μ-DDRL: A QoS-Aware Distributed Deep Reinforcement Learning Technique for Service Offloading in Fog computing Environments

Mohammad Goudarzi, Maria A. Rodriguez, Majid Sarvi, and Rajkumar Buyya

Abstract—Fog and Edge computing extend cloud services to the proximity of end users, allowing many Internet of Things (IoT) use cases, particularly latency-critical applications. Smart devices, such as traffic and surveillance cameras, often do not have sufficient resources to process computation-intensive and latency-critical services. Hence, the constituent parts of services can be offloaded to nearby Edge/Fog resources for processing and storage. However, making offloading decisions for complex services in highly stochastic and dynamic environments is an important, yet difficult task. Recently, Deep Reinforcement Learning (DRL) has been used in many complex service offloading problems; however, existing techniques are most suitable for centralized environments, and their convergence to the best-suitable solutions is slow. In addition, constituent parts of services often have predefined data dependencies and quality of service constraints, which further intensify the complexity of service offloading. To solve these issues, we propose a distributed DRL technique following the actor-critic architecture based on Asynchronous Proximal Policy Optimization (APPO) to achieve efficient and diverse distributed experience trajectory generation. Also, we employ PPO clipping and V-trace techniques for off-policy correction for faster convergence to the most suitable service offloading solutions. The results obtained demonstrate that our technique converges quickly, offers high scalability and adaptability, and outperforms its counterparts by improving the execution time of heterogeneous services.

Index Terms—Fog/Edge Computing, Internet of Things (IoT), Deep Reinforcement Learning (DRL), QoS-Aware Service Offloading.

1 Introduction

Internet of Things (IoT) devices have been widely adapted for many application scenarios, resulting in more intelligent and efficient solutions. The domain of IoT applications is diverse and ranges from smart transportation and mobility systems to healthcare [1], [2]. To illustrate, ongoing urbanization along with ever increasing traffic has led to a high demand for smart transportation and mobility systems, where the main goal is to obtain faster, cheaper, and safer transportation by collecting, augmenting, and analyzing heterogeneous data generated from various sources. Smart transportation and mobility systems comprise a wide range of services with heterogeneous characteristics, including but not limited to vehicle automation, dynamic traffic light control, and road condition monitoring [3], [4]. These growing IoT services, either computationally intensive or latency-critical, require high computing, storage, and communication resources for smooth and precise execution [5], [6]. On their own, IoT devices with limited computing and storage capacities cannot efficiently process and analyze the vast amount of generated data in a timely manner, and hence they require surrogate resources for processing and storage. We refer to the process of allocating services on surrogate resources as offloading/outsourcing [7], [8].

Cloud computing is fundamental for IoT service deployment, providing elastic computing and storage resources [4], [9]. However, it falls short for latency-critical services due to high latency and low bandwidth between IoT devices and cloud servers. In contrast, fog computing, using Fog Servers (FSs) near IoT devices, offers low-latency, high-bandwidth resources [10], [11]. FSs are resource-limited compared to Cloud Servers (CSs) and may not handle all services, especially computation-intensive ones. In our view, edge computing only leverages nearby resources, while fog computing uses a hierarchical structure of FSs and CSs to handle computation-intensive and latency-sensitive services. Fig. 1 shows an overview of Edge and fog computing.

Offloading services to fog computing facilitates practical deployment of computation-intensive and latency-critical services. However, this poses a challenging problem due to resource limitations in service providers and complex dependencies in service models, represented as Directed Acyclic Graphs (DAGs) [2], [5], [12]. Also, IoT services require specific Quality of Service (QoS) constraints for smooth execution [13]. Given the scale of IoT services, resources, and constraints, service offloading in fog computing becomes an NP-hard problem. Existing heuristics and rule-based algorithms struggle to adapt to the dynamic and stochastic nature of fog computing [2].

In the stochastic fog computing environment, adaptive service offloading decisions are essential. Deep Reinforcement Learning (DRL) combines Reinforcement Learning (RL) with Deep Neural Networks (DNN) to enable dynamic learning of optimal policies and rewards in stochastic

- *M. Goudarzi and R. Buyya are with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia.*
- *M. A. Rodriguez is with the School of Computing and Information Systems, University of Melbourne, Australia.*
- *M. Sarvi is with the Department of Infrastructure Engineering, University of Melbourne, Australia*
  E-mail: m.goudarzi@unsw.edu.au, marodriguez@unimelb.edu.au, majid.sarvi@unimelb.edu.au, rbuyya@unimelb.edu.au.

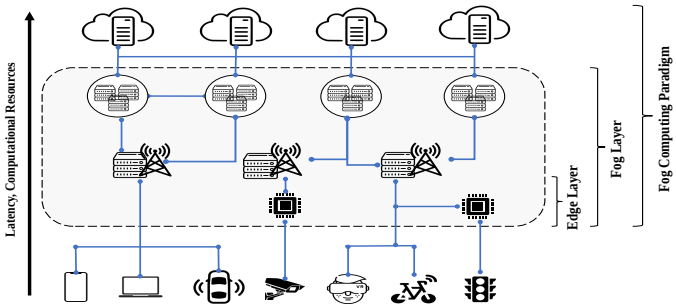


Figure 1: An overview of Edge and Fog computing

settings without prior system knowledge [14]. However, the DRL agent requires collecting a large and diverse set of experience trajectories in the exploration phase to be used for training the agent to learn the optimal policy. Thus, centralized DRL methods suffer from high exploration costs and slow convergence, affecting service offloading performance in heterogeneous and stochastic fog computing environments. Moreover, most of the distributed DRL techniques don't efficiently utilize experience trajectories from different actors, leading to slow convergence to optimal results [14].

To address these challenges, we propose μ-DDRL, a distributed deep reinforcement learning technique designed for efficient offloading of diverse services with varying QoS requirements in heterogeneous fog computing environments. μ-DDRL leverages Asynchronous Proximal Policy Optimization (APPO) [15], employing an actor-learner framework, in which deployed distributed agents across different Fog Servers (FSs) work in parallel. This approach reduces exploration costs, accelerates trajectory generation, diversifies trajectories, and enhances the learning of optimal service offloading policies. Moreover, μ-DDRL employs V-trace [16] and PPO Clipping [17] to address the policy gap between actors and the learner caused by decoupled acting and learning. The main contributions of this paper are summarized as follows.

- An execution time model for minimizing service offloading of DAG-based services while satisfying their required QoS is proposed.
- A distributed DRL-based service offloading technique, called μ-DDRL, for dynamic and stochastic fog computing environments is proposed, which works based on the APPO framework. It helps to efficiently use the experience trajectories generated by several distributed actors to train a better service offloading model. Consequently, we propose a reward function for μ-DDRL to minimize the execution time of each DAG-based service while satisfying QoS. We use two techniques, called V-trace and PPO Clipping, to solve the policy gap between actors and learners.
- We conduct comprehensive experiments using a wide range of synthetic service DAGs derived from real-world services to cover the requirements of different services. Also, we compare the performance of μ-DDRL with three state-of-the-art and widely adopted DRL and DDRL techniques.

The rest of the paper is organized as follows. Section 2 reviews related work on service offloading techniques in fog computing environments. Section 3 describes the system model and problem formulation. The DRL model and its main concepts are presented in Section 4. Our proposed DDRL-based service offloading framework is presented in Section 5. In Section 6, the performance of μ-DDRL is studied and compared with state-of-the-art techniques. Finally, Section 7 concludes the paper and draws future work.

