the concept of task deadlines and online scheduling to determine which node in the cluster should execute the task. Nodes communicate in a peer-to-peer fashion and the node which executes that task gets rewarded based on whether the task is executed by the predefined deadline or not, then the client is able to train the node based on the knowledge of the node's state when the task was sent, the scheduling action taken by the node, and the reward gained based on the task deadline. Consequently, based on this reinforcement learning approach, the environment is not modeled, and the state of each node is determined online in a real deployment.

3   System Model

The system model is formulated using a Markov decision process, which does not require the knowledge of any previous states, but only the current one is meaningful. For learning the policy, since we use a model-free approach in which we do not know the probabilities $p(s', r|s, a)$, we rely on a time differential approach. Therefore the agent needs to interact directly with the environment, obtain the reward and train the model. This basic RL framework comprises the environment, the agent, and the reward, which we will now see in detail.

3.1   Environment

The system is composed of a set $\mathcal{N}$ of nodes with the same capabilities (same computing power and memory), the nodes are aware of each other's existence, the nodes are communicating only in a peer-to-peer fashion, no consensus or broadcast communication algorithms are used, all the nodes have the same configuration settings of tasks concurrent execution and queuing. Also, all the computing nodes have predefined functions to execute different types of tasks. The main purpose of this distributed system is to collaborate to execute as many tasks as possible within the deadline by executing these tasks locally or delegating them to other peers to achieve optimal performance in aggregate.

3.2   Agent

Each node is seen as an independent agent responsible for making a decision based on the current state of the node, and this is a powerful capability of the system

as the nodes have independent agents, so they communicate in a peer-to-peer fashion. The node's state is calculated when a new task arrives, and a scheduling decision must be taken. Equation 1 shows the state of node i when the k-th task arrives at the node at time t. The state S_{ik}^t is a triple composed by the type of the task k (we suppose $v_i^k \in \Upsilon$), the load of node i at time t which is l_i^t (where $l_i^t \in \mathbb{N}$ and $0 \le l_i^t \le K$) and the node to which the client sent the task at first ($n_{if}^k \in \mathcal{N}$), the originator node.

$$S_{ik}^t = \{v_i^k,\ l_i^t,\ n_{if}^k\} \tag{1}$$

The node's actions also depend on its role in the execution flow. Since we allow only one hop, if a node i receives the task k from a client (and therefore $n_{if}^k \ne i$) it can reject the request (action Re), execute it locally (action Ex), forward the request to a random node (action Fw), or to delegate the request to another peer, otherwise, if the same node i receives the task k from another peer node (suppose $j \ne i$ and n_{if}^k) then it only can execute the task locally or reject it. Equation 2 shows the formalization of the set of the actions that node i can execute on task k at time t.

$$A_{ik}^t = \begin{cases} \{Ex, Re, Fw\} \cup \mathcal{N} & \text{if } n_{if}^k \ne i \\ \{Ex, Re\} & \text{otherwise} \end{cases} \tag{2}$$

3.3   Reward

As shown in Table 1, the reward is calculated by evaluating if the task is executed within the deadline, and it's calculated on the client side to consider the network latency. This is how the node's location and network capabilities are considered in the learning process, and time is calculated on the client side, from sending the request until the response is returned. As the execution flow might involve one or two nodes based on the action of the node which receives the request from the client whether to forward the task or not, in this case, both nodes get the same reward (which is equal to one) also depending on whether the task is executed within the deadline or not. This approach makes sense as in the case of having two nodes in the task execution flow, both nodes act as a team, and they should collaborate to execute the task within the deadline, and therefore they'll get the same reward. Also, this is the best abstraction from a client's point of view as it should not know how the execution flow works and the client should only care about the task execution time. We do not choose to use a more articulated reward, since once the task is over the deadline we suppose to become useless to the client as it would be in a real-time image processing application.

