A Distributed Deep Reinforcement Learning Technique for Application Placement in Edge and Fog Computing Environments

Mohammad Goudarzi, *Member, IEEE*, Marimuthu Palaniswami, *Fellow, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

Abstract—Fog/Edge computing is a novel computing paradigm supporting resource-constrained Internet of Things (IoT) devices by placement of their tasks on edge and/or cloud servers. Recently, several Deep Reinforcement Learning (DRL)-based placement techniques have been proposed in fog/edge computing environments, which are only suitable for centralized setups. The training of well-performed DRL agents requires manifold training data while obtaining training data is costly. Hence, these centralized DRL-based techniques lack generalizability and quick adaptability, thus failing to efficiently tackle application placement problems. Moreover, many IoT applications are modeled as Directed Acyclic Graphs (DAGs) with diverse topologies. Satisfying dependencies of DAG-based IoT applications incur additional constraints and increase the complexity of placement problem. To overcome these challenges, we propose an actor-critic-based distributed application placement technique, working based on the IMPortance weighted Actor-Learner Architectures (IMPALA). IMPALA is known for efficient distributed experience trajectory generation that significantly reduces exploration costs of agents. Besides, it uses an adaptive off-policy correction method for faster convergence to optimal solutions. Our technique uses recurrent layers to capture temporal behaviors of input data and a replay buffer to improve the sample efficiency. The performance results, obtained from simulation and testbed experiments, demonstrate that our technique significantly improves execution cost of IoT applications up to 30% compared to its counterparts.

Index Terms—Fog computing, edge computing, deep reinforcement learning, application placement, Internet of Things (IoT)

1 Introduction

In recent years, new computing and communication technologies have rapidly advanced, leading to the proliferation of smart Internet of Things (IoT) devices (e.g., sensors, smartphones, cameras, vehicles) [1]. These advancements empower IoT devices to run a multitude of resource-hungry and latency-sensitive IoT applications. These emerging IoT applications increasingly demand computing, storage, and communication resources for the execution [2]. Also, the execution of such resource-hungry applications requires a significant amount of energy consumption. Hence, limited computing, storage, and battery capacity of IoT devices, directly affect the performance of IoT applications and user experience [3].

The Cloud computing paradigm, as a centralized solution, is one of the main enablers of the IoT, providing unlimited and elastic remote computing and storage resources for the execution of computation-intensive IoT applications [4]. All/some computation-intensive constituent parts (e.g., service, modules, tasks) of IoT applications can be placed (i.e., offloaded) on remote Cloud Servers (CSs) for execution and storage in order to reduce the execution time of IoT applications and energy consumption of IoT devices [5], [6]. However, due to low bandwidth and high communication latency between IoT devices and CSs, the requirements of latency-sensitive IoT applications cannot be efficiently satisfied [7]. Besides, low bandwidth and high latency of CSs may incur more energy consumption for IoT devices due to higher active communication time with CSs. To improve the high communication latency and low bandwidth of CSs, fog computing paradigm, as a distributed solution, has emerged. In fog computing, heterogeneous Fog Servers (FSs) are distributed in the proximity of IoT devices, through which IoT devices can access the computing and storage resources with higher bandwidth and less communication latency, compared to CSs [8]. However, these FSs usually have limited resources (e.g., CPU, RAM) in comparison to CSs. In our view, edge computing harnesses only distributed edge resources at the proximity of IoT devices while fog computing harnesses both edge and cloud resources to address the requirements of both computation-intensive and latency-sensitive IoT applications (although some works use these terms interchangeably).

In real-world scenarios, many IoT applications (e.g., face recognition [9], smart healthcare [10], and augmented reality [11]) are modeled as a Directed Acyclic Graph (DAG), in

- *Mohammad Goudarzi and Rajkumar Buyya are with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Melbourne, VIC 3010, Australia. E-mail: mgoudarzi@student.unimelb.edu.au, rbuyya@unimelb.edu.au.*
- *Marimuthu Palaniswami is with the Department of Electrical and Electronic Engineering, The University of Melbourne, Melbourne, VIC 3010, Australia. E-mail: palani@unimelb.edu.au.*

Manuscript received 16 Mar. 2021; revised 18 Oct. 2021; accepted 22 Oct. 2021. Date of publication 27 Oct. 2021; date of current version 4 Apr. 2023. (Corresponding author: Mohammad Goudarzi.) Digital Object Identifier no. 10.1109/TMC.2021.3123165

which nodes and edges represent tasks and data communication among dependent tasks, respectively. These DAG-based IoT applications incur higher complexity and constraints when making placement decisions for the execution of IoT applications. Hence, placement/offloading of IoT applications, comprised of dependent tasks, on/to suitable servers with the minimum execution time and energy consumption is an important and yet challenging problem in fog computing. Many heuristics, approximation, and rule-based solutions are proposed for this NP-hard problem [12], [13], [14]. Although these techniques work well in general cases, they heavily rely on comprehensive knowledge about the IoT applications and resource providers (e.g., CSs or FSs). The fog computing environment is stochastic in several aspects, such as arrival rate of application placement requests, dependency among tasks, number of tasks per IoT application, resource requirements of applications, and available remote resources, just to mention a few. Therefore, heuristic-based techniques cannot efficiently adapt to constant changes in the fog computing environments [15].

Deep Reinforcement Learning (DRL) provides a promising solution by combining Reinforcement Learning (RL) with Deep Neural Network (DNN). Since DRL agents can accurately learn the optimal policy and long-term rewards without prior knowledge of the system [16], they help solve complex problems in dynamic and stochastic environments such as fog computing, especially when the state space is so large [15], [17]. Although the effectiveness of DRL techniques is shown in several works [18], [19], [20], [21], [22], there are yet several challenges for practical realizations of these techniques in fog computing environments. In DRL, the agent interacts with the environment using trial and error (i.e., exploration) and records the trajectories of experiences (i.e., sequences of states, actions, and rewards) in large quantities with high diversity. These experience trajectories are used to learn the optimal policy in the training phase. In complex environments, such as fog computing, DRL agents require a large number of interactions with the environment to obtain sufficient trajectories of experience to capture the properties of the environment. Therefore, the exploration cost of agents increases. Obviously, it negatively affects the user experience in the fog computing environment, because the training of the DRL agents in such complex environments is a time-consuming process. The centralized DRL agents used in fog computing environments are not suitable for the highly distributed and stochastic environments [23]. Hence, a key problem is how to adapt distributed DRL techniques to efficiently perform in fog computing environments. Considering the distributed nature of fog computing environments, the application placement engines can be placed on different FSs, that work in parallel and efficiently produce diverse experience trajectories with less exploration costs. However, other challenges may arise such as how these trajectories can be efficiently and practically used to learn the optimal policy.

To address the aforementioned challenges, we propose an EXperience-sharing Distributed Deep Reinforcement Learning-based application placement technique, called X-DDRL, to efficiently capture complex dynamics of DAG-based IoT applications and FSs' resources. The X-DDRL uses IMPortance weighted Actor-Learner Architectures (IMPALA), proposed by Espeholt *et al.* [24], which is a distributed DRL agent that uses an actor-learner framework to learn the optimal policy. In IMPALA, several actors interact with the environments in parallel and produce diverse experience trajectories in a timely manner. Then, these experience trajectories are periodically forwarded to the learner for the training and learning of the optimal policy. After each policy update of the learner, actors reset their parameters with the learner's one and independently continue their explorations. As a result of this distributed and collaborative experience-sharing between actors and learners, the exploration costs reduce significantly, and the experience trajectories are efficiently reused. However, due to decoupled acting and learning, a policy gap between actors and learners arises, which can be corrected by V-trace off-policy correction method [24]. Moreover, we use Recurrent Neural Networks (RNN) to accurately identify the temporal patterns across different features of the input. Finally, the X-DDRL uses experience replay to break the strong correlation between generated experience trajectories and improve sample efficiency.

The main contributions of this paper are summarized as follows:

- A weighted cost model for application placement of DAG-based IoT applications is proposed to minimize the execution time of IoT applications and energy consumption of IoT devices. Then, this weighted cost model is adapted to be used in DRL-based techniques.
- A pre-scheduling technique is put forward to define an execution order for dependent tasks within each DAG-based IoT application.
- We propose a dynamic and distributed DRL-based application placement technique for complex and stochastic fog computing environments, working based on the IMPALA framework. Our technique uses RNN to capture complex patterns across different features of the input. Moreover, it uses an experience replay buffer which remarkably helps sampling efficiency and breaks the strong correlation between experience trajectories.
- We conduct simulation and testbed experiments using a wide range of synthetic DAGs, derived from the real-world IoT applications, to cover diverse application dependency models, task numbers, and execution costs. Also, the performance of our technique is compared with two state-of-the-art DRL techniques, called Double Deep Q Learning (Double-DQN), and Proximal Policy Optimization (PPO), and a greedy-based heuristic.

The rest of the paper is organized as follows. Relevant DRL-based application placement techniques in edge and fog computing environments are discussed in Section 2. The system model and problem formulations are presented in Section 3. Section 4 describes the DRL-based model and its main concepts. Section 5 presents our proposed distributed DRL-based application placement framework. We evaluate the performance of our technique and compare it with state-of-the-art techniques in Section 6. Finally, Section 7 concludes the paper and draws future works.

2 Related Work

Considering the large number of works in application placement techniques, in this section, related works for DRL-based application placement techniques in fog/edge computing environments are studied. However, detailed related works for the non-learning-based application placement techniques and frameworks are available in [3], [25], [26].

DRL-based works are first divided into edge computing and fog computing. Edge computing works only consider the resources in the proximity of IoT users while fog computing ones take advantage of both edge resources and remote cloud resources. Hence, the heterogeneity of resources is higher in the fog computing works, which leads to higher complexity for DRL-based application placement techniques to identify the features of the environments. Besides, works are further categorized into independent and dependent categories based on the dependency model of their IoT applications' granularity (e.g., tasks, modules). In IoT applications with dependent tasks (i.e., DAGs), each task can be executed only when its parent tasks finish their execution, while tasks of independent IoT applications do not have such constraints for execution. Therefore, works in the dependent category have more constraints, and hence the DRL agent requires specific considerations compared to works in the independent category to efficiently learn the optimal policy.