2 Related Work

In this section, we discuss service offloading techniques in fog computing environments.

Several works in the literature consider an IoT service either as a single service or as a set of independent components. In these works, the constituent parts of each service can be executed without any data dependency among the different components. Bahreini et al. [18] proposed a greedy technique to manage the computing resources of connected electric vehicles to minimize energy consumption. Chen et al. [19] proposed a collaborative service placement technique in an Edge computing environment, in which a network of small cell base stations is deployed to minimize the execution cost of services. The problem is first solved in a centralized manner to find the optimal solution, and then a graph coloring technique is proposed to solve the service placement problem in a decentralized way. Maleki et al. [20] proposed location-aware offloading approaches based on approximation and greedy techniques to minimize the response time of applications in a stochastic and dynamic Edge computing environment. Badri et al. [21] modeled the location-aware offloading problem in an Edge computing environment as a multistage stochastic problem, and proposed a Sample Average Approximation (SAA) algorithm to maximize the QoS of users while considering the energy budget of Edge servers. Ouyang et al. [22] proposed a Markov approximation technique to obtain a near-optimal solution for the cost-performance trade-off of the service offloading in the Edge computing environment, which does not require prior knowledge of users' mobility. Zhang et al. [23] and Huang et al. [24] proposed a Deep Q-Network (DQN) based technique to learn the optimal migration policy and computation offloading in mobile Edge computing, respectively. Zhang et al. [23] considered scenarios where user mobility is high, such as autonomous driving, to improve user QoS, while Huang et al. [24] considered fixed IoT devices, such as smart cameras. Furthermore, Chen et al. [25] proposed the use of a more stable version of the DQN technique, Double DQN (DDQN), to learn the optimal offloading policy. Garaali et al. [26] proposed a multi-agent deep reinforcement learning solution based on the Asynchronous Advantage Actor Critic (A3C) to reduce system latency, so that each agent aims to learn interactively the best offloading policy independently of other agents. Li et al. [27] designed a hierarchical software-defined network architecture for vehicles that send their computational tasks to remote Edge servers for processing. Next, a DQN technique is proposed to solve the load balancing optimization problem so that the mean square deviation of the loads among different Edge servers is minimized. Liao et al. [28] developed a double RL-based offloading solution for mobile

devices to minimize the energy consumption of devices and the delay of tasks based on the Deep Deterministic Policy Gradient (DDPG) and DQN, respectively. Ramezani et al. [29] proposed a Q-learning-based technique to reduce load imbalance and waste of resources in fog computing environments. Zhou et al. [30] proposed a Mixed Integer Non-Linear Programming (MINLP) formulation to jointly optimize the computation offloading and service caching in fog computing environments. To solve the optimization problem, an A3C-based technique is designed to optimize the offloading decision and service caching.

Several works in the literature considered IoT services as a set of dependent tasks, modeled as DAG. In these services, each task can only be executed when its parent tasks finish their execution. Wang et al. [7] proposed a mobility-aware microservice coordination scheme to reduce the service delay of real-time IoT applications in the fog computing environment. First, the authors formulated the problem as an MDP and then proposed a Q-learning technique to learn the optimal policy. The authors in [31] proposed a mobility-aware application placement technique for microservices based on the Non-Dominated Sorting Genetic Algorithm (NSGA-II) in a fog computing environment to minimize the execution time, energy consumption, and cost. The authors in [32] proposed fast heuristics for the placement and migration of real-time IoT applications, consisting of microservices, in a hierarchical fog computing environment. Shekhar et al. [33] proposed a service placement technique for microservice-based IoT applications while considering a deterministic mobility pattern for users based on previous mobility data. The main goal of the centralized controller is to make the placement decision to satisfy the latency requirements. Qi et al. [2] proposed an adaptive and knowledge-driven service offloading technique based on DRL for microservice-based smart vehicles' applications in an Edge computing environment. Mouradian et al. [34] and Sami et al. [35] studied metaheuristic-based service placement in Edge and fog computing environments, respectively. Mouradian et al. [34] formulated the service placement problem as using Integer Linear Programming (ILP) and proposed a Tabu Search-based placement algorithm to minimize the response time and cost of applications. Sami et al. [35] proposed an evolutionary Memetic Algorithm (MA) for the placement of containerized microservices in vehicle onboard units. Wang et al. [14] proposed a centralized DRL technique based on Proximal Policy Optimization (PPO) for the placement of a diverse set of IoT applications to minimize their response time. Bansal et al. [36] proposed a dueling DQN technique to minimize the energy consumption of IoT devices and the response time of modular IoT applications in multi-access edge computing environments.

Table 1 identifies key elements identified in the literature: service structure, environmental architecture, and decision engine. This table compares our work with existing literature in these areas to emphasize our contributions. The service structure explores IoT service design models, including monolithic, independent tasks, modular, and microservices, and introduces granular heterogeneity in terms of computation size and data flow. The environmental architecture assesses tiering, IoT device and Edge/Fog server properties, potential cooperation among Fog Servers (FSs), and consideration of multiple cloud service providers. The decision engine evaluates optimization characteristics, problem modeling, QoS constraints, decision planning techniques, adaptability to changing computing environments, and high scalability in each technique.

Considering the literature, the most complex and heterogeneous fog computing environment comprises properties that include diverse IoT devices, services with varying QoS requirements (often with dependent constituent parts), heterogeneous FSs, multi CSs, and high dynamicity. Traditional approximation techniques, heuristics, and metaheuristics, such as [18], [19], [31] are inadequate for efficient offloading in such complex fog computing environments [2], [26], [30]. Moreover, the high cost of exploration and the low convergence rate of centralized DRL agents in the literature, such as [14], [23], [24], [25], [27], [36], form a barrier to the practical deployment of centralized DRL-based decision engines, especially when the number of features, the complexity of environments, and applied constraints to the offloading problem increase. There are some DDRL works in the literature, such as [2], [26], [30], that use parameter-sharing DDRL techniques. These techniques, such as [2], [26], [30], though distributed, are inefficient in utilizing experience trajectories from different actors because each actor trains its local policies based on its limited experience trajectories and then forwards these parameters to the learner for aggregation/training. Moreover, sharing parameters between actors and learners in these DDRL techniques is more costly since the size of parameters shared among actors and learners is larger than the weights of experience trajectories. To tackle these issues, we introduce μ-DDRL, an experience-sharing distributed deep reinforcement learning approach, which leverages an Asynchronous Proximal Policy Optimization (APPO) technique. It employs an actor-learner framework with multiple distributed actors working in parallel to generate diversified batches of experience trajectories, significantly reducing exploration costs, improving the convergance rate, and enhancing offloading efficiency. Additionally, μ-DDRL incorporates V-trace [16] and PPO Clipping [17] techniques to rectify discrepancies between the learner and actor policies, arising from the decoupled acting and learning process.

3 System Model and Problem Formulation

In our system model, we assume that each service may have a different number of tasks with various dependency models, which can be represented as directed cyclic graphs (DAGs) [2]. Moreover, some predefined servers (i.e., brokers) in proximity of IoT devices are responsible for making service placement decisions based on the resource requirements of each service, its QoS requirements, and the system's properties. These servers are accessible with low latency and high bandwidth, which helps reduce the start-up time of the services. An overview of our system model in fog computing is shown in Fig. 2[1]. Appendix A presents a summary of the parameters and definitions used.