4   Reinforcement Learning based Scheduling

Recalling the RL-based scheduler in [16], when the node's action is to forward the task to another node, the node receiving the task always executes the task

Originator Node Action	Task Result	Reward on Node	
		Originator	Delegated
Rejected	-	0	-
Executed Locally	Within-Deadline	1	-
	Over-Deadline	0	-
Forwarded	Within-Deadline	1	1
	Over-Deadline	0	0

Table 1: Summary of reward assignment. The reward is positive when the task is completed within the deadline and when forwarded, the reward is assigned to both nodes.

locally unless it is fully utilized so, in this case, the task is not executed. The proposed approach introduces an extra learning step to be done on the delegated node side so that instead of having only one option, which is to execute the task locally, it can now choose if it executes the task locally or rejects it. This approach enhanced the system as the delegated node could be trained in such a way that if the task is achievable within the deadline so the node executes the task, and the system works the same as before, and if the task would not be executed within the deadline so the node would reject it, and a node rejecting a task gets the same reward as executing a task which does not meet the deadline but in this case rejecting the task is better as this would save more time for the cluster to execute other tasks which would meet the deadline, and hence improve the overall system efficiency, which is maximizing the number of successfully executed tasks in a time unit. Therefore, the node's state and actions selections are based on the node's role in the task execution flow as described in equations 1 and 2

The reinforcement learning algorithm used for training the agent uses Sarsa properly adapted for continuous learning. The average reward is used as a baseline for directing the policy in such approaches. Therefore, given a node i which receives a task k and its current state S_{ik}^t, the optimal policy π selects an action A_{ik}^t, a reward r is obtained, and the long-term reward is maximized [19].

Given the state description in Equation 1, the set of states is finite and can be stored in a Q table. Therefore, the Q-table is used as a function approximator mechanism. The average-reward concept is used in Differential Semi-Gradient Sarsa Algorithm, and the table is updated according to Equation 3.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Delta_t \tag{3}$$

Where $Q(S_t, A_t)$ is the Q value of the state S_t and an action A_t, and it gets updated by using Δ_t defined in equation 4 multiplied by a hyper-parameter α as follows:

$$\Delta_t = [R_{t+1} - \bar{R}_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4}$$

Here, R_{t+1} is the immediate reward, and $\bar{R}$ is the average reward which is updated according to the following Equation 5, given β as another hyper-parameter of the algorithm.

$$\bar{R}_{t+1} = \bar{R}_t + \beta \Delta_t \tag{5}$$

In the proposed double decision scheduler, the execution flow starts when the client sends a request to execute a task to a node in the cluster and records the task's start timestamp, then the node executes Algorithm 1, which calculates the current node state according to the node's current load and the type of the task, it also stamp the request with a generated request number, which is enumerated in an ascending order, then the node sends the state to the learner for inference, it receives an action which could be one of the following:

- deliberately rejecting the request;
- executing the task locally;
- picking random node, and if this node is less loaded than the current node, the random node executes the task. Otherwise, the task is executed locally by the current node;
- the task is forwarded to a random node in the nodes' list, including the current node itself.

If the action is to send the request to another node, then the other node (or, as we call it, the delegated node) does the following:

- calculates the current state with respect to the following:
  - node's current load;
  - the type of the task;
  - the sender node id;
- It stamps the request with a generated request number, which is enumerated in an ascending order;
- It sends the state to the learner for inference;
- It receives an action that could be either deliberately rejecting the request or executing the task locally;
- If the latter is the case, then it executes the task;
- it sends back its state, which action is executed, and the task execution result if the chosen action was to execute the task;

Finally, the peer node returns the response to the client with all the learning information.

As mentioned before, each node could act as an originator node (receiving the request from the client) or a delegated node (receiving the request from another peer) based on its role in the execution flow, so the algorithm should be compact to accommodate for both roles, and this is done by using the $requestIsExternal$ Boolean, if it's true so the node acts as a delegated node, otherwise, it acts as the originator node.

Algorithm 1 Double Learner Scheduler

Require: Node, Task, A, qTable, requestIsExternal, SenderNodeId
  if requestIsExternal **then** (Get State for Delegated Node)
     $s \leftarrow$ aggregate(Node.getLoad(), Task.getType(), SenderNodeId)
  else
     $s \leftarrow$ aggregate(Node.getLoad(), Task.getType())
  end if
  $a \leftarrow$ max(qTable.getActionsList(s)) with prob. 1-e otherwise random(A)
  if $a == 0$ **then**
    Node.reject(Task)
  else if $a == 1$ **or** requestIsExternal **then**
    Node.execute(Task)
  else if $a == 2$ **then** (Probe and Forward)
    RandomNode $\leftarrow$ pickRandom(Node.getNeighbors())
    if RandomNode.getLoad() $<$ Node.getLoad() **then**
      forwardTo(RandomNode, Task)
    else
      Node.execute(Task)
    end if
  else
    Node $\leftarrow$ pickNode(a)
    forwardTo(Node, Task)
  end if