2.1 Edge Computing

In the independent category, Huang *et al.* [27] proposed a DRL-based offloading algorithm to minimize the system cost, in which parallel computing is used to speed up the computation of a single edge server. Min *et al.* [28] proposed a fast deep Q-network (DQN) based offloading scheme, combining the deep learning and hotbooting techniques to improve the learning speed of Q-learning. Huang *et al.* [18] proposed a quantized-based DRL method to optimize the system energy consumption for faster processing of IoT devices' requests. Chen *et al.* [19] proposed a double DQN-based algorithm to minimize the energy consumption and execution time of independent tasks of IoT applications. Huang *et al.* [20] also proposed a DRL-based offloading framework based on DQN that jointly considers offloading decisions and resource allocations. Chen *et al.* [29] proposed a joint offloading framework with DRL to make an offloading decision based on the information of applications' tasks and network conditions where the training data is generated from the searching process of the Monte Carlo tree search algorithm. Lu *et al.* [21] proposed a Deep Deterministic Policy Gradients (DDPG)-based algorithm for computation offloading of multiple IoT users to a single edge server to improve the quality of experience of users. To improve the convergence of the DQN algorithm in an edge computing environment, Xiong *et al.* [30] proposed a DQN-based algorithm combined with multiple replay memories to minimize the execution time of one IoT application. Qiu *et al.* [31] studied the distributed DRL in an edge computing environment with a single edge server to minimize the energy cost of running IoT applications, consisted of independent tasks. To obtain this goal, they combined deep neuro-evolution and policy gradient to improve the convergence results.

In the dependent category, Wang *et al.* [16] proposed a meta reinforcement learning algorithm based on the Proximal Policy Optimization (PPO). The main goal of this work is to minimize the execution time of dependent IoT applications, situated in the proximity of a single edge server.

2.2 Fog Computing

In the independent category, Gazori *et al.* [32] targeted task scheduling of independent IoT applications to minimize long-term service delay and system cost. To obtain this, they used a double DQN-based scheduling algorithm combined with an experience replay buffer. Tuli *et al.* [23] proposed Asynchronous-Advantage-Actor-Critic (A3C) learning-based technique combined with Recurrent Neural Network (RNN) for the scheduling of independent IoT applications to minimize total system cost.

In the dependent category, Lu *et al.* [33] proposed a DQN-based algorithm to minimize the overall system cost. Although they consider dependencies among constituent parts of each IoT application, they only consider the sequential dependency model among tasks of an IoT application, where there are no tasks for parallel execution.

2.3 A Qualitative Comparison

Table 1 identifies and compares the main elements of related works with ours in terms of their IoT application, architectural, and application placement engine properties. In the IoT application section, the dependency model of each proposal is studied, which can be either independent or dependent. Moreover, we study how each proposal models IoT applications in terms of the number of tasks and heterogeneity. This demonstrates whether IoT applications consist of homogeneous or heterogeneous tasks in terms of their computation and data flow. In the architectural properties, the attributes of IoT devices, fog/edge servers, and cloud servers are studied. For IoT devices, the overall number of devices and their type of requests are identified. The heterogeneous request type shows that each device has a different number of requests with various requirements compared to other IoT devices. For edge/fog servers, the number of deployed servers between IoT devices and cloud servers and the heterogeneity of their resources are studied. Moreover, the multi-cloud shows either these works consider different cloud service providers with heterogeneous resources or not. In the application placement engine, the main employed DRL methods are identified. Besides, it is studied either these works consider any mechanism to provide priority for the execution of tasks or not. Finally, the decision parameters of these DRL-based techniques are identified.

Considering DRL-based application placement techniques in edge and fog computing and their identified properties, the environment with multiple heterogeneous IoT devices, heterogeneous FSs, and heterogeneous multi CSs has the highest number of features. Moreover, DAG-based IoT applications incur more constraints on DRL agents as they need to consider the dependency among tasks within each IoT application. The exploration cost of DRL agents

TABLE 1
A Qualitative Comparison of Related Works With Ours

Techniques	Category	Application Properties			Architectural Properties					Application Placement Engine Properties				
					IoT Device Layer		Edge/Fog Layer					Decision Parameters		
		Dependency	Task Number	Heterogeneity	Number	Request Type	Number	Heterogeneity	Multi Cloud	Main Method	Task Priority	Time	Energy	Weighted
[27]	Edge Computing	Independent	Multiple	Heterogeneous	Multiple	Homogeneous	Single	Homogeneous	×	_	×	×	×	✓
[28]			Single	Homogeneous	Multiple	Heterogeneous	Single	Homogeneous	×	_	×	×	✓	×
[18]			Single	Homogeneous	Single	Homogeneous	Multiple	Heterogeneous	×	DQN	×	×	✓	×
[19]			Single	Heterogeneous	Multiple	Heterogeneous	Single	Homogeneous	×	Double DQN	×	✓	✓	×
[20]			Multiple	Heterogeneous	Multiple	Heterogeneous	Single	Homogeneous	×	DQN	×	✓	✓	✓
[29]			Single	Heterogeneous	Multiple	Heterogeneous	Multiple	Heterogeneous	×	MCTS	×	✓	✓	×
[21]			Multiple	Heterogeneous	Multiple	Heterogeneous	Single	Homogeneous	×	DDPG	×	✓	✓	✓
[30]			Multiple	Heterogeneous	Multiple	Homogeneous	Single	Homogeneous	×	DQN	×	✓	×	×
[31]			Multiple	Heterogeneous	Multiple	Heterogeneous	Single	Homogeneous	×	Deep Neuroevolution	×	×	✓	×
[16]		Dependent	Multiple	Heterogeneous	Multiple	Heterogeneous	Single	Homogeneous	×	PPO	✓	✓	×	×
[32]	Fog Computing	Independent	Multiple	Heterogeneous	Multiple	Heterogeneous	Multiple	Heterogeneous	×	Double DQN	×	✓	×	×
[23]			Multiple	Heterogeneous	Multiple	Heterogeneous	Multiple	Heterogeneous	×	A3C	×	✓	✓	✓
[33]		Dependent	Multiple	Heterogeneous	Multiple	Homogeneous	Multiple	Heterogeneous	×	DQN	×	×	✓	×
X-DDRL			Multiple	Heterogeneous	Multiple	Heterogeneous	Multiple	Heterogeneous	✓	IMPALA	✓	✓	✓	✓

increases as the number of features and complexity of the environment increases. It negatively affects the training and convergence time of DRL techniques, and accordingly users' experience. To address these issues, we propose a distributed DRL technique based on the IMPALA architecture, called X-DDRL, in which several actors independently interact with fog computing environments and create experience trajectories in parallel. Then, these distributed experience trajectories are forwarded to the learner for training and policy updates. This significantly reduces the exploration and training costs of centralized DRL techniques. Furthermore, since the learner directly uses the batches of experience trajectories of distributed actors, rather than gradients with respect to the parameters of the policy (similar to how the A3C algorithm works), it can more efficiently learn and identify the features of input data [24]. Also, the transmission of gradients among actors and learners is more expensive in terms of data exchange size and time (similar to how A3C works) in comparison to sharing trajectories of experience. Hence, experience-sharing DRL techniques such as IMPALA are more practical and data-efficient in highly distributed and stochastic environments [24], such as fog computing. Since the policy used to generate the trajectories of experiences in distributed actors can lag behind the policy of the learner in the time of gradient calculations, a V-trace off-policy actor-critic algorithm is used to correct this discrepancy. Besides, to capture the temporal behavior of input data, we embed RNN layers in the network of actors and learners. Moreover, X-DDRL uses a replay buffer to improve the sample efficiency for training.

3 System Model and Problem Formulation

Fig. 1 represents an overview of our system model in fog computing. IoT devices send their application placement requests to brokers, situated at the edge of the network to be accessed with less latency and higher bandwidth [3],

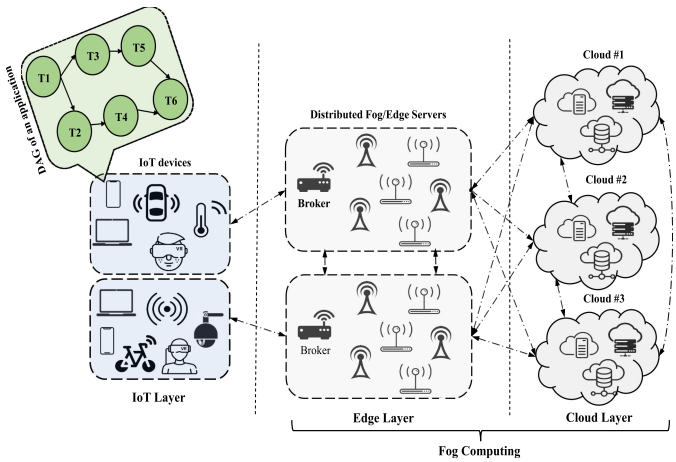


Fig. 1. An overview of our system model.

[34]. For each arriving application request, the broker makes a placement decision based on the corresponding DAG of the IoT application, its constraints, and the system status. Accordingly, each task of an IoT application may be assigned to the IoT device for the local execution or one of heterogeneous FSs or CSs for the execution.

3.1 IoT Application

Each IoT applications is modeled as a DAG $G = (\mathcal{V}, \mathcal{E})$ of its tasks, where $\mathcal{V} = \{v_i | 1 \leq i \leq |\mathcal{V}|\}, |\mathcal{V}| = L$ depicts vertex set of one application, in which v_i denotes the ith task. Moreover, $\mathcal{E} = \{e_{i,j} | v_i, v_j \in \mathcal{V}, i \neq j\}$ represents edge set, in which $e_{i,j}$ denotes there is a data flow between v_i (i.e., parent), v_j (i.e., child) and hence, v_j cannot be executed before v_i. Accordingly, for each task v_j, a predecessor task set $\mathcal{P}(v_j)$ is defined, containing all tasks that should be executed before v_j. Moreover, for each DAG G, exit tasks are referred to tasks without any children.

The amount of CPU cycles, required for the processing of each task, is represented as v_j^w, while the required amount of RAM for processing of each task is v_j^{ram}. Moreover, the weight on each edge $e_{i,j}^w$ illustrates the amount of data that task v_i sends as its output to task v_j as its input.

3.2 Problem Formulation

Each task of an IoT application can either be executed locally on the IoT device or on one of the FSs or CSs. We define the set of all available servers as $\mathcal{M}$ where $|\mathcal{M}| = M$. Each server is represented as $m^{y,z} \in \mathcal{M}$ where y shows the type of server (IoT device ($y = 0$), FSs ($y = 1$), CSs ($y = 2$)) and z denotes the server index. Therefore, the placement configuration of task v_j, belonging to an IoT application, can be defined as:

$$x_{v_j} = m^{y,z} \tag{1}$$

and accordingly, the placement configuration of an IoT application $\mathcal{X}$ is defined as the set of placement configurations for all of its tasks:

$$\mathcal{X} = \{x_{v_j} | v_j \in \mathcal{V}, 1 \leq j \leq |\mathcal{V}|\}. \tag{2}$$

We consider that tasks of an IoT application are sorted in a sequence so that all parent tasks are scheduled for the execution before their children. Hence, the dependencies among tasks are satisfied. Besides, among tasks that can be executed in parallel (i.e., tasks that all of their dependencies are satisfied), the $CP(v_i)$ is an indicator function to demonstrate whether the task is on the critical path of the IoT application or not [35] (i.e., a path containing vertices and edges that incurs the highest execution cost).