[1]For a detailed description of protocols, technologies, and technical aspects, please refer to https://github.com/Cloudslab/FogBus2

Table 1: A qualitative comparison of related works with ours

Ref	Service Structure		Environmental Architecture						Multi Cloud	Decision Engine					
	Design	Granular Het	Tiering	IoT Device		Edge/Fog Servers				Optimization Characteristics			Decision Planner Technique	High Scalability	High Adaptability
				Number	Het	Number	Het	Cooperation		Perspective	Problem Modelling	QoS Constraints			
[18]	Monolithic	✓	Edge	Multiple	✓	Multiple	✓	×	×	IoT	MINLP	×	Greedy	×	×
[19]	Independent	✓	Edge	Multiple	ND	Multiple	ND	✓	×	IoT	IP	×	Optimal	×	×
[20]	Monolithic	×	Edge	Multiple	✓	Multiple	✓	×	×	IoT	MINLP	×	Approx	×	×
[7]	Microservice	✓	Fog	Multiple	✓	Multiple	✓	✓	×	IoT	MDP	×	Q-learning	×	✓
[31]	Microservice	✓	Fog	Multiple	✓	Multiple	✓	✓	×	Hybrid	ND	×	NSGA2	×	×
[21]	Monolithic	×	Edge	Multiple	✓	Multiple	✓	×	×	Hybrid	IP	Energy	SAA	×	×
[32]	Microservice	✓	Fog	Multiple	✓	Multiple	✓	✓	×	IoT	ILP	×	Heuristic	×	×
[22]	Monolithic	✓	Edge	Multiple	✓	Multiple	✓	✓	×	Hybrid	Lyapu	Deadline	Approx	×	×
[33]	Microservice	✓	Fog	Single	×	Multiple	✓	×	×	IoT	ND	×	GTB	×	×
[2]	Microservice	✓	Edge	Multiple	✓	Multiple	✓	✓	×	IoT	MDP	×	A3C	✓	✓
[23]	Monolithic	×	Edge	Multiple	×	Multiple	✓	×	×	IoT	ND	×	DQN	×	✓
[34]	Modular	✓	Fog	Multiple	✓	Multiple	✓	ND	✓	IoT	ILP	×	Tabu	×	×
[35]	Microservice	✓	Edge	Multiple	✓	Multiple	✓	✓	×	IoT	ND	×	MA	×	×
[24]	Monolithic	ND	Edge	Multiple	✓	Single	×	×	×	IoT	MDP	×	DQN	×	✓
[25]	Independent	✓	Edge	Single	✓	Multiple	×	×	×	IoT	MDP	×	DDQN	×	✓
[14]	Modular	✓	Edge	Multiple	✓	Single	×	×	×	IoT	MDP	×	PPO	×	✓
[26]	Independent	✓	Edge	Multiple	✓	Multiple	✓	×	×	IoT	MDP	×	A3C	✓	✓
[27]	Independent	✓	Edge	Multiple	✓	Multiple	✓	×	×	System	MDP	×	DQN	×	✓
[28]	Independent	✓	Edge	Multiple	✓	Single	×	×	×	IoT	MDP	×	DDPG-DQN	×	✓
[29]	independent	✓	Fog	Multiple	×	Multiple	ND	×	×	System	MDP	×	Q-learning	×	×
[36]	Modular	✓	Edge	Multiple	✓	Multiple	✓	✓	×	IoT	MDP	×	DuDQN	×	✓
[30]	Independent	✓	Fog	Multiple	✓	Multiple	✓	×	×	Hybrid	MDP	×	A3C	✓	✓
Our Technique	Microservice	✓	Fog	Multiple	✓	Multiple	✓	✓	✓	IoT	MDP	Deadline	APPO	✓	✓

Het: Heterogeneity, ND: Not Defined, MDP: Markov Decision Process, IP: Integer Programming, ILP: Integer Linear Programming, MINLP: Mixed Integer Non-Linear Programming, Lyapu: Lyapunov
SAA: Sample Average Approximation, GTB: Gradient Tree Boosting, PPO: Proximal Policy Optimization, MA: Memetic Algorithm, DDQN: Double DQN, DuDQN: Dueling DQN

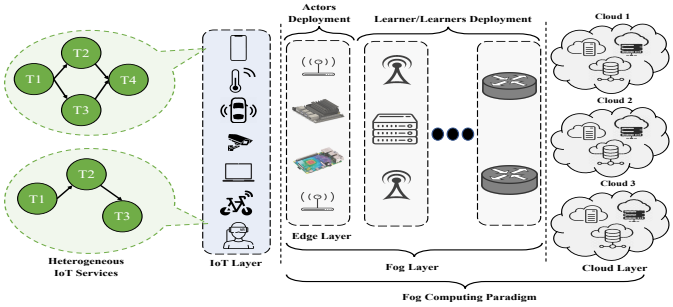


Figure 2: An overview of our system model

3.1 Service Model

We define each service as a DAG $G = (\mathcal{V}, \mathcal{E})$, in which $\mathcal{V} = \{v_i | 1 \leq i \leq |\mathcal{V}|\}, |\mathcal{V}| = L$ shows the set of tasks within each service, where the ith task is shown as v_i. The edges of the graph, $\mathcal{E} = \{e_{i,j} | v_i, v_j \in \mathcal{V}, i \neq j\}$, denote the data dependencies between tasks, with $e_{i,j}$ representing a data flow between v_i (parent) and v_j (child). Additionally, the weight of an edge, $e^w_{i,j}$, represents the amount of input data that task v_j receives from task v_i.

A task v_j is represented as a tuple $< v^w_j, v^{ram}_j, \zeta_{v_j} >$, where v^w_j is the number of CPU cycles required for the processing of the task, v^{ram}_j is the required amount of RAM required to run the task, and ζ_{v_j} is the maximum tolerable delay for the task (which restricts the execution time of each task). Considering the dependency model between tasks, $\mathcal{P}(v_j)$ is defined as the set of all predecessor tasks of v_j. For each DAG G, tasks without parent tasks are called input tasks, while tasks without children are called exit tasks.

3.2 Problem Formulation

The set of available servers is defined as $\mathcal{M}$, where $|\mathcal{M}| = M$. We represent each server as $m^{y,z} \in \mathcal{M}$, in which y illustrates the server's type (i.e., IoT device, FSs, CSs) and z presents the index of the corresponding server's type. Therefore, we define the offloading configuration of task v_j as follows:

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

Accordingly, for each service with L tasks, the offloading configuration $\mathcal{X}$ can be defined as:

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

Tasks within a service can be sorted in a sequence by an upward ranking algorithm that guarantees each task can only be executed after the successful execution of its predecessor tasks [5]. Also, the upward ranking algorithm defines a priority for tasks that can be executed in parallel and, accordingly, sorts them [14], [32]. For parallel tasks, we define $CP(v_i)$ as a function to indicate whether or not each task is on the critical path of the service based on the upward ranking function and the associated computational cost of running tasks [2], [14] (i.e., a set of tasks and the corresponding data flow resulting in the highest execution time). Also, $\mathcal{CP}$ presents the set of tasks on the critical path of the service.