---

5   Experimental Results

The proposed algorithm has been implemented into the P2PFaas framework [17]. In particular, the modules that have been improved are the learner service which implements the RL model and is also responsible for training the model and the decision-making process, and the scheduler service, which implements the actual scheduler algorithm. In this architecture, the client is not only responsible for sending task requests to the cluster but also for parsing the learning entries from the node's returned state and action, calculating the reward, and training the nodes. The client sends the learning parameters to every node in batches that contain, in order: the request's id, state, action, and reward. The node's learner service receives the batch, sorts it based on the request's id, and starts the learning process using the current average reward method.

    Two studies have been performed comparing a single decision scheduler (where the decision is made only in the originator node) and our proposed double decision scheduler regarding performance. The first study was done by applying static loads (requests/second) but not balanced over the nodes (heterogeneous loads), and the second experiment is done using data extracted from the open data for New York City as done in the previous work as well [16]. The performance metric will always be the same: the number of completed tasks within the deadline per second, also called "within-deadline rate".

The cluster consists of 11 Raspberry Pis 4 Model B, five of which are 8GB of RAM, and the rest are 4GB of RAM. For each Raspberry, we installed a face recognition FaaS (Function-as-a-Service) triggered upon the HTTP call with an image as payload. The experiment is done using a script that generates 11 concurrent flows of requests to all the nodes in the cluster at different rates (requests per second), and it uses two payloads as in Table 2.

	Image A	Image B
Resolution	320×210	180×118
Size (kB)	28.3	23.8
Processing Time (ms)	188.25	74.95

Table 2: Benchmark payload images used in the tests

To calculate the reward, the client has to set a deadline for the nodes executing the tasks, calculates the node's task execution time, and compare these values, and consequently, the reward is assigned as described in Section 3.3. The assigned deadlines setting are the Processing Time of the image multiplied by a factor of 1.1, and it sends requests for Image A and Image B with a ratio of 1:1. The benchmark time is set to 1800 seconds, and the nodes are loaded heterogeneously, so every node is experiencing fixed traffic but loaded differently from the other nodes, and this traffic is distributed with the following values (in requests per second): 4, 6, 8, 12, 13, 14, 15, 16, 17, 18, 19. The "Dynamic Study" traffic is extracted from open data of New York City data set, which estimates the average taxi traffic across several locations in the city. The traffic data covers only six nodes, so it is repeated for the other five nodes. As the traffic is normalized, it is scaled by a range from 0 to 20, so, for example, when the traffic data at a point in time is 0.9, the actual load on the node is 0.9 * 20 = 18.

5.1   The Within-Deadline Rate Comparison

In Figures 2 and Figure 3, the comparison shows that the double decision scheduler (called "Double Learner" in charts) is at least as good as the single decision scheduler (called "Single Learner" in charts) on average for both the static and the dynamic traffic.

In Figure 2, the single decision scheduler was better than the double decision one at the first 200 seconds for most of the nodes before the double decision taking over for the rest of the experiment. This behavior is expected as the double decision needs some time for delegated node's learner to take effect, starting from time 200 seconds and upwards. The double decision scheduler within-deadline rate was noticeably higher than the single decision for some nodes, the double decision within-deadline rate is better than the single decision by 3% on average,