3.2.1 Execution Time Model

The execution time of each task v_j depends on the availability time of required input data for that task $\psi_{x_{v_j}}^{input}$ and its processing time on the assigned server $\psi_{x_{v_j}}^{proc}$:

$$\psi_{x_{v_j}} = \psi_{x_{v_j}}^{proc} + \psi_{x_{v_j}}^{input} \tag{3}$$

where $\psi_{x_{v_j}}^{proc}$ depends on the required CPU cycles for that task v_j^w and the processing speed of the corresponding assigned server $f_{x_{v_j}}^s$, calculated as follows:

$$\psi_{x_{v_j}}^{proc} = \frac{v_j^w}{f_{x_{v_j}}^s} \tag{4}$$

The $\psi_{x_{v_j}}^{input}$ is calculated as the maximum time that the required input data for the execution of task v_j become available on the corresponding assigned server (i.e., x_{v_j}) from its parent tasks:

$$\psi_{x_{v_j}}^{input} = \max\left(\left(\frac{e_{i,j}^w}{b_{x_{v_i},x_{v_j}}} + l_{x_{v_i},x_{v_j}}\right) \times SS(x_{v_i}, x_{v_j})\right), \quad \forall v_i \in \mathcal{P}(v_j) \tag{5}$$

where $b_{x_{v_i},x_{v_j}}$ shows the current bandwidth (i.e., data rate) between the servers to which v_i and v_j are assigned, respectively. Moreover, $l_{x_{v_i},x_{v_j}}$ demonstrates the communication latency between two servers. The $SS(x_{v_i}, x_{v_i})$ is equal to 0 if $x_{v_i} = x_{v_j}$ (i.e., same assigned servers) or 1, otherwise. Since fog computing environments are heterogeneous and stochastic, the $f_{x_{v_j}}^s$, $b_{x_{v_i},x_{v_j}}$, and $l_{x_{v_i},x_{v_j}}$ may be different among IoT devices, FSs, and CSs.

The main goal of the execution time model is to find the best-possible placement configuration for the IoT application so that its execution time becomes minimized. Assuming an IoT application consists of L tasks, the execution time model is defined as:

$$\Psi(\mathcal{X}) = \min\left(\sum_{j=1}^{L} CP(v_j) \times \psi_{x_{v_j}}\right) \tag{6}$$

where $CP(v_j)$ is 1 if task v_j is on the critical path and 0 otherwise. Due to the parallel execution of some tasks, only the execution time of tasks on the critical path is considered, which incurs the highest execution time and involves the execution time of other parallel tasks as well.

3.2.2 Energy Consumption Model

We only consider the energy consumption of IoT devices in this work since FSs and CSs are usually connected to constant power supplies [7]. From the IoT devices' perspective, the energy consumption that execution of each task v_j incurs depends on the amount of energy the IoT device consumes until the required input data for that task $\omega_{x_{v_j}}^{input}$ becomes ready plus the required energy for the processing of that task $\omega_{x_{v_j}}^{proc}$:

$$\omega_{x_{v_j}} = \omega_{x_{v_j}}^{proc} + \omega_{x_{v_j}}^{input} \tag{7}$$

where $\omega_{x_{v_j}}^{proc}$ depends whether the task is assigned to the IoT device for local execution or not. Hence, we define an IoT Server identifier $IS(x_{v_j})$ to show whether the x_{v_j} refers to an IoT device ($IS(x_{v_j}) = 1$) or other servers ($IS(x_{v_j}) = 0$). Accordingly, the $\omega_{x_{v_j}}^{proc}$ is calculated as what follows:

$$\omega_{x_{v_j}}^{proc} = \begin{cases} \psi_{x_{v_j}}^{proc} \times P^{cpu}, & IS(x_{v_j}) = 1 \\ \psi_{x_{v_j}}^{proc} \times P^{idle}, & IS(x_{v_j}) = 0 \end{cases} \tag{8}$$

If the task is assigned to the IoT device (i.e., $IS(x_{v_j}) = 1$), the energy consumption of the IoT device is equal to the amount of time that it processes the task multiplied by the CPU power of IoT device P^{cpu}. However, if the task is assigned to the other servers for processing (i.e., $IS(x_{v_j}) = 0$), the energy consumption of the IoT device depends on its idle time and corresponding idle power P^{idle}.

The $\omega_{x_{v_j}}^{input}$ depends on the assigned servers to current task (i.e., x_{v_j}) and its predecessors, and is calculated as what follows:

$$\omega_{x_{v_j}}^{input} = \begin{cases} \psi_{x_{v_j}}^{input} \times P^{tra}, & IS(x_{v_j}) = 1 \\ \max(IS(x_{v_i}) \times \left(\frac{e_{i,j}^w}{b_{x_{v_i},x_{v_j}}} + l_{x_{v_i},x_{v_j}}\right) & IS(x_{v_j}) = 0 \\ \times SS(x_{v_i}, x_{v_j})) \times P^{tra} + (\psi^{idle} \times P^{idle}), & \\ \forall v_i \in \mathcal{P}(v_j), & \end{cases} \tag{9}$$

where $IS(x_{v_j})$ and $IS(x_{v_i})$ demonstrates whether the current task v_j and/or its parent task $v_i \in \mathcal{P}(v_j)$ in each edge are assigned to the IoT device or not, respectively. It is important to note that the transmission energy consumption for each edge in DAG is only considered when one of the tasks is placed on the IoT device. Hence, if the current task is assigned to the IoT device (i.e., $IS(x_{v_j}) = 1$), the $\omega^{input}_{x_{v_j}}$ depends on the $\psi^{input}_{x_{v_j}}$. However, if the current task is not assigned to the IoT device (i.e., $IS(x_{v_j}) = 0$), it is possible that the predecessor tasks of the current task (i.e., $\forall v_i \in \mathcal{P}(v_j)$) are previously assigned to the IoT device, and hence the IoT device should forward the data to the server on which the current task is assigned (which incurs energy consumption). If none of the tasks are assigned to the IoT device for local execution, the IoT device is in its idle state. Besides, P^{tra}, ψ^{idle} represent the transmission power of the IoT device and its idle time, respectively. Similar to [7], [36], [37], we used constant values for P^{tra}, ψ^{idle}, however, these parameters also can be dynamically configured.

The main goal of the energy consumption model is to find the best-possible placement configuration for the IoT application so that its energy consumption becomes minimized. Assuming an IoT application consists of L tasks, the energy consumption model is defined as:

$$\Omega(\mathcal{X}) = \min\left(\sum_{j=1}^{L} CP(v_j) \times \omega_{x_{v_j}}\right) \tag{10}$$

3.2.3 Weighted Cost Model

The weighted execution cost of task v_j is defined based on its assigned server x_{v_j}:

$$\phi_{x_{v_j}} = (w_1 \times \psi_{x_{v_j}}) + (w_2 \times \omega_{x_{v_j}}) \tag{11}$$

where $\psi_{x_{v_j}}$ and $\omega_{x_{v_j}}$ refer to the execution time and energy consumption for the execution of task v_j. Moreover, the w_1 and w_2 are control parameters to represent the importance of decision parameters in weighted execution cost of each task. Also, the weighted cost of each task can be changed to execution time or energy consumption cost of each task by assigning $w_1 = 1, w_2 = 0$ or $w_1 = 0, w_2 = 1$, respectively.

Finally, the goal of weighted cost model is to find the best placement configuration for tasks of an IoT application while minimizing the weighted cost of parameters. In this work, we consider execution time of IoT applications and energy consumption of IoT devices as decision parameters, however, this weighted cost can be extended using other decision parameters. The weighted cost model is defined as:

$$\min\Phi(\mathcal{X}) = \min w_1 \times \Psi(\mathcal{X}) + w_2 \times \Omega(\mathcal{X}) \tag{12}$$

$s.t.$

$$C1: \ Size(x_{v_j}) = 1, \ \forall x_{v_j} \in \mathcal{X} \tag{13}$$

$$C2: \ \Phi(v_i) \leq \Phi(v_i + v_j), \ \forall v_i \in \mathcal{P}(v_j) \tag{14}$$

$$C3: \ v_j^{ram} \leq RAM(x_{v_j}), \ \forall v_j \in \mathcal{V} \tag{15}$$

$$C4: w_1 + w_2 = 1 \tag{16}$$

where $\Psi(\mathcal{X})$, $\Omega(\mathcal{X})$ are obtained from Eqs. (6) and (10), respectively. Besides, w_1 and w_2 are control parameters for execution time and energy consumption, by which the weighted cost model can be tuned. $C1$ denotes that each task can only be assigned to one server at a time for processing. Moreover, $C2$ states that the task v_j can only be executed after the execution of its predecessors, and hence the cumulative execution cost of v_j is always larger or equal to execution cost of its predecessors' tasks [7]. Besides, $C3$ states that the assigned server to the task v_j should have sufficient amount of available RAM $RAM(x_{v_j})$ for the processing. Also, $C4$ defines a constraint on the values of control parameters. These constraints are also valid for execution time and energy consumption models. Moreover, the weighted cost model can be changed to execution time or energy consumption model by assigning $w_1 = 1, w_2 = 0$ or $w_1 = 0, w_2 = 1$, respectively.

Since the application placement problem in heterogeneous environments is an NP-hard problem [31], the problem's complexity grows exponentially as the number of heterogeneous servers and/or the number of tasks within an IoT application increases. Thus, the optimal policy of the application placement problem cannot be obtained in polynomial time by iterative approaches. The existing application placement techniques are mostly based on heuristics, rule-based policies, and approximation algorithms [16], [23]. Such techniques work well in general cases, however, they cannot fully adapt to dynamic computing environments where the effective parameters of workloads and computational resources continuously change [16], [38]. To address these issues, DRL-based scheduling/placement algorithms are promising avenues for dynamic optimizations of the system [15], [23].