3.2.1 *Optimization model*

For each task v_j, the execution time can be defined as the time it takes for the input data to become available for that task $\mathcal{T}^{input}_{x_{v_j}}$ plus the processing time of the task on the corresponding server $\mathcal{T}^{proc}_{x_{v_j}}$:

$$\mathscr{T}_{x_{v_j}} = \mathscr{T}^{proc}_{x_{v_j}} + \mathscr{T}^{input}_{x_{v_j}} \tag{3}$$

where $\mathscr{T}^{proc}_{x_{v_j}}$ can be obtained based on the CPU cycles required by the task (v^w_j) and the processing speed of the corresponding assigned server $f^s_{x_{v_j}}$:

$$\mathscr{T}^{proc}_{x_{v_j}} = \frac{v^w_j}{f^s_{x_{v_j}}} \tag{4}$$

$\mathscr{T}^{input}_{x_{v_j}}$ is estimated as the time it takes for all input data to arrive to the server assigned to v_j (i.e., x_{v_j}) from its predecessors:

$$\mathscr{T}^{input}_{x_{v_j}} = \max((\frac{e^w_{i,j}}{b(x_{v_i}, x_{v_j})} + l(x_{v_i}, x_{v_j})) \times SS(x_{v_i}, x_{v_j})),\ \forall v_i \in \mathcal{P}(v_j) \tag{5}$$

in which $b(x_{v_i}, x_{v_j})$ illustrates the data rate (i.e., bandwidth) between the selected servers for the execution of v_i and v_j, respectively. Moreover, $l(x_{v_i}, x_{v_j})$ depicts the communication latency between two servers (i.e., an IoT device and a remote server or two remote servers), and is calculated based on the propagation speed for the communication medium (i.e., $\eth^c$) and the Euclidean distance between the coordinates of the participating servers (i.e., $d(x_{v_i}, x_{v_j})$) in the Cartesian coordinate system:

$$l(x_{v_i}, x_{v_j}) = \frac{d(x_{v_i}, x_{v_j})}{\eth^c} \tag{6}$$

where $d(x_{v_i}, x_{v_j})$ is calculated as follows:

$$d(x_{v_i}, x_{v_j}) = \sqrt{(x^x_{v_i} - x^x_{v_j})^2 + (x^y_{v_i} - x^y_{v_j})^2} \tag{7}$$

The $SS(x_{v_i}, x_{v_j})$ is a function that indicates whether the assigned servers to each pair of tasks are the same (i.e., 0 if $x_{v_i} = x_{v_j}$) or different (i.e., 1 if $x_{v_i} \neq x_{v_j}$). Because the fog computing environment is dynamic, heterogeneous, and stochastic, $f^s_{x_{v_j}}$, $b(x_{v_i}, x_{v_j})$, and $l(x_{v_i}, x_{v_j})$ may have different values from time to time.

The principal optimization objective is minimizing the execution time of each service by finding the best possible configuration of surrogate servers for the execution of the service's tasks, as defined below.

$$\min(\mathcal{T}(\mathcal{X})) \tag{8}$$

where

$$\mathcal{T}(\mathcal{X}) = \sum_{j=1}^{L} CP(v_j) \times \mathscr{T}_{x_{v_j}} \tag{9}$$

s.t.

$$CS1:\ S_n(x_{v_j}) = 1,\ \forall x_{v_j} \in \mathcal{X} \tag{10}$$

$$CS2:\ \mathcal{T}(v_i) \leq \mathcal{T}(v_i + v_j),\ \forall v_i \in \mathcal{P}(v_j) \tag{11}$$

$$CS3:\ \sum_{\forall v_j \in \mathcal{V}} IA(v_j, m^{y,z}) \times v^{ram}_j \leq R(m^{y,z}) \tag{12}$$

$$CS4:\ \mathscr{T}_{x_{v_j}} \leq \zeta_{v_j},\ \forall v_j \in \mathcal{V} \tag{13}$$

where $CP(v_j)$ can be obtained from:

$$CP(v_j) = \begin{cases} 1, & v_j \in \mathcal{CP} \\ 0, & otherwise \end{cases} \tag{14}$$

$CP(v_j)$ is equal to 1 if the task v_j is on the critical path and 0 otherwise. $CS1$ restricts the assignment of each task to exactly one server at a time. Also, $CS2$ shows that the cumulative execution time of v_j is always larger or equal to the execution time of its predecessors' tasks [37], which guarantees the execution order of tasks with data dependency. $CS3$ states that the summation of the required RAM for all tasks assigned to one server should always be less than or equal to the available memory on that server $R(m^{y,z})$. $IA(v_j, m^{y,z})$ is an indicator function to check if the task v_j is assigned to the server $m^{y,z}$ ($IA = 1$) or not ($IA = 0$). Finally, $CS4$ states that the execution time of each task should be within the range of the task's tolerable delay.

The service offloading problem in heterogeneous computing environments is an NP-hard problem [5], [38], and the complexity of the problem grows exponentially as the number of servers and/or tasks within a service increases. Therefore, the optimal solution to the service offloading problem cannot be obtained in polynomial time.

4 DRL Model

In the DRL, reinforcement learning and deep learning are combined to transform a set of high-dimensional inputs into a set of outputs to make decisions. In DRL, the learning problem is formally modeled as a Markov Decision Process (MDP) which is mainly used to make decisions in stochastic environments. We define a learning problem by a tuple $< \mathbb{S}, \mathbb{A}, \mathbb{P}, \mathbb{R}, \gamma >$, where $\mathbb{S}$ shows the state space and $\mathbb{A}$ presents the action space. The probability of a state transition between states is shown by $\mathbb{P}$. Finally, $\mathbb{R}$ and $\gamma \in [0, 1]$ denote the reward function and discount factor, respectively. Also, the continuous time horizon is divided into multiple time steps $t \in \mathbb{T}$. In each time step t, the DRL agent interacts with the environment and receives the state of the environment s_t. Consequently, the agent uses its policy $\pi(a_t|s_t)$ and chooses an action a_t. The agent performs the action a_t and receives a reward in that time step r_t. The agent aims at maximizing the expected total of future discounted rewards:

$$\mathbb{V}^{\pi}(s_t) = \mathbb{E}_{\pi}[\sum_{t \in \mathbb{T}} \gamma^t r_t] \tag{15}$$

in which $r_t = \mathbb{R}(s_t, a_t)$ represents the reward at time step t, and $a_t \sim \pi(.|s_t)$ is the action at t when following the policy π. Furthermore, because DNN is used to approximate the function, θ shows the corresponding parameters.