and it has a within-deadline rate of 90% on average. For the dynamic study
shown in Figure 3, the results are the same as the static load study as the double
decision scheduler reacts the same for dynamic loads as the static loads, the
double decision is constantly performing over the single decision, for nodes 1, 2,
and 9, the performance is almost the same, but for the whole system on average,
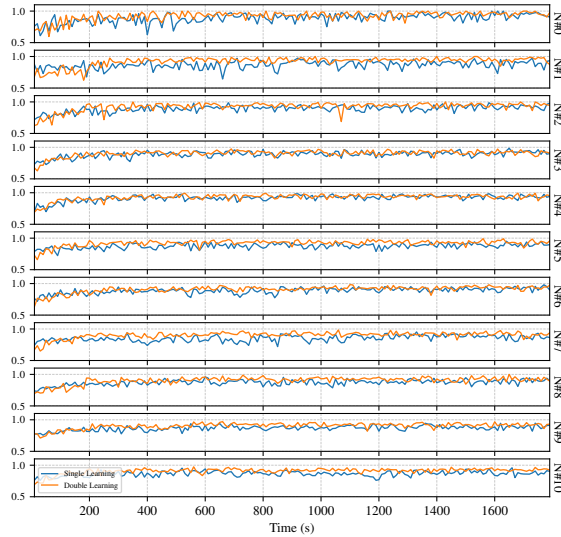the double decision rate was better by 4.1%

Fig. 2: The within-deadline rate for the single decision vs double decision when
applied static loads in Raspberry Pis cluster, the double decision was better than
the single decision by 3% on average

However, it is also noticeable that when the loads increase, the double deci-
sion degrades to the performance of the single decision, as nodes 8, 9, and 10
performances were lower than nodes 5, 6, and 7 by 0.3%, and this is because the
double decision takes more time to execute the task in the general case, as it
involves extra inference step on the delegated node side. This latency takes effect
when the traffic increases and the double decision degrades.

5.2   Delegated Node Performance

A separate monitor has been applied to the decision behavior of all nodes but only
acting in the second node role during the task execution process. It was necessary
to be sure that the delegated node's decision was getting better over time and it
is choosing a decision that maximized the number of successfully executed tasks
per second. Figure 4 describes the node's execution in the "delegated node role"
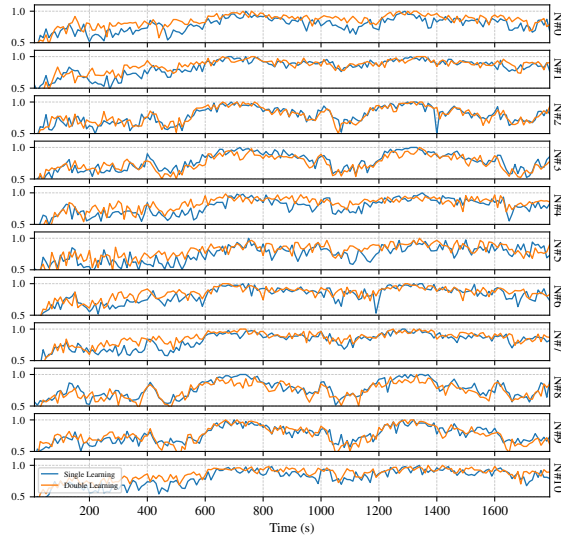
Fig. 3: The within-deadline rate for the single decision vs. double decision when applied dynamic loads in Raspberry Pis cluster, the double decision was better than the single decision by 4.1% on average
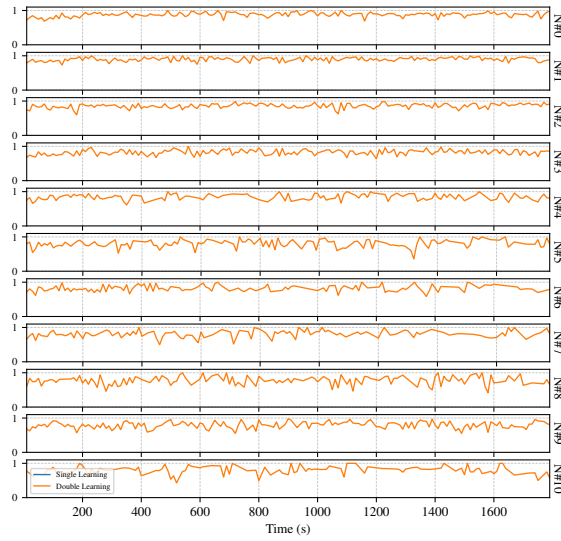