4 Deep Reinforcement Learning Model

The DRL is a general framework that incorporates deep learning to solve decision-making problems with high-dimensional inputs. Formally, learning problems in DRL can be modeled as Markov Decision Processes (MDP), which is extensively used in sequential stochastic decision making problems. A learning problem can be defined by a tuple $< \mathbb{S}, \mathbb{A}, \mathbb{P}, \mathbb{R}, \gamma >$, in which $\mathbb{S}$ and $\mathbb{A}$ denote the state and action spaces, respectively. $\mathbb{P}$ illustrates the state transition probability, and $\mathbb{R}$ is a reward function. Finally, $\gamma \in [0, 1]$ is a discount factor, determining the importance of future rewards. We suppose that the time horizon is separated into multiple time periods, called time steps $t \in \mathbb{T}$. The DRL agent interacts with the environment, and in each time step t, it perceives the current state of the environment s_t, and selects an action a_t based on its policy $\pi(a_t|s_t)$, mapping states to actions. Considering the selected action a_t, the agent receives a reward r_t from the environment, and it can perceive the next state s_{t+1}. The main goal of the agent is to find a policy in order to maximize the expected total of future discounted reward [24]:

$$\mathbb{V}^{\pi}(s_t) = \mathbb{E}_{\pi}\left[\sum_{t \in \mathbb{T}} \gamma^t r_t\right] \tag{17}$$

where $r_t = \mathbb{R}(s_t, a_t)$ is the reward at time step t, and $a_t \sim \pi(.|s_t)$ is the generated action at time step t by following the

policy π. Moreover, when DNN is used to approximate the function, the parameters are denoted as θ.

Considering the application placement in fog computing environments, we define the main concept of the DRL for our problem as what follows:

- *State space* $\mathbb{S}$*:* In our application placement problem, the state is the observations of the agent from the heterogeneous fog computing environment. Thus, the state at time step t (s_t) consists of information about all heterogeneous servers (such as processing speed of CPU, number of CPU cores, CPU utilization, access Bandwidth (i.e., data rate) of servers, access latency of servers, and CPU, transmission, and idle power consumption values of IoT device). For the rest of the servers, their power consumption values are ignored as we only consider energy consumption from IoT devices' perspective [7]. If for each server we have n features to represent its information, the feature vector of all M servers at time step t ($FV_t^{\mathcal{M}}$) can be presented as:

$$FV_t^{\mathcal{M}} = \{f_i^{m^{y,z}} | \forall m^{y,z} \in \mathcal{M}, 1 \leq \ i \leq n\} \tag{18}$$

where $f_i^{m^{y,z}}$ shows the ith feature of the server $m^{y,z}$. Moreover, s_t contains the information about the current task to be processed within a DAG of an IoT application (such as computation requirements of the task, required RAM, amount of output data per parent task, and current placement configuration of all tasks). Since we consider that tasks are sorted and their dependencies are satisfied before their execution, the current placement configuration of tasks contains the information regarding assigned servers to all previous tasks. The values of unprocessed tasks are set to -1. If we assume that each task has b features, the feature vector of task v_j ($FV_t^{v_j}$) can be represented as:

$$FV_t^{v_j} = \{f_i^{v_j} | v_j \in \mathcal{V}, \forall i \ 1 \leq i \leq b\} \tag{19}$$

where $f_i^{v_j}$ shows the ith feature of the task v_j. Thus, the system space can be defined as:

$$\mathbb{S} = \{s_t | s_t = (FV_t^{\mathcal{M}}, FV_t^{v_j}), \forall t \in \mathbb{T}\} \tag{20}$$

- *Action space* $\mathbb{A}$*:* Actions are assignments of available servers to tasks of an IoT application. Therefore, the action at time step t (a_t) is equal to assigning a server $m^{y,z}$ to the current task v_j. Considering the placement configuration of each task $x_{v,j}$ in Section 3.2, a_t can be defined as:

$$a_t = x_{v_j} = m^{y,z} \tag{21}$$

Thus, the action space $\mathbb{A}$ can be defined as the set of all available servers, presented as follows:

$$\mathbb{A} = \mathcal{M} \tag{22}$$

- *Reward function* $\mathbb{R}$*:* The goal is to minimize the weighted cost model, defined in Eq. (12). To obtain this, we consider Eq. (11) as the weighted cost of each task and define the $\mathbb{R}$ as the negative value of Eq. (11) if the task can be executed ($done = 1$). Moreover, we define a constant $penalty$ value, which is usually a very large negative number [31]. Furthermore, the $penalty$ value can be dynamically set based on the goal of the optimization problem and environmental variables. If the selected action by the agent (i.e., server assignment for the current task) cannot be performed due to any reason ($done = 0$), the reward becomes equal to $penalty$. Accordingly, r_t is defined as:

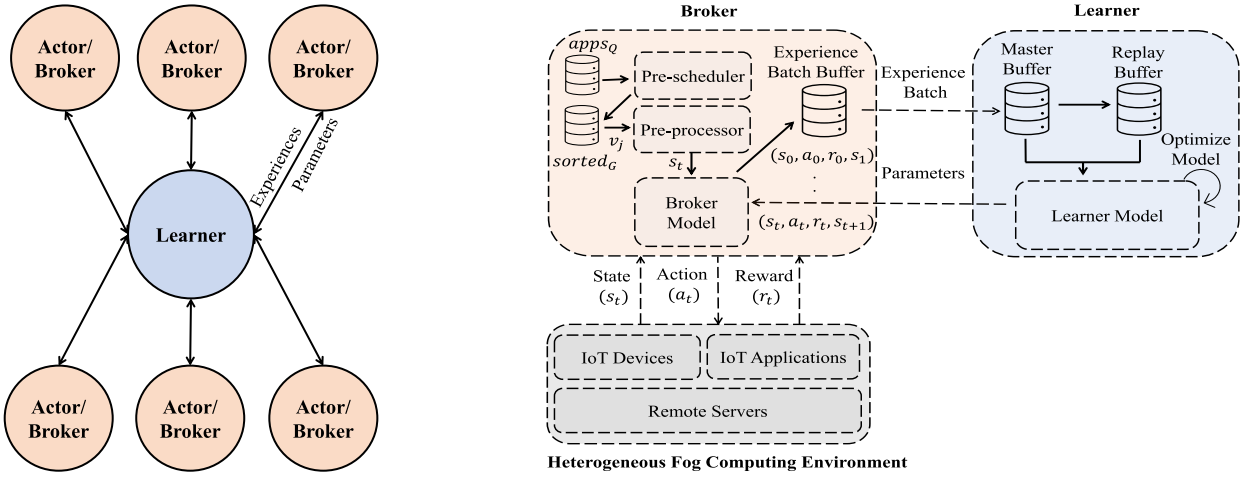
$$r_t = \begin{cases} -\phi_{x_{v_j}} & done = 1 \\ penalty & done = 0 \end{cases} \tag{23}$$

5 Distributed DRL-Based Framework

To address the challenges of DAG-based application placement in the heterogeneous fog computing environment, the X-DDRL works based on an actor-critic framework, aiming at taking advantage of both value-based and policy-based techniques while minimizing their drawbacks [35].

Actor-Critic Framework. In an actor-critic framework, the policy is directly parameterized, denoted as $\pi(a_t|s_t;\theta)$, and the θ is updated by calculating the gradient ascent on the variance of the expected total future discounted reward (i.e, $\sum_{k=0}^{\infty} \gamma^k r_{t+k}$) and the learned state-value function under policy π (i.e., $\mathbb{V}^{\pi}(s_t)$) [35]. The actor interacts with the environment and receives state s_t, outputs the action a_t based on $\pi(a_t|s_t;\theta)$, and receives the reward r_t and next state s_{t+1}. The critic, on the other hand, uses rewards to evaluate the current policy based on the Temporal Difference (TD) error between current reward and the estimation of the value function $\mathbb{V}(s_t;\theta)$. Both actor and critic use DNNs as their function approximators, which are trained separately. To improve the selection probability of better actions by the actor, the parameters of the actor network are updated using the feedback of the TD-error, while the network parameters of the critic network are updated to achieve better value estimation. While the actor-critic frameworks work very well in long-term performance optimizations, their learning speeds are slow and they incur huge exploration costs, especially in problems with high dimensional-state space [31]. The distributed learning techniques in which diverse trajectories are generated in parallel can greatly improve the exploration costs and learning speed of actor-critic frameworks.

The X-DDRL works based on an actor-learner framework, in which the process of generating experience trajectories is separated from learning the parameters of π and $\mathbb{V}^{\pi}$. Fig. 2a demonstrates a high-level overview of learner and actors. The distributed actors in fog computing environments, which can be multiple CPUs within a broker (i.e., FS) or different brokers, interact with their fog computing environments. Arriving application placement requests to each broker are queued in the $apps_Q$ based on the First-In-First-Out (FIFO) policy. As Fig. 2b depicts, brokers performs pre-scheduling phase for each IoT application. Then, based on features of available servers and current task of selected IoT application, each broker pre-processes the current state and



(a) High-level overview of learner and actors/brokers

(b) Communication between learner and each broker

Fig. 2. An overview of X-DDRL framework.

makes an application placement decision. Each broker periodically sends its local experience trajectories to the learner. Besides, the learner updates the target policy π based on collection of received trajectories from different brokers and past trajectories stored in the replay buffer. After each policy update of the learner, brokers update their local policy μ with the policy of the learner π.

The X-DDRL is divided into two phases: pre-scheduling and application placement technique. In the pre-scheduling, tasks of the received IoT application are ranked and sorted in a sequence for the execution. Afterward, for each task of an IoT application, X-DDRL makes a placement decision to minimize the execution cost of the IoT application.

5.1 X-DDRL: Pre-Scheduling Phase

IoT applications are heterogeneous in terms of the number of tasks per application, the dependency model, and corresponding weights of vertices and edges. Considering the dependency model of an IoT application, tasks should be sorted for execution, so that task v_j cannot be executed before any task $v_i \in \mathcal{P}(v_j)$. Furthermore, there are several tasks that can be executed in parallel, and the order of execution of such parallel tasks are also important and may affect the execution cost of an IoT application. Fig. 3 shows a sample IoT application, dependencies among tasks, and parallel tasks with the same colors in each row.

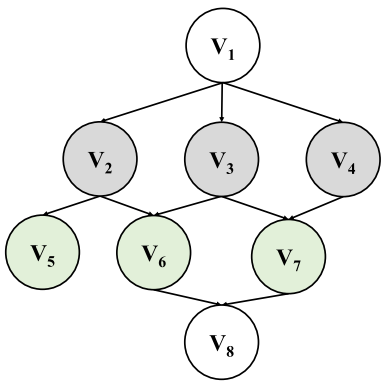


Fig. 3. A sample IoT application (parallel tasks have same colors in each row).