In what follows, the main concepts of the DRL for the service offloading problem in the fog computing environment are described:

- **State space $\mathbb{S}$:** The state is defined as the agent's observation of fog computing environments. Accordingly, we represent the state space as a set of characteristics for all servers, IoT devices, and corresponding services in different time steps:

$$\mathbb{S} = \{s_t | s_t = (F^{\mathcal{M}}_t, F^{v_j}_t), \forall t \in \mathbb{T}\} \tag{16}$$

where s_t shows the system state at time step t, $F^{\mathcal{M}}_t$ represents the feature vector of all M servers at time step t, and $F^{v_j}_t$ presents the feature vectors of the current task in a service. The $F^{\mathcal{M}}_t$ includes the servers and devices' properties, such as their positions, number of CPU cores, corresponding CPU frequency speed, data

rate, ram, etc. If each server/device has a maximum of K_1 features, we can define the feature vector of all M servers/devices at time step t as:

$$F_t^{\mathcal{M}} = \{f_i^{m^{y,z}} | \forall m^{y,z} \in \mathcal{M}, 1 \leq \ i \leq k_1\} \tag{17}$$

in which $f_i^{m^{y,z}}$ represents the ith feature corresponding to the server $m^{y,z}$. Also, the $F_t^{v_j}$ includes the features of a current task , v_j, such as required computation, RAM, dependency model of the corresponding tasks, and service offloading configuration of prior tasks, just to mention a few. As tasks within a service are sorted by upward ranking, the current tasks' dependencies are already solved. Supposing that each task at maximum contains k_2 features, we can define the feature vector of task v_j at time step t as:

$$F_t^{v_j} = \{f_i^{v_j} | v_j \in \mathcal{V}, \forall i \ 1 \leq i \leq k_2\} \tag{18}$$

in which $f_i^{v_j}$ shows the ith feature of the task v_j.

- **Action space $\mathbb{A}$:** Since we aim to find the best configuration of servers for tasks within a service, the action space $\mathbb{A}$ is related to all available servers, defined in what follows:

$$\mathbb{A} = \mathcal{M} \tag{19}$$

Also, each action in time step t, i.e., a_t, is defined in the assignment of a server to the current task:

$$a_t = x_{v_j} = m^{y,z} \tag{20}$$

- **Reward function $\mathbb{R}$:** As presented in Eq. 8, the optimization goal is to minimize the execution time of each service while satisfying the QoS. Hence, the reward function per time step t is defined as the negative value of Eq. 3 if the task can be properly executed within the maximum tolerable delay (i.e., $\mathscr{T}_{x_{v_j}} \leq \zeta_{v_j}$). As tasks are currently sorted based on the upward-ranking function, tasks that incur the highest execution cost (i.e., tasks on the critical path of the service) receive higher priority. Thus, tasks incurring the highest execution cost can be assigned to more powerful servers at the initial stages of the offloading decision-making process. Although it is not obligatory for the step-reward function to directly align with the long-term reward for the DRL agent, enhancing their correlation can enhance the agent's overall performance. In our reward function, the step reward is correlated to the long-term reward because it tries to reduce the execution time of each task, ultimately resulting in the reduction of the overall service execution time. Moreover, Φ is defined as a failure penalty to penalize actions that violate the maximum tolerable delay for that task for any reason. Accordingly, r_t is defined as:

$$r_t = \begin{cases} -\mathscr{T}_{x_{v_j}}, & \mathscr{T}_{x_{v_j}} \leq \zeta_{v_j} \\ \Phi, & otherwise \end{cases} \tag{21}$$

5 μ-DDRL: Distributed DRL Framework

In this section, we mainly describe the μ-DDRL framework, which is a technique for high-throughput distributed deep reinforcement learning based on the actor-critic APPO architecture. μ-DDRL is designed for QoS-aware service offloading and aims to minimize the execution time of distinct services in highly heterogeneous and stochastic computing environments.

In the actor-critic architecture, the policy, $\pi(a_t|s_t;\theta)$ is parameterized by θ. 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., $\mathbb{V}^{\pi}(s_t)$) are used to update θ. The actor interacts with its environment and obtains the state s_t and selects an action a_t according to the parameterized policy $\pi(a_t|s_t;\theta)$, resulting in a reward r_t and the next state s_{t+1}. The critic utilizes the obtained rewards to evaluate the current policy based on the Temporal Difference (TD) error between the current reward and the estimation of the value function. Different DNNs are used as function approximators for the actor and the critic, which are trained independently. The parameters of the actor network are updated by feedback from the TD error to improve the selection probability of suitable actions. Additionally, the parameters of the critic network are updated to obtain a higher quality value estimation. The decoupled act and learning in actor-critic architecture makes this method a suitable option for long-term performance optimizations in distributed computing environments.

APPO and its role in μ-DDRL: While the actor-critic architecture is suitable for highly distributed environments, it still suffers from slow learning speed and high exploration costs. APPO is a distributed DRL architecture that helps significantly parallelize the collection of the distributed experience trajectories from different actors and achieves very high throughput. The distributed experience-sharing approach, in which actors generate heterogeneous trajectories in parallel and share their experience trajectories with learners/learners, can significantly improve the learning speed and exploration costs of actor-critic frameworks (and DRL frameworks in general). μ-DDRL works based on the decoupled actor-critic and APPO. Hence, each actor in the distributed fog computing environments makes service offloading decisions and creates experience trajectories. Each actor asynchronously forwards its collection of experience trajectories to the learner for training. Hence, the experience trajectories of slow actors (actors generating fewer experience trajectories) and fast actors (actors generating more experience trajectories) can be mixed together in the learner, resulting in more diversified experience trajectories. Moreover, when the learner updates the policy, the local policy of slow and fast actors will be updated together, so that each actor can more efficiently make service offloading decisions. Fig. 3 presents an overview of the μ-DDRL framework. In what follows, the roles of each actor and critic (i.e., learner) are described in detail.

5.1 μ-DDRL: Actor Role

In the μ-DDRL framework, actors act as brokers in the environment that make offloading decisions using their own local policy κ for incoming service offloading requests from smart devices (e.g., smart transportation cameras). In a fog computing environment, where computational resources

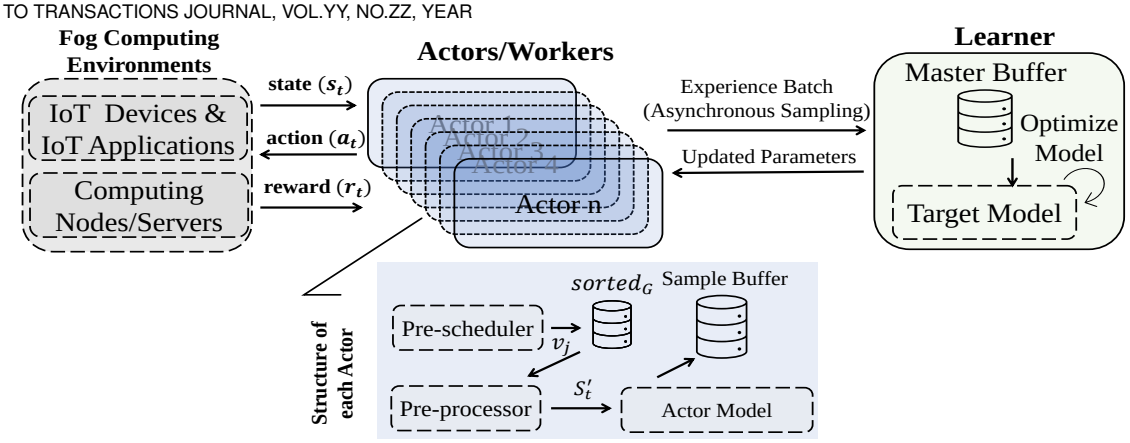


Figure 3: An overview of μ-DDRL framework

are distributed, actors can be deployed on distributed and heterogeneous servers in the environment. Moreover, distributed actors can work in parallel and interact with the environment to make service-offloading decisions. This distributed behavior reduces the incoming requests' queuing time, which improves the service-ready time for users. Algorithm 1 presents how each actor/broker interacts with the environment, makes service offloading decisions, and builds experience trajectories.