Fig. 4: The within-deadline rate for nodes Raspberry Pis cluster with respect to static loads when they act as delegated nodes as double learning is activated

and how it behaves over time. It shows that the learning process starts low at first and then it saturates by the time, and the average within-deadline rate for nodes with lower loads for example, nodes from 0 to 4 is 81% which is by a small amount better than the ones with nodes from 5 to 10 with 80%, so that the within-deadline rate for the nodes with lower loads is 1% higher than the ones with high loads (from node 5 to node 10), and this behavior is expected as the more loads applied to the node, the less it achieves task within the deadline. It is important to note that the more loads the node experiences, the more stochastic the delegated node performance will be.
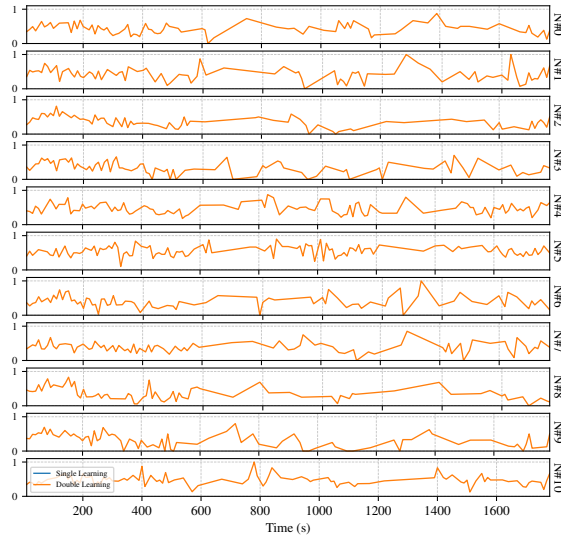

Fig. 5: The within-deadline rate for nodes in Raspberry Pis cluster with respect to dynamic loads when they act as delegated nodes as double learning is activated

It is shown in Figure 4 for the nodes which experience high loads (from node 5 to node 10), and it is also shown in Figure 5 as the more the loads are dynamic, the more spikes and oscillations the delegated node experiences. The reason for the within-deadline rate instability shown in Figure 5 is that the delegated node is always trying to follow the dynamic loads by adapting it is policy. This is normal, as shown in Figure 3. Figure 5 also shows that the delegated node within-deadline rate is resilient to load changes for most of the nodes, as the node is always trying not to fall below minimum effectiveness because of the varying loads, keeping the average rate for all nodes equals to 1.06 requests/seconds.

6   Conclusions

This paper proposes a double-decision online distributed scheduler using reinforcement learning and improves an online learner scheduler [16] with a second decision if the node's action is to delegate the task to another node so that now the second node learns to choose the correct action between accepting or rejecting the task based on its state and the originator node. The delegated node learns to reject a task if it would not be executed within the deadline and therefore saving time for executing other tasks which potentially can be completed within the deadline. By performing different experiments implementing the approach in a cluster of 11 Raspberry Pi we show that our approach increases the in-deadline rate showing an improvement over the single decision scheduler performances. However, different aspects could have a more detailed analysis. For example, it could be further studied the energy aspect of the nodes which may make a decision also based on the power consumption.

References

1. Ali, A.M., Tirel, L.: Action masked deep reinforcement learning for controlling industrial assembly lines. In: 2023 IEEE World AI IoT Congress (AIIoT). pp. 0797–0803 (2023). https://doi.org/10.1109/AIIoT58121.2023.10174426
2. AlOrbani, A., Bauer, M.: Load balancing and resource allocation in smart cities using reinforcement learning. In: 2021 IEEE International Smart Cities Conference (ISC2). pp. 1–7. IEEE (2021)
3. Aydin, M.E., Öztemel, E.: Dynamic job-shop scheduling using reinforcement learning agents. Robotics and Autonomous Systems **33**(2-3), 169–178 (2000)
4. Bansal, S., Kumar, P., Singh, K.: Duplication-based scheduling algorithm for interconnection-constrained distributed memory machines. In: High Performance Computing—HiPC 2002: 9th International Conference Bangalore, India, December 18–21, 2002 Proceedings 9. pp. 52–62. Springer (2002)
5. Barthélemy, J., Verstaevel, N., Forehead, H., Perez, P.: Edge-computing video analytics for real-time traffic monitoring in a smart city. Sensors **19**(9), 2048 (2019)
6. Bonomi, F., Milito, R., Natarajan, P., Zhu, J.: Fog computing: A platform for internet of things and analytics. In: Big data and internet of things: A roadmap for smart environments, pp. 169–186. Springer (2014)
7. Broucke, S.V., Deligiannis, N.: Visualization of real-time heterogeneous smart city data using virtual reality. In: 2019 IEEE International Smart Cities Conference (ISC2). pp. 685–690. IEEE (2019)
8. Hosseinioun, P., Kheirabadi, M., Tabbakh, S.R.K., Ghaemi, R.: A new energy-aware tasks scheduling approach in fog computing using hybrid meta-heuristic algorithm. Journal of Parallel and Distributed Computing **143**, 88–96 (2020)
9. Houidi, O., Zeghlache, D., Perrier, V., Quang, P.T.A., Huin, N., Leguay, J., Medagliani, P.: Constrained deep reinforcement learning for smart load balancing. In: 2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC). pp. 207–215. IEEE (2022)
10. Hu, F., Deng, Y., Saad, W., Bennis, M., Aghvami, A.H.: Cellular-connected wireless virtual reality: Requirements, challenges, and solutions. IEEE Communications Magazine **58**(5), 105–111 (2020)