Algorithm 1. The Role of Each Broker/Actor

Input : π: The learner policy
/* N: number of steps, μ: the actor's local policy, EBB: expeience batch buffer, $apps_Q$: Queue of all received IoT applications, G: current IoT application */
1: flag-init=True
2: **for** $t = 0$ *to* ∞ **do**
3: μ=UpdateLocalPolicy(μ, π)
4: **for** $i = t$ *to* $N + t - 1$ **do**
5: **if** *flag-init=True* **then**
6: G=$apps_Q$.dequeue()
7: $sorted_G$ = Pre-scheduling (G) % based on Eq. (24)
8: s_i=ReceiveInitialState(G, $\mathcal{M}$, $sorted_G$)
9: flag-init=False
10: **else**
11: s_i=ReceiveCurrentState()
12: **end**
13: s_i=Pre-processor(s_i)
14: a_i=PlacementEngine(s_i, μ) % calculates the action
%The environment then executes this action
15: r_i=TaskCostCalculator(s_i, a_i) % baed on Eq. (23)
16: s_{i+1} = BuildNextState(s_i, a_i)
17: EBB.update(s_i, a_i, r_i, s_{i+1})
18: **if** *Finish*(G) **then**
19: CalculateTotalCost(G) % based on Eq. (12)
20: flag-init=True
21: **end**
22: **end**
23: **if** *size*(EBB)==N **then**
24: SendExpeienceToLearner(EBB)
25: **end**
26: **end**

Whenever a broker receives a DAG-based IoT application request from a user, it creates a sequence of tasks for the execution while considering above-mentioned challenges. Tasks within the IoT application are ranked based

on the non-increasing order of their rank value. The rank value of a task is defined as:

$$Rank(v_j) = \begin{cases} \widetilde{\phi_{x_{v_j}}} + \max(\widetilde{\phi_{x_{v_i}}}) & \text{if } v_{n,j} \neq exit \\ \quad \forall v_i \in \mathcal{P}(v_j), & \\ \widetilde{\phi_{x_{v_j}}}, & \text{if } v_{n,j} = exit \end{cases} \tag{24}$$

where $\widetilde{\phi_{x_{v_j}}}$ shows the average weighted execution cost of task $v_{n,j}$ on considering different servers. The rank is calculated recursively by traversing the DAG of the application, starting from the exit module. Using the rank function, the tasks on the critical path of DAG (i.e., CP) can also be identified. Hence, not only does the rank function satisfy the dependency among tasks, but it also defines an execution order for tasks that can be executed in parallel. To achieve this, it gives higher priority to tasks that incur higher total execution costs among parallel tasks.

5.2 X-DDRL: Application Placement Phase

If we assume that each broker makes placement decisions for tasks of IoT applications, using their local policy μ, for N steps in the time horizon starting at time $i = t$, Algorithm 1 shows how brokers perform application placement decisions and generate experience trajectories. Each broker performs the following steps: At the beginning of each trajectory, the broker updates its policy μ with the policy of the learner (line 3). When broker starts making placement decisions for tasks of a new IoT application G (i.e., when the flag-init=$True$), it receives the current IoT application from the $apps_Q$ (contains all received application requests to this broker) (line 6). Then, the broker performs the pre-scheduling to obtain the sorted list of application' tasks based on the Eq. (24) (line 7). Next, the system state is generated using the initial state of the IoT application G and available servers $\mathcal{M}$ (line 8). Moreover, the broker changes the flag-init to $False$, indicating that in the subsequent steps there is no need to re-calculate the ranking and initial state of the G (line 9), and the broker only requires to obtain the current state of the environment based on Eq. (20) (line 11). The current state of the broker's environment s_i consists of feature vectors of servers $FV_t^{\mathcal{M}}$ and the current task of IoT application $FV_t^{v_j}$. The current task of each IoT application is obtained from the ordered sequence of tasks $sorted_G$. Then, the broker pre-processes and normalizes values of the current state (line 13). Considering s_i and current policy μ, an application placement decision (i.e., the assignment of a server for the processing of the current task) is made (line 14). The current task is then forwarded to the assigned server (based on a_i) for processing. After the execution of the task, the broker receives the reward of this action, which is the negative value of the weighted execution cost of this task Eq. (23) (line 15). The next state of the environment is then created using the $BuildNextState$ function (line 16). Then, the broker creates an experience tuple (s_i, a_i, r_i, s_{i+1}) and stores it in its local experience batch buffer (line 17). When the broker finishes assignment of servers to all tasks of the current IoT application G, meaning the application placement is done for the current IoT application, the total weighted execution cost of this IoT application is calculated using Eq. (12) (line 19). Moreover, the broker sets flag-init to $False$ so that the next IoT application in the queue of this broker $apps_Q$ can be served (line 20). After N steps, each broker forwards its experience batch buffer to the learner (lines 23-25). The learner periodically updates its policy (i.e., π) on batches of experience trajectories, collected from several brokers.

Since policies of brokers μ are updated based on the learner's policy (trained on trajectories of different brokers), each broker gets the benefit of trajectories generated by other brokers. It significantly reduces the exploration cost of each broker, and also provides brokers with a more accurate local policy μ. Furthermore, the X-DDRL uses an experience-sharing approach, which significantly reduces communication overhead between brokers and learners, in comparison to gradient-sharing techniques such as A3C [24].

Due to the gap between the policy of broker μ (when generating new decisions) and the policy of the learner π in the training time (when the learner estimates the gradients), the learner in the X-DDRL uses the off-policy correction method, called V-trace [24], to correct this discrepancy.

Algorithm 2. The Role of Each Learner

Input : EB_{broker}: Experience batch of different brokers
/* $list_{brokers}$: list of brokers, π: the learner's policy, MB: master buffer, MBS: master buffer size, RB: replay buffer, RBS: replay buffer size, TB: training batch, TBS: training batch size */
1: **while** *True* **do**
2: flag-training=False
3: MB=$\emptyset$
4: **while** *flag-training==False* **do**
5: MB.update(EB_{broker})
6: **if** $TBS \leq MBS + RBS$ **then**
7: TB=BuildTrainBatch(MB, RB)
8: flag-training==True
9: **end**
10: **end**
11: OptimizeModel(TB) % based on Eq. (28), 29
12: UpdateBrokers($list_{brokers}$, π)
13: **end**

- *V-trace off-policy correction method*: We assume that each broker generates an experience trajectory for N steps while following its local policy μ as $(s_t, a_t, r_t)_{t=i}^{i+N}$. The value approximation of state s_i, defined as N-step V-trace target for $\mathbb{V}(s_i)$, is as follows:

$$\overline{\mathbb{V}_i} = \mathbb{V}(s_i) + \sum_{t=i}^{i+N-1} \gamma^{t-i} \left(\prod_{j=i}^{t-1} c_j \right) \delta_t \mathbb{V} \tag{25}$$

where $\delta_t \mathbb{V}$ is a TD for $\mathbb{V}$, defined as:

$$\delta_t \mathbb{V} = \rho_t (r_t + \gamma \mathbb{V}(s_{t+1}) - \mathbb{V}(s_t)) \tag{26}$$

where $\rho_t = min(\overline{\rho}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)})$ and $c_j = min(\overline{c}, \frac{\pi(a_j|s_j)}{\mu(a_j|s_j)})$ are truncated Importance Sampling (IS) weights, while

$\overline{c} \leq \overline{\rho}$. The $\overline{c}$ and $\overline{\rho}$ play different roles in the V-trace. The $\overline{\rho}$ has a direct effect on the value function $\mathbb{V}^{\pi}$ toward which we converge, while $\overline{c}$ has a direct effect on speed of the convergence. Considering ρ, the target policy of the learner π can be defined as:

$$\pi_{\overline{\rho}}(a|s) = \frac{\min(\overline{\rho}\mu(a|s), \pi(a|s))}{\sum_{b \in \mathbb{A}} \min(\overline{\rho}\mu(b|s), \pi(b|s)))} \tag{27}$$

We consider: (1) the brokers generate trajectories while following policy μ, (2) the parameterized state-value function under θ as $\mathbb{V}_{\theta}$, (3) the current policy of learner is π_u, and (4) the V-trace target $\overline{\mathbb{V}_i}$ is defined based on Eq. (25). The learner updates value parameters θ, at time step i, in the direction of:

$$(\overline{\mathbb{V}_i} - \mathbb{V}_{\theta}(s_i))\nabla_{\theta}\mathbb{V}_{\theta}(s_i) \tag{28}$$

Moreover, the policy parameters u are updated in the direction of the policy gradient using Adam optimization algorithm [39]:

$$\rho_i \nabla_u \log(\pi_u(a_i|s_i))(r_i + \gamma\overline{\mathbb{V}_{i+1}} - \mathbb{V}_{\theta}(s_i)) \tag{29}$$

Algorithm 2 summarizes the learners' role in the X-DDRL. The learner continuously receives and stores experience batches of brokers EB_{broker} and updates the master Buffer MB until the training batch TB becomes full (line 4-10). Then, the learner optimizes the current target policy π based on Eqs. (28) and (29) (line 11). After policy update of the learner, brokers update their local policies μ with the latest policy of the learner π (i.e., brokers set their policies to the new learner policy), and hence, new application placement decisions are made using the updated policy μ in the brokers. The learner in the X-DDRL uses the replay buffer RB, which remarkably improves sample efficiency. The X-DDRL can easily scale as the number of servers, IoT application requests, and brokers increases, which is a principal factor in highly distributed environments such as fog computing. If a new broker joins the environment, the broker updates its local policy with the latest policy of the learner, and hence it takes advantage of all trajectories that previously generated by other brokers. Besides, it generates new sets of trajectories which help to better diversify the trajectories of the learner. If the number of servers in the environment increases, distributed brokers quickly generate new sets of trajectories, and accordingly the learner can update its target policy promptly. Such a collaborative distributed broker-learner architecture not only significantly improves the exploration costs, but also improves the convergence speed. The other improvement in the X-DDRL is using RNN layers since they can accurately identify highly non-linear patterns among different input features, resulting in significant speedup in the learner [23], [40].

5.3 Discussion on Resource Contention

In heterogeneous computing environments where multiple applications are forwarded to heterogeneous servers, resource contention for computing resources may occur. Let's assume, there are three IoT applications named $A1$, $A2$, and $A3$ while there are two servers (either at the edge or cloud) called $S1$ and $S2$. The type of applications may be different from each other with different resource requirements, and the number of servers or their computing capability may differ so that resource contention occurs among IoT applications. Moreover, for the DAG-based IoT applications, consisting of several dependent tasks, another type of resource contention may happen among tasks of one IoT application.