Initially, the local policy of each actor κ is updated by the policy of the learner π, to ensure that each actor works with the most optimized and updated policy to make service offloading decisions (line 3). Then, each actor uses the local policy κ to make N offloading decisions. The actor fetches the metadata of a service request from S_Q whenever it begins making the offloading decision for a new service G (line 7). Next, the metadata of the new service G is fed into the *Pre-sch()* function, which sorts the tasks within each service while satisfying the dependency constraints among tasks considering the DAG of each service and outputs the list of sorted tasks, $sort_G$ (line 8). The *Pre-sch()* function works based on the upward ranking to sort tasks within a service [5], [14]. While other sorting algorithms, such as topological sort, can also satisfy the dependency constraints of tasks within a DAG, these sorting algorithms, by default, do not provide any order for tasks that can be executed in parallel. The upward ranking algorithm not only defines an order for the parallel tasks but also helps identify the critical path of a DAG-based service by estimating the average execution cost of each task and its related edges on different servers. Next, the DAG of service G, list of available servers $\mathcal{M}$, and sorted lists of tasks $sort_G$ are given as input to the *RCVInitState()* function to build the system state s_i (line 9). Afterwards, the state of $flag_{in}$ is changed to $False$ until the actor starts making a decision for a new incoming service requiring the re-execution of *Pre-sch()* to obtain a new sequence of ordered tasks (line 10). Otherwise, the current state s_i of the system is obtained by *RCVCurrState()* function based on Eq. 16, which contains the feature vectors of the servers $F_t^{\mathcal{M}}$ and the current task of service $F_t^{v_j}$ (line 12). The *Pre-proc()* function receives the feature vector of the current state s_i and normalizes the values (line 14). Next, the actor performs the service offloading based on the current local policy κ and outputs the action a_i using the *SOEngine()* function (line 15). The action a_i is then executed, which means that the current task is offloaded onto the assigned surrogate server. The reward of this offloading action is then calculated by the *StepCostCal()* function based on Eq. 21, which takes into account the execution of the task on the assigned server within the specified maximum tolerable delay ζ_{v_j} (line 16). The *BuildNextState()* function then prepares the next state of the environment, and the experience batch EB of the current actor is updated by the experience tuple (s_i,a_i,r_i,s_{i+1}) (lines 17-18). If the service offloading for the current service is finished, the actor calculates the total execution cost of the current service using *TotalCostCal()* function based on Eq. 8 and adjusts the state of $flag_{in}$ (lines 19-20). This process is continuously repeated for the N steps, after which the actor sends the experience batch to the learner and restarts the process with an updated local policy κ (line 24).

Algorithm 1: Each actor's role

```
   Input  : π: The policy of the learner
   /* N:Maximum steps's number, κ: the local
      policy of each actor, EB: experience
      batch, S_Q: Queue of received requests, G:
      current service                          */
1  flag_in=True
2  while True do
3      κ=UpdateLocalPolicy(κ, π)
4      i = 0
5      while i < N do
6          if flag_in=True then
7              G=S_Q.dequeue()
8              sort_G = Pre-sch (G)
9              s_i=RCVInitState(G, M, sort_G)
10             flag_in=False
11         else
12             s_i=RCVCurrState()
13         end
14         s_i=Pre-proc(s_i)
15         a_i=SOEngine(s_i, κ) % Service Offloading Engine
16         r_i=StepCostCal(s_i, a_i) % → Eq. 21
17         s_{i+1} = BuildNextState(s_i, a_i)
18         EB.update(s_i, a_i, r_i, s_{i+1})
19         if Finish(G) then
20             TotalCostCal(G) % → Eq. 8
21             flag_in=True
22         end
23     end
24     ShareExpeienceToLearner(EB)
25 end
```

5.2 μ-DDRL: Learner Role

In the μ-DDRL framework, the main roles of the learner are managing the incoming experience batches from different actors, training and updating the target policy π, and updating the actor policies κ.

V-trace, PPO Clipping, and their role in μ-DDRL: In asynchronous RL techniques, a policy gap between the actor policy κ and learner policy π may happen. Generally, two major types of techniques are designed to cope with off-policy learning [15]. First, applying trust region methods by which the technique can stay closer to the behavior policy (i.e., the policy used by actors when generating experiences) during the learning process, which results in higher quality gradient estimates obtained using the samples from this policy [17]. The second type is employing importance sampling to correct targets for the value function to improve the approximation of the discounted sum of the rewards under the learner policy. In this work, we use PPO Clipping [17] as a trust region technique and the V-trace algorithm [16] as an importance sampling technique, which uses truncated importance sampling weights to correct the value targets. It has been shown that these techniques can be applied independently since V-trace corrects the training objective and PPO Clipping protects against destructive parameter updates. Overall, the combination of PPO-Clipping and V-trace leads to more stable training in asynchronous RL [15].

Algorithm 2 presents the learner role in μ-DDRL. The learner actively interacts with the actors and asynchronously receives the experience trajectories EB_a. The learner has a master buffer, called MB, which stores the incoming trajectories from actors (line 5). Whenever the size of the master buffer MB reaches the size of the training batch TB, the *BuildTrainBatch()* function fetches sufficient samples from the master buffer MB for training (lines 6-10). Next, the *OptimizeModel()* function computes the policy, computes value network gradients, and updates the policy and value network weights, as follows [39]:

$$\nabla_\theta J(\theta) = \frac{1}{|TB|} \sum_{j \in TB} \min(\mathbb{Z} \times \hat{A}_{\mathbb{V}(\text{GAE})}, Clip(Z, \epsilon) \hat{A}_{\mathbb{V}(\text{GAE})}) \quad (22)$$

where $\mathbb{Z}$ is $\frac{\pi_t}{\kappa}$, $Clip(.,\epsilon) = Clip(.,1-\epsilon,1+\epsilon)$ is a clipping function, TB shows the training batch, π_t shows the target policy, and the V-trace GAE-λ (i.e., $\hat{A}_{\mathbb{V}(\text{GAE})}$) modifies the advantage function by adding clipped importance sampling terms to the summation of TD errors:

$$\hat{A}_{\mathbb{V}(\text{GAE})} = \sum_{i=t}^{t+n-1} (\lambda\gamma)^{i-t} \left(\prod_{j=t}^{i-1} c_j \right) \delta_i \mathbb{V} \quad (23)$$

where $\delta_i \mathbb{V}$ is the importance sampled TD error introduced in V-trace [16], defined as follows:

$$\delta_i \mathbb{V} = \rho_i (r_i + \gamma \mathbb{V}(s_{i+1}) - \mathbb{V}(s_i)) \quad (24)$$

The γ is a discount factor. Moreover, $c_j = min(\overline{c}, \frac{\pi_t(a_j|s_j)}{\kappa(a_j|s_j)})$ and $\rho_i = min(\overline{\rho}, \frac{\pi(a_i|s_i)}{\kappa(a_i|s_i)})$ are clipped Importance Sampling (IS) weight in V-trace, controlling the speed of convergence and the value function to which the technique converges, respectively. Finally, the value network gradients are computed as follows.

Algorithm 2: Learner's role

```
Input  : EB_a: Actors' batch of experience
/* list_a: actors' list, π: the policy of the
   learner, MB: master buffer, TB: training
   batch                                        */
1  while True do
2      flag_tr=False
3      MB=∅
4      while flag_tr==False do
5          MB.add(EB_a)
6          if size(TB) ≤ size(MB) then
7              TB=BuildTrainBatch(MB)
8              flag_tr==True
9          end
10     end
11     OptimizeModel(TB) % → Eqs. 22, 23, 24, 25
12     UpdateActors(list_brokers)
13 end
```

$$\nabla_w L(w) = \frac{1}{|TB|} \sum_j \left(\mathbb{V}_w(s_j) - \hat{\mathbb{V}}_{\mathbb{V}(\text{GAE})}(s_j) \right) \nabla_w \mathbb{V}_w(s_j) \quad (25)$$

Next, the policy and value network weights are updated. Finally, the learner sends the updated policy and weights to the actors (line 12). Taking into account the target network stored in the learner, the *OptimizeModel()* function has the ability to run multiple stochastic gradient descent/Ascent (SGD/SGA). Moreover, μ-DDRL offers a highly scalable solution because as the number of actors increases, the number of shared experiences with the learner increases, and also the training batch is further diversified. As a result, distributed DRL agents can learn faster and adapt themselves to stochastic fog computing environments. In addition, the asynchronous nature of acting and learning in μ-DDRL helps newly joined actors in the environment to initialize their policy and parameters with the most recent and optimized policy of the learner rather than start making service offloading decisions using the naive actor policy, resulting in better offloading decisions.

6 Performance Evaluation

In this section, we describe the evaluation setup and baseline techniques used for the performance evaluation. Then, the hyperparameters of μ-DDRL are illustrated and the performance of μ-DDRL and its counterparts are evaluated.

6.1 Evaluation Setup

OpenAI gym [40] is used to develop a simulation environment for a heterogeneous fog computing environment. The environment comprises Smart Cameras in Transportation Systems (SCTS) as IoT devices with heterogeneous service requests, heterogeneous resource-limited FSs, and powerful CSs. For SCTS' resources, we consider a 1 GHz single-core CPU. As heterogeneous FSs, we consider 30 FSs, where each FS has a CPU with four cores with 1.5-2GHz processing speed and 1-4GB of RAM[2]. Moreover, we consider 20 CSs, where each CS has an eight-core CPU with 2-3Ghz processing speed and 16-24GB of RAM for CSs.

[2]The resources of FSs are aligned with resources of Raspberrypi and Nvidia Jetson platform

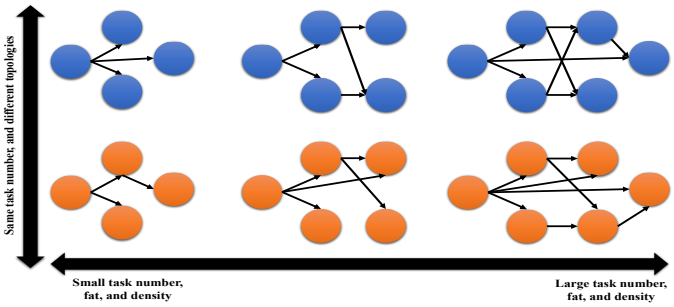


Figure 4: Role of density, fat, and number of tasks in dataset

The bandwidth (i.e., data rate) among different servers and SCTS devices is set based on the values profiled from the testbed setup [9]. Accordingly, the bandwidth between SCTS devices and FSs is randomly selected between 10-12MB/s, while the bandwidth between SCTS devices and FSs to CSs is randomly selected between 4-8MB/s, similar to [5], [41]. The latency between servers and SCTS devices depends on the respective coordinates of the SCTS devices and remote servers, as defined in Eq. 6.

Several complex services based on DAG for SCTS devices, which contain heterogeneous tasks with different dependency models, are designed to represent heterogeneous services with different preferences, similar to [2], [14]. The service property data contains the number of tasks, the required CPU cycles (v_j^w), RAM usage per task (v_j^{ram}), the dependency model between tasks, the amount of data transmission between a pair of tasks ($e_{i,j}^w$) and the maximum tolerable delay for each task (ζ_{v_j}). Thus, in order to generate heterogeneous DAGs, two important points must be considered. First, the topology of the DAGs depends on the number of tasks within the service (L), fat (identifies the width and height of the DAG), and density (controls the number of edges within the DAG). The second point is the weights assigned to each task and edge within each DAG. To generate different DAG topologies, we assume that the number of tasks for different services ranges from 5 to 50 tasks (i.e., $L \in \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$) to show a heterogeneous number of tasks for services [2], [5], [14]. Moreover, $fat \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$, and $density \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$ to create different dependency models between tasks of each service. Taking into account these values, we have generated 250 different topologies for DAGs, so that each DAG presents the potential topology of a service. Fig. 4 shows the role of fat, density, and L in the creation of DAGs. Next, for each DAG topology, we generate 100 DAGs with different sets of weights, where v_j^w ranges from 10^7 cycles to $3 * 10^8$ cycles [14], [42], v_j^{ram} ranges from 25MB to 100MB [43], $e_{i,j}^w$ ranges from 50KB to 2000KB [14], [44], and ζ_{v_j} is randomly selected from $[25-100]$ms [44], [45], [46], [47]. Consequently, 250 different DAG topologies and 100 different weighting configurations per DAG topology result in 25,000 service DAGs with heterogeneous L, fat, and $density$, and weights. Among these DAGs, 80% have been used for training μ-DDRL and baseline techniques, while 20% of the DAGs are used for evaluation.

The performance of μ-DDRL is compared with the following baseline techniques:

Table 2: Hyperparameters

Parameter	Value	Parameter	Value
FC layers	2	Learning Rate lr	0.01
Gradient Steps	2	Discount Factor γ	0.99
Optimization Technique	Adam	V-trace $\overline{\rho}$	1
Activation Function	Tanh	V-trace $\overline{c}$	1
Clipping Constant ϵ	0.2	GAE Discount Factor λ	0.95

- MRLCO: The improved version of the technique proposed in [14] is considered for evaluation. This technique is extended to be used in heterogeneous fog computing environments where several IoT devices, FSs, and CSs are available. Also, the reward function is updated to support deadline constraints. This technique uses synchronous PPO as its DRL framework while the networks of the agent are wrapped by the RNN. Furthermore, the agent is tuned based on the hyperparameters used in [14].
- Double-DQN: In the literature, several works such as [23], [24], [25] have used standard deep Q-learning (DQN) or double-DQN as the agent. Here, we used the optimized Double-DQN technique with an adaptive exploration which provides better efficiency for the agent. The agent model is tuned based on the hyperparameters used in [23].
- DRLCO: The enhanced version of the method introduced in [30] is considered for evaluation. Furthermore, we updated the reward function in this work to accommodate heterogeneous services with different topologies and deadline constraints. This approach uses A3C as its underlying DDRL framework.

6.2 Performance Study

This section describes the μ-DDRL hyperparameters and presents the performance of our proposed technique and its counterparts.