11. Iftikhar, S., Ahmad, M.M.M., Tuli, S., Chowdhury, D., Xu, M., Gill, S.S., Uhlig, S.: Hunterplus: Ai based energy-efficient task scheduling for cloud–fog computing environments. Internet of Things **21**, 100667 (2023). https://doi.org/https://doi.org/10.1016/j.iot.2022.100667, `https://www.sciencedirect.com/science/article/pii/S2542660522001482`

12. Iorga, M., Feldman, L., Barton, R., Martin, M.J., Goren, N.S., Mahmoudi, C.: Fog computing conceptual model (2018)

13. Kaur, N., Bansal, S., Bansal, R.K.: Survey on energy efficient scheduling techniques on cloud computing. Multiagent and Grid Systems **17**(4), 351–366 (2021)

14. Liu, Y., Fieldsend, J.E., Min, G.: A framework of fog computing: Architecture, challenges, and optimization. IEEE Access **5**, 25445–25454 (2017)

15. Orhean, A.I., Pop, F., Raicu, I.: New scheduling approach using reinforcement learning for heterogeneous distributed systems. Journal of Parallel and Distributed Computing **117**, 292–302 (2018)

16. Proietti Mattia, G., Beraldi, R.: On real-time scheduling in fog computing: A reinforcement learning algorithm with application to smart cities. In: 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). pp. 187–193 (2022). https://doi.org/10.1109/PerComWorkshops53856.2022.9767498

17. Proietti Mattia, G., Beraldi, R.: P2pfaas: A framework for faas peer-to-peer scheduling and load balancing in fog and edge computing. SoftwareX **21**, 101290 (2023). https://doi.org/https://doi.org/10.1016/j.softx.2022.101290, `https://www.sciencedirect.com/science/article/pii/S2352711022002084`

18. Sehgal, N., Bansal, S., Bansal, R.: Task scheduling in fog computing environment: An overview. International Journal of Engineering Technology and Management Sciences **7**(1), 47–54 (2023)

19. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)

20. Talaat, F.M., Ali, H.A., Saraya, M.S., Saleh, A.I.: Effective scheduling algorithm for load balancing in fog environment using cnn and mpso. Knowledge and Information Systems **64**(3), 773–797 (2022)

21. Wang, J., Zhao, L., Liu, J., Kato, N.: Smart resource allocation for mobile edge computing: A deep reinforcement learning approach. IEEE Transactions on emerging topics in computing **9**(3), 1529–1541 (2019)

22. Wang, Q., Chen, S.: Latency-minimum offloading decision and resource allocation for fog-enabled internet of things networks. Transactions on Emerging Telecommunications Technologies **31**(12), e3880 (2020)

23. Wang, S., Guo, Y., Zhang, N., Yang, P., Zhou, A., Shen, X.: Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. IEEE Transactions on Mobile Computing **20**(3), 939–951 (2019)

24. Witanto, J.N., Lim, H., Atiquzzaman, M.: Adaptive selection of dynamic vm consolidation algorithm using neural network for cloud resource management. Future generation computer systems **87**, 35–42 (2018)