One approach to solve the resource contention, either among different IoT applications or tasks of one application, is prioritization. In X-DDRL, the FIFO policy is used to prioritize different IoT applications. These policies can be changed according to the targeted problem. Besides, for the tasks of one application, there are two important points to consider. In DAG-oriented IoT application, each task within the application can only be executed if its predecessor tasks are completed. However, for tasks that can be executed in parallel, a priority should be defined. In X-DDRL, we use a rank function, Eq. (24), that prioritizes tasks of one IoT application while considering the dependency among tasks and the average execution cost of tasks. Such prioritization between different IoT applications and tasks of one IoT application help to solve the resource contention.

Moreover, the DRL agent receives the sequence of tasks based on the above-mentioned policies. So, these prioritization techniques are very important for the long-term reward, especially for DAG-based IoT applications having extra constraints. Without such prioritization, the DRL agent may converge to a good solution but the convergence time is significantly higher while in some scenarios the DRL agent even cannot converge to good solution. The DRL agent learns to assign the best server to each task while considering the tasks' dependency model of an IoT application, resource requirements of each task, and available heterogeneous resources in the environment. That is, the DRL agent is considering the resource contention while assigning a server to each task, to minimize the execution cost of each task (based on short-term reward) and accordingly each IoT application (based on long-term reward).

6 Performance Evaluation

This section first describes the experimental setup, used to evaluate our technique and baseline algorithms. Next, the hyperparameters of our proposed technique X-DDR are discussed. Finally, we study the performance of X-DDRL and its counterparts in detail.

6.1 Experimental Setup

To evaluate the performance of the X-DDRL, we use both simulation environment and testbed, which their specification are provided in what follows.

6.1.1 Simulation Setup

We developed an event-driven simulation environment in Python using the OpenAI Gym [41] for the application placement in heterogeneous fog computing environments, similar to [16]. For each of the two learners, we set the

number of brokers to 8, which have access to a set of servers, and make application placement decisions accordingly. Hence, we vectorized the fog computing environment, generated using OpenAI Gym, so that distributed brokers can interact with their fog computing environments and make application placement decisions in parallel. Unlike prior work [16], [31], [32], [33], we consider a heterogeneous fog computing environment consisting of IoT devices, resource-constrained FSs, and resource-rich CSs. In fog computing environment, we used the following server setup, unless it is stated in the experiments: two Raspberrypi 3B (Broadcom BCM2837 4 cores @1.2GHz, 1GB RAM)[1], one Raspberrypi 4B (ARM Cortex-A72 4 cores @1.5GHz, 4GB RAM),[2] and one Jetson Nano (ARM Cortex-A57 4 cores @1.43GHz, 4GB RAM, 128-core Maxwell GPU)[3] as heterogeneous FSs. Besides, to simulate a heterogeneous multi-cloud environment, we used specifications of six m3.large instances of Nectar Cloud infrastructure (AMD 8 cores @2GHz, 16GB RAM)[4] and two instances of the University of Melbourne Horizon Cloud (Intel Xeon 8 cores @2.4GHz, 24GB RAM, NVIDIA P40 3GB RAM GPU).[5] For IoT devices, the server type is a single core @1GHz device embedded with 512MB RAM [16]. Besides, the power consumption of each IoT device in processing, idle, and transmission state is 0.5W, 0.002W, and 0.2W, respectively [7]. The bandwidth (i.e., data rate) and latency among different servers and IoT devices are also obtained based on average profiled values from testbed, similar to [23]. Hence, the latency of FSs and CSs are considered as 1ms and 10ms respectively, similar to [23]. The bandwidth between IoT devices and FSs is randomly selected between 10-12MB/s, while the bandwidth between IoT devices and FSs to the CSs is randomly selected between 4-8 MB/s, similar to [37]. Although we obtained these values based on testbed experiments, they are referred to some similar works as well to show the credibility of these values. Also, both w_1 and w_2 are set to 0.5, meaning that the importance of execution time and the energy consumption is equal in the results. However, these parameters can be adjusted based on the users' requirements and network conditions.

Many real-world IoT applications can be modeled by DAGs with a different number of tasks and dependency models. Hence, we generated several synthetic DAG sets with a different number of tasks and dependency models to represent scenarios where IoT devices generate heterogeneous DAGs with different preferences, similar to [16], [42]. The dependency model of each DAG can be identified using three parameters: number of tasks within an application L, fat that controls the width and heights of a DAG, and $density$ that identifies the number of edges between different levels of the DAG. Accordingly, we generated different DAG datasets, where each dataset contains 100 DAGs with a similar number of tasks, fat, and density while the weights are randomly selected to represent heterogeneous task requirements in IoT applications with the same DAG structure. To generate heterogeneous DAG datasets, we set task numbers $L \in \{10, 15, 20, 25, 30, 35, 40, 45, 50\}$, $fat \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$, and $density \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$. To illustrate, one dataset of DAGs is $L = 10$, $fat = 0.4$, and $density = 0.4$, containing 100 DAGs. Accordingly, for each task number L, we have 25 different combinations of fat and $density$, resulting in 25 different topologies and 2500 DAGs. Finally, the simulation experiments are all performed on an instance of Horizon Cloud with the abovementioned specifications.

6.1.2 Testbed Setup

To evaluate the performance of X-DDRL in a real-world scenario, we created a testbed, similar to [31]. The type of servers are the same as simulation setup while the number of servers of each type is as follows: two Raspberry pi 3B, one Raspberry pi 4B, one Jetson Nano, one instance of Horizon Cloud, and six m3.large instances of Nectar Cloud infrastructure). As IoT devices, we created several single-core VMs within a PC (HP Elitebook 840 G5 with Intel Core i7-8550U 8 cores @2GHz and 16GB RAM). These VMs are used to send application placement requests, using described DAG datasets, to the brokers. Moreover, to estimate the energy consumption of IoT devices, we used computing power, transmission power, and idle power as discussed in Section 6.1.1, similar to the approach in [31]. For the connectivity, we set up a virtual network using VPN among IoT devices, FSs, and CSs, as described in [26]. Due to the limited CPU and RAM of the IoT devices' VMs, they can send application placement requests, using a message-passing protocol (implemented using HTTP requests), to the broker that is the Jetson Nano in this testbed. The broker runs a multi-threaded server application that receives application placement requests from different IoT devices and puts them in the queue based on the FIFO policy. The broker dequeues the requests and makes placement decisions for the tasks according to its policy μ. According to the placement configuration for each IoT application, each server that receives a task for processing assigns that task to one of its threads for processing. The thread is kept busy according to the weight of task and processing speed of the server. After the execution of each task, the size of output results that should be forwarded to the children tasks is obtained based on the weights of the task's outgoing edges in each DAG. Since weights of edges in each DAG (i.e., data to be transferred between tasks) are different, we generate files with different sizes to represent the weights on edges. Finally, the broker logs the execution cost of each IoT application and all of its constituent tasks in terms of selected evaluation metrics.

6.1.3 Baseline Algorithms

We evaluate the performance of the X-DDRL with a greedy heuristic algorithm, and two DRL-based techniques from the literature that proposed DRL-based solutions for DAG-based IoT applications. In what follows, we briefly describe how these techniques are implemented, while their detailed specifications are provided in Section 2.

1. https://www.raspberrypi.org/products/raspberry-pi-3-model-b
2. https://www.raspberrypi.org/products/raspberry-pi-4-model-b
3. https://developer.nvidia.com/embedded/jetson-nano-developer-kit
4. https://nectar.org.au/
5. https://people.eng.unimelb.edu.au/lucasjb/horizon/

TABLE 2
The DNN and Training Hyperparameters

Parameter	Value	Parameter	Value
Fully Connected layers	2	Learning Rate lr	0.01
LSTM Layers	2	Discount Factor γ	0.99
Optimization Method	Adam	V-trace $\overline{\rho}$	1
Activation Function	Tanh	V-trace $\overline{c}$	1

- *PPO-RNN*: It is the extended and adapted version of the technique proposed in [16].[6] We extended this technique so that it can be used in multi-objective scenarios to minimize the weighted cost of execution. Besides, this technique is extended to be used in heterogeneous fog computing environments where several IoT devices, FSs, and CSs are available. This technique uses PPO as its DRL framework while the networks of the agent are wrapped by the RNN. Besides, we used the same hyperparameters as [16].
- *PPO-No-RNN*: This technique is the same as PPO-RNN, while the networks are not wrapped by the RNNs.
- *Double-DQN*: Many works in the literature uses standard Deep Q-Learning (DQN) based RL approach such as [18], [30], [32], [33]. We implemented the optimized Double-DQN technique with an adaptive exploration for application placement in heterogeneous fog computing environments.[7] The hyperparameters of this technique are set based on [33], which is a DQN-based application placement technique for DAG-based IoT applications.
- *Greedy*: In this technique, tasks are greedily assigned to the servers if their execution cost is less than the estimated local execution cost, similar to [16].

6.2 X-DDRL Hyperparameters

In the implementation of X-DRRL, where the standard implementation of IMPALA is used[8], the DNN structure of all agents is similar, consisting of two fully connected layers followed by two LSTM layers as recurrent layers. Moreover, we performed a grid search to tune hyperparameters. According to tuning experiments, we set the learning rate lr to 0.01, the discount factor γ to 0.99. Besides, values of $\overline{\rho}$ and $\overline{c}$, controlling the performance of V-trace are set to 1 [24] to obtain the best result. Table 2 summarizes the setting of hyperparameters.

6.3 Performance Study

In this section, four experiments are conducted to evaluate and compare the performance of X-DDRL with other techniques in terms of weighted execution cost, execution time of IoT applications, and energy consumption of IoT devices.