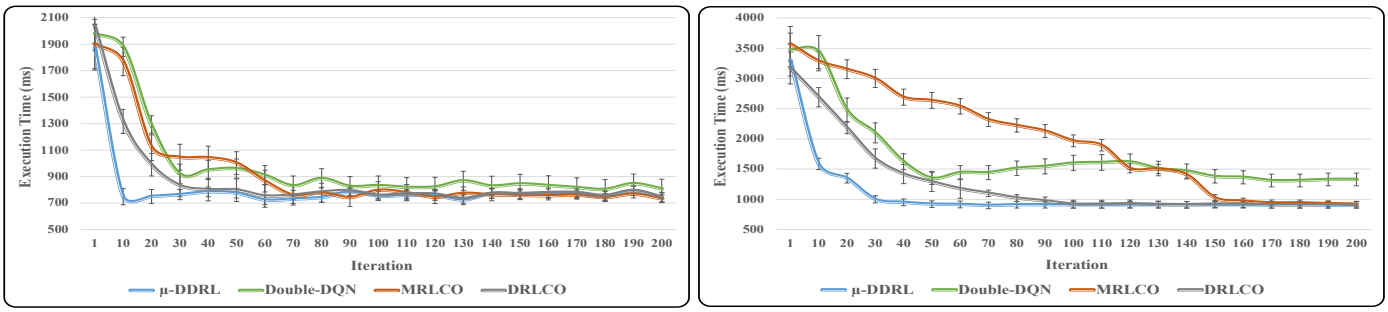
6.2.1 μ-DDRL Hyperparameters

In μ-DDRL, the DNN contains two fully connected layers, and the DNN structure of all agents is exactly the same. For hyperparameter tuning, we performed a grid search and, accordingly, the learning rate lr, discount factor γ, and gradient steps are set to 0.01, 0.99, and 2, respectively. The control parameters of the V-trace, $\overline{\rho}$ and $\overline{c}$, are set to 1. In addition, the clipping constant ϵ and the GAE discount factor are set to 0.2 and 0.95, respectively. The summary of the hyperparameter configuration is presented in Table 2.

6.2.2 Execution Time Analysis

6.2.3 System Size Analysis

This experiment demonstrates the performance of different service offloading techniques after specified target policy updates (i.e., training iteration) for different services. At first, the training dataset is selected as $L \in \{5, 10, 20, 25, 30, 35, 40, 45, 50\}$, while for the evaluation $L = 15$. Next, we consider the training dataset as $L \in \{5, 10, 15, 20, 25, 30, 35, 45, 50\}$ while for the evaluation $L = 40$. Hence, in both cases, the evaluation dataset is different from the training dataset, which helps understand



(a) Training dataset: $L \in \{5, 10, 20, 25, 30, 35, 40, 45, 50\}$, Evaluation dataset: $L = 15$

(b) Training dataset: $L \in \{5, 10, 15, 20, 25, 30, 35, 45, 50\}$, Evaluation dataset: $L = 40$

Figure 5: Execution time analysis vs number of target policy updates

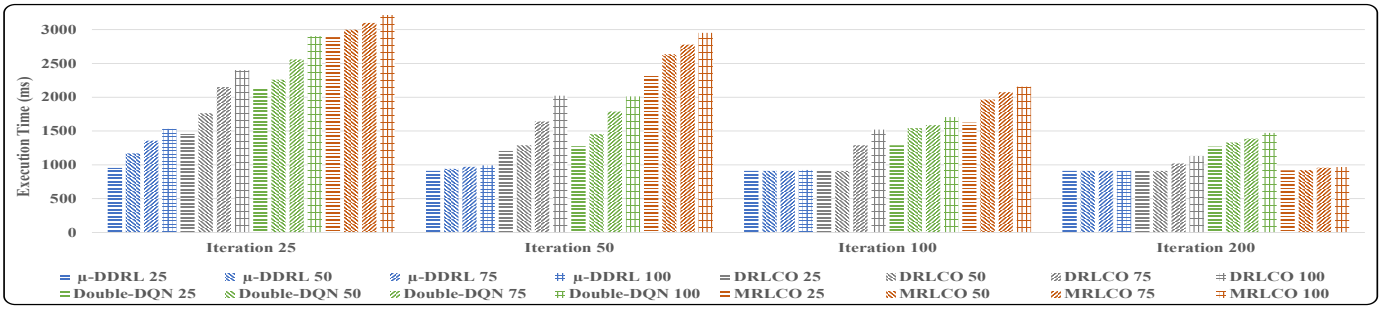


Figure 6: System size analysis

the performance of different techniques when receiving new service offloading requests.

Fig 5 represents the average execution time of all techniques in this study. As the number of iterations increases, the execution time of all techniques decreases, meaning that target policy updates lead to better service offloading decisions. However, the convergence speed of service offloading techniques is clearly different. μ-DDRL converges faster to the optimal solution compared to MRLCO, DRLCO, and Double DQN in both cases, as shown in Fig 5a and Fig 5b. The V-trace and clipping techniques used in μ-DDRL help for more efficient training on experience trajectories, leading to faster convergence to optimal service offloading solutions. Besides, the learner in μ-DDRL gets the benefit of more diverse experience trajectories in each training iteration on the learner compared to other techniques because it employs experience trajectories generated by several distributed actors. Also, comparing Fig. Fig 5a and Fig 5b demonstrates that the complexity of services has a direct impact on the performance of service offloading techniques, where suitable service offloading solutions for more complex services require more experience trajectories and policy updates. In addition, DRLCO reaches better service offloading solutions compared to Double DQN and MRLCO but its convergence speed is slower than μ-DDRL. Accordingly, the superiority of μ-DDRL is obtaining the best solution and also converging faster towards best solution. This experiment represents the performance of service offloading techniques when the number of surrogate computing servers increases. The larger number of servers leads to increased search space and the number of features that directly affect the complexity of the service offloading problem. In this experiment, we consider a different number of servers, where $M \in 25, 50, 75, 100$. Moreover, we consider the training dataset as $L \in \{5, 10, 15, 20, 25, 30, 35, 45, 50\}$, while for the evaluation, we use $L = 40$.

Fig. 6 presents the obtained execution time of all service offloading techniques while considering different number of servers after different target policy updates (i.e., training iterations). The execution time obtained from each technique decreases as the number of training iterations increases, which exactly follows the pattern shown in the previous experiment. Moreover, as the number of servers increases, the performance of service offloading techniques within the same iteration number is diminished. The background reason is that when the search space and the number of features increase, the DRL-based service offloading techniques require more interactions with the environment, diverse experience trajectories, and training to precisely obtain the offloading solutions. Although the pattern is the same for all techniques, μ-DDRL converges faster to more suitable service offloading solutions and outperforms other techniques even when the system size grows. This is mainly because μ-DDRL uses the shared experience trajectories (i.e. steps in the environment) of different actors in each training iteration, which are usually more diverse compared to experience trajectories of non-DDRL counterparts. Moreover, μ-DDRL training is more efficient than DRLCO because it works on actual experience trajectories forwarded from different actors, while the main policy in DRLCO is trained based on the parameters forwarded from different actors. Thus, μ-DDRL offers higher scalability and adaptability features.

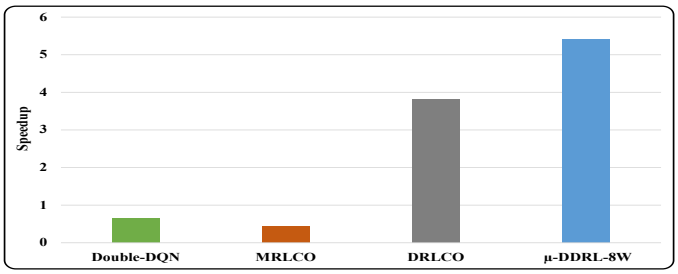


Figure 7: Speedup Analysis