6.3.1 Execution Cost versus Policy Update Analysis

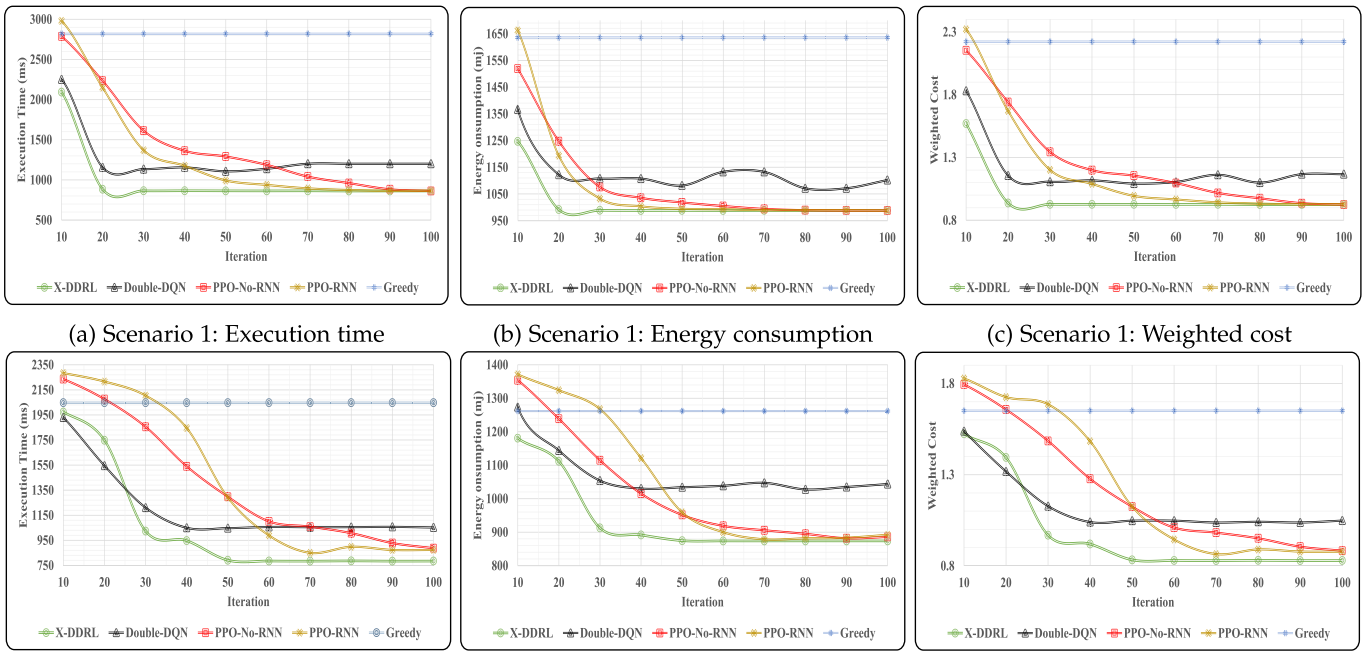
In this experiment, we study the performance of application placement techniques in different iterations of the policy updates. We consider two scenarios for datasets of IoT applications to analyze how efficiently these techniques can extract features of different datasets of IoT applications and optimize their target policy. In the first scenario, we consider the number of tasks within IoT applications $L = 30$. Hence, 25 datasets of IoT applications with the same task number and different fat and $density$ are used, among which 20 datasets are used for the training and 5 datasets are used for the evaluation. In the second scenario, for the training $L \in \{10, 15, 25, 30\}$ while for the evaluation $L = 20$. Therefore, the training and evaluation are performed on datasets with a different number of tasks. Fig. 4 shows the obtained results of this study in terms of the average execution time of IoT applications, the energy consumption of IoT devices, and weighted cost for the above-mentioned two scenarios.

As Fig. 4 shows, the average execution cost of all techniques, except the greedy, decreases in different scenarios as the iteration number increases. However, the X-DDRL converges faster and to better placement solutions in comparison to other techniques. This is mainly because the V-trace function embedded in the X-DDRL uses n-step state-value approximation rather than 1-step state-value approximation [24], improving convergence speed of X-DDRL to better solutions. Moreover, trajectories generated by distributed brokers are diverse, leading to a more efficient learning process. The execution cost of the greedy technique is fixed and does not change with different iteration numbers, but it can be used as a baseline technique to compare the performance of DRL-based techniques. The convergence speed of PPO-RNN and PPO-NO-RNN techniques is slower than the Double-DQN technique however, they finally converge to better placement solutions. In addition, the obtained results of the second scenario (Figs. 4d, 4e, and 4f) shows that all DRL-based techniques has lower convergence speed in comparison to the obtained results of first scenario (Figs. 4a, 4b, and 4c). However, still X-DDRL outperforms other techniques in terms of execution time, energy consumption, and weighted cost. This proves that the X-DDRL can more efficiently adapt itself with different DAG structures (i.e., task numbers, and dependency model), and hence it makes better application placement decisions in unforeseen scenarios.

6.3.2 System Size Analysis

In this experiment, the effect of different numbers of servers on application placement techniques is studied. The number of candidate servers has a direct effect on the complexity of application placement problems because the larger number of servers leads to a bigger search space. Hence, to analyze the performance of X-DDRL, the default number of servers in this experiment is multiplied by two and four; i.e, we have 24 and 48 servers respectively. Moreover, in this experiment, the training and evaluation datasets are specified as the same as the first scenario in Section 6.3.1; i.e., a total of 25 datasets where $L = 30$ and different fat and

6. https://github.com/linkpark/metarl-offloading
7. https://docs.ray.io/en/master/
8. https://docs.ray.io/en/master/



(a) Scenario 1: Execution time (b) Scenario 1: Energy consumption (c) Scenario 1: Weighted cost

(d) Scenario 2: Execution time (e) Scenario 2: Energy consumption (f) Scenario 2: Weighted cost

Fig. 4. Execution cost versus policy update analysis: In scenario 1, the training and evaluations are performed on datasets where $L = 30$. In scenario 2, the training is performed on datasets where $L \in \{10, 15, 25, 30\}$ and the evaluation is performed on datasets where $L = 20$.

density values. Due to the space limit and the fact that patterns for execution time, energy consumption, and weighted cost were roughly the same, only the obtained results from the weighted cost are provided in this experiment.

Fig. 5 shows the weighted cost of different techniques, where brokers in the system have access to 24 and 48 candidate servers when making application placement decisions. It is crystal clear that the weighted cost of the greedy technique is steady for 24 and 48 servers as the number of iterations increases. All DRL-based techniques perform better than greedy technique either when the number of servers is 24 or 48. Also, it can be seen that the weighted execution costs of techniques are higher when the number of servers is 48 than weighted costs when the servers' number is 24. As the number of iterations increases, the DRL-based techniques can more accurately make placement decisions, leading to less weighted execution cost. However, the X-DDRL always outperforms other techniques and converges faster to better solutions. It shows that the X-DDRL has better scalability when the system size grows. This helps X-DDRL to make better application placement decisions in a fewer number of iterations. Among other DRL-based techniques, PPO-RNN performs better than PPO-No-RNN and Double-DQN and makes better placement decisions as the iteration numbers increases.

6.3.3 *Speedup and Placement Time Overhead Analysis*

In this section, we study the speedup and placement time overhead of different DRL-based techniques. We follow the same experimental setup as the first scenario in Section 6.3.1. We define the average Placement Time Overhead (PTO) as the average required amount of time for each technique to make an application placement decision divided by the average local execution time of IoT applications on IoT devices. To obtain the local execution time

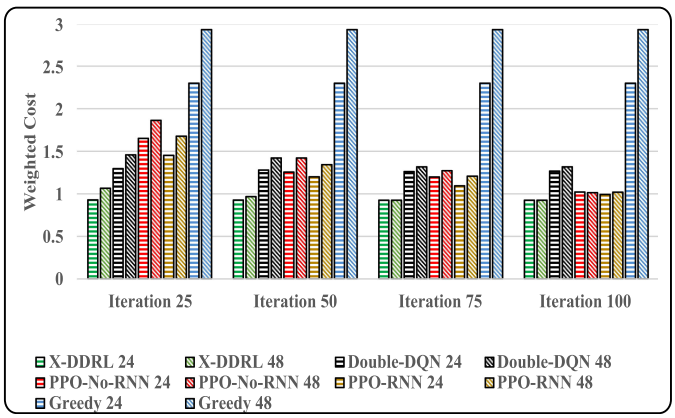


Fig. 5. System size analysis.

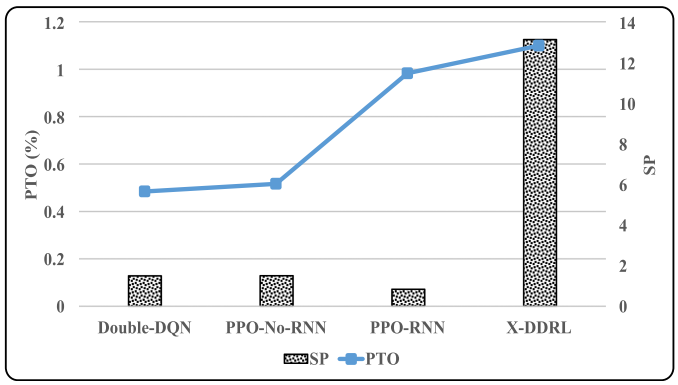
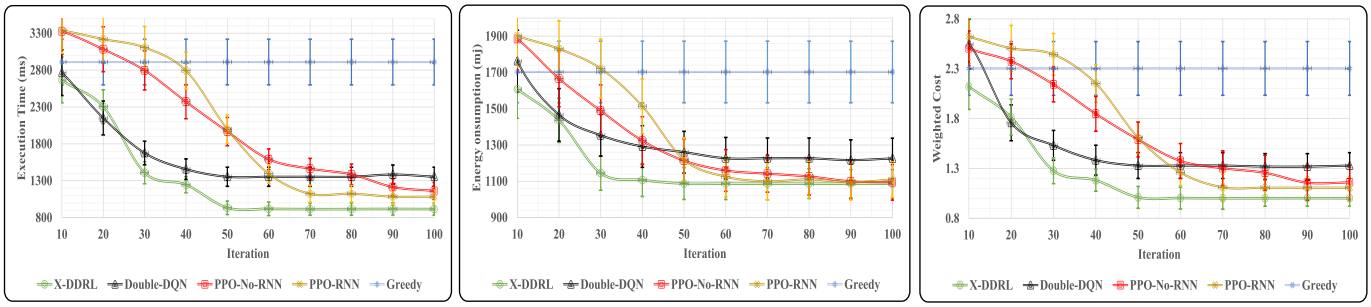


Fig. 6. Placement time overhead and speedup analysis.



(a) Execution time (b) Energy consumption (c) Weighted cost

Fig. 7. Evaluation on testbed.

of IoT applications on IoT devices, we assume that tasks within an IoT application are executed sequentially, similar to [7]. Besides, we define the time taken by the X-DDRL technique with one broker to reach the value 1.1 from the weighted execution cost as $Time_R$. The reason why 1.1 is considered as the reference weighted execution cost is that this value is the minimum weighted execution cost that all DRL-based techniques can obtain. Moreover, the time taken by each technique to reach the reference weighted execution cost is defined as $Time_T$. Accordingly, similar to [23], the Speedup value of each technique (SP) is defined as $SP = \frac{Time_R}{Time_T}$.

Fig. 6 shows results of PTO and SP for all DRL-based techniques. The placement time overhead of techniques using RNN (i.e., X-DDRL and PPO-RNN) is usually higher than techniques that do not use RNN (i.e., Double-DQN, and PPO-No-RNN). The PTO of the X-DDRL is higher than other DRL-based techniques by less than 1% in the worst-case scenario, which is not significantly large. However, the obtained results of SP show that X-DDRL performs 8 to 16 times faster than other techniques. Hence, considering the speedup performance and execution cost results of the X-DDRL, its placement time overhead is negligible, and X-DDRL can more efficiently perform application placement decisions compared to other techniques for heterogeneous fog computing environments.

6.3.4 Evaluation on Testbed

To evaluate the performance of X-DDRL in real-world scenarios, we conducted experiments on the testbed whose configuration is discussed earlier in Section 6.1.2. In this experiment, for the training $L \in \{30, 35, 45, 50\}$ while for the evaluation $L = 40$.

Fig. 7 shows the execution cost of different techniques in terms of execution time, energy consumption, and weighted cost by 95% confidence interval. It can be observed that, similar to the simulation results, X-DDRL can outperform other techniques in terms of execution time, energy consumption, and weighted cost. Moreover, even after 100 iterations, where all techniques converged, there are no techniques that obtain better results in comparison to X-DDRL. It demonstrates that not only does X-DDRL converge faster, and its training time is significantly less than other techniques, but it also provides better results. As the results depict, the optimized Double-DQN technique converges faster than PPO-RNN and PPO-No-RNN, but it cannot obtain results as well as them. Overall, compared to converged results of other DRL-techniques, achieved results of X-DDRL show an average performance gain up to 30%, 11%, and 24% in terms of execution time, energy consumption, and weighted cost, respectively.

7 Conclusions and Future Work

This paper proposes a distributed DRL-based technique, called X-DDRL, to efficiently solve the application placement problem of DAG-based IoT applications in heterogeneous fog computing environments, where edge and cloud servers are collaboratively used. First, a weighted cost model for optimizing the execution time and energy consumption of IoT devices with DAG-based applications in heterogeneous fog computing environments is proposed. Besides, a pre-scheduling phase is used in the X-DDRL, by which dependent tasks of each IoT application are prioritized for execution based on the dependency model of the DAG and their estimated execution cost. Moreover, we proposed an application placement phase, working based on the IMPALA framework for the training of distributed brokers, to efficiently make application placement decisions in a timely manner. Distinguished from existing works, the X-DDRL can rapidly converge well-suited solutions in heterogeneous fog computing environments with a large number of servers and users. The effectiveness of X-DDRL is analyzed through extensive simulation and testbed experiments while comparing with the state-of-the-art techniques in the literature. The obtained results indicate that X-DDRL performs 8 to 16 times faster than other DRL-based techniques. Besides, compared to other DRL-based techniques, it achieves a performance gain up to 30%, 11%, and 24% in terms of execution time, energy consumption, and weighted cost, respectively.

As part of future work, we plan to extend our proposed weighted cost model to consider other aspects such as monetary cost, dynamic changes of transmission power, and total system cost. Moreover, we plan to apply mobility models in this scenario and adapt our proposed application placement technique accordingly.

Acknowledgments

The authors would like to thank Shashikant Ilager, Amanda Jayanetti, and Samodha Pallewatta for their comments on improving this paper.

REFERENCES

[1] P. Hu, S. Dhelim, H. Ning, and T. Qiu, "Survey on fog computing: Architecture, key technologies, applications and open issues," *J. Netw. Comput. Appl.*, vol. 98, pp. 27–42, 2017.

[2] J. Wang, K. Liu, B. Li, T. Liu, R. Li, and Z. Han, "Delay-sensitive multi-period computation offloading with reliability guarantees in fog networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 9, pp. 2062–2075, Sep. 2020.

[3] M. Goudarzi, H. Wu, M. S. Palaniswami, and R. Buyya, "An application placement technique for concurrent IoT applications in edge and fog computing environments," *IEEE Trans. Mobile Comput.*, vol. 20, no. 4, pp. 1298–1311, Apr. 2021.

[4] S. Deng *et al.*, "Optimal application deployment in resource constrained distributed edges," *IEEE Trans. Mobile Comput.*, vol. 20, no. 5, pp. 1907–1923, May 2021.

[5] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Trans. Mobile Comput.*, vol. 20, no. 4, pp. 939–951, Mar. 2021.

[6] M. Goudarzi, M. Zamani, and A. T. Haghighat, "A fast hybrid multi-site computation offloading for mobile cloud computing," *J. Netw. Comput. Appl.*, vol. 80, pp. 219–231, 2017.

[7] X. Xu *et al.*, "A computation offloading method over big data for IoT-enabled cloud-edge computing," *Future Gener. Comput. Syst.*, vol. 95, pp. 522–533, 2019.

[8] M. Goudarzi, M. Palaniswami, and R. Buyya, "A fog-driven dynamic resource allocation technique in ultra dense femtocell networks," *J. Netw. Comput. Appl.*, vol. 145, 2019, Art. no. 102407.

[9] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *Proc. 9th Int. Conf. Mobile Syst.s, Appl., Serv.*, 2011, pp. 43–56.

[10] T. N. Gia, M. Jiang, A.-M. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen, "Fog computing in healthcare Internet of Things: A case study on ECG feature extraction," in *Proc. IEEE Int. Conf. Comput. Inf. Technol.; Ubiquitous Comput. Commun.; Dependable, Autonomic Secure Comput.; Pervasive Intell. Comput.*, 2015, pp. 356–363.

[11] A. Al-Shuwaili and O. Simeone, "Energy-efficient resource allocation for mobile edge computing-based augmented reality applications," *IEEE Wirel. Commun. Lett.*, vol. 6, no. 3, pp. 398–401, Jun. 2017.

[12] A. Brogi and S. Forti, "QoS-aware deployment of IoT applications through the fog," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1185–1192, Oct. 2017.

[13] L. F. Bittencourt, J. Diaz-Montes , R. Buyya, O. F. Rana, and M. Parashar, "Mobility-aware application scheduling in fog computing," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 26–35, Mar./Apr. 2017.

[14] M. Goudarzi, Z. Movahedi, and M. Nazari, "Mobile cloud computing: A multisite computation offloading," in *Proc. 8th Int. Symp. Telecommun.*, 2016, pp. 660–665.

[15] D. Jeff, "ML for system, system for ML, keynote talk in workshop on ML for systems, NIPS," 2018. [Online]. Available: http://mlforsystems.org/

[16] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 242–253, Jan. 2021.

[17] Q. Mao, F. Hu, and Q. Hao, "Deep learning for intelligent wireless networks: A comprehensive survey," *IEEE Commun. Surv. Tut.*, vol. 20, no. 4, pp. 2595–2621, Fourth Quarter 2018.

[18] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 11, pp. 2581–2593, Nov. 2020.

[19] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019.

[20] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digit. Commun. Netw.*, vol. 5, no. 1, pp. 10–17, 2019.

[21] H. Lu, X. He, M. Du, X. Ruan, Y. Sun, and K. Wang, "Edge QoE: Computation offloading with deep reinforcement learning for Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9255–9265, Oct. 2020.

[22] Y. Sun, M. Peng, Y. Zhou, Y. Huang, and S. Mao, "Application of machine learning in wireless networks: Key techniques and open issues," *IEEE Commun. Surv. Tut.*, vol. 21, no. 4, pp. 3072–3108, Fourthquarter 2019.

[23] 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 Trans. Mobile Comput.*, 2020.

[24] L. Espeholt *et al.*, "IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1407–1416.

[25] M. Goudarzi, M. Palaniswami, and R. Buyya, "A distributed application placement and migration management techniques for edge and fog computing environments," 2021, *arXiv:2108.02328*.

[26] Q. Deng, M. Goudarzi, and R. Buyya, "FogBus2: A lightweight and distributed container-based framework for integration of IoT-enabled systems with edge and cloud computing," in *Proc. Int. Workshop Big Data Emergent Distrib. Environ.*, 2021, pp. 1–8.

[27] L. Huang, X. Feng, A. Feng, Y. Huang, and L. P. Qian, "Distributed deep learning-based offloading for mobile edge computing networks," *Mobile Netw. Appl.*, pp. 1–8, 2018.

[28] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, "Learning-based computation offloading for IoT devices with energy harvesting," *IEEE Trans. Veh. Technol.*, vol. 68, no. 2, pp. 1930–1941, Feb. 2019.

[29] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu, "iRAF: A deep reinforcement learning approach for collaborative mobile edge computing IoT networks," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 7011–7024, Aug. 2019.

[30] X. Xiong, K. Zheng, L. Lei, and L. Hou, "Resource allocation based on deep reinforcement learning in IoT edge computing," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 6, pp. 1133–1146, Jun. 2020.

[31] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1085–1101, May 2021.

[32] P. Gazori, D. Rahbari, and M. Nickray, "Saving time and cost on the scheduling of fog-based IoT applications using deep reinforcement learning approach," *Future Gener. Comput. Syst.*, vol. 110, pp. 1098–1115, 2020.

[33] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu, "Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning," *Future Gener. Comput. Syst.*, vol. 102, pp. 847–861, 2020.

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

[35] Q. Qi *et al.*, "Knowledge-driven service offloading decision for vehicular edge computing: A deep reinforcement learning approach," *IEEE Trans. Veh. Technol.*, vol. 68, no. 5, pp. 4192–4203, May 2019.

[36] S. E. Mahmoodi, R. Uma, and K. Subbalakshmi, "Optimal joint scheduling and cloud offloading for mobile applications," *IEEE Trans. Cloud Comput.*, vol. 7, no. 2, pp. 301–313, 2016.

[37] H. Wu, W. Knottenbelt, and K. Wolter, "An efficient application partitioning algorithm in mobile environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 7, pp. 1464–1480, Jan. 2019.

[38] G. Fox, J. A. Glazier, J. Kadupitiya, V. Jadhao, M. Kim, J. Qiu, J. P. Sluka, E. Somogyi, M. Marathe, A. Adiga *et al.*, "Learning everywhere: Pervasive machine learning for effective high-performance computation," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2019, pp. 422–429.

[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Representations*, 2015, pp. 1–15.

[40] J. Appleyard, T. Kociskỳ, and P. Blunsom, "Optimizing performance of recurrent neural networks on GPUs. corr abs/1604.01946 (2016)," 2016, *arXiv:1604.01946*.

[41] G. Brockman *et al.*, "OpenAI Gym," 2016, *arXiv:1606.01540*.

[42] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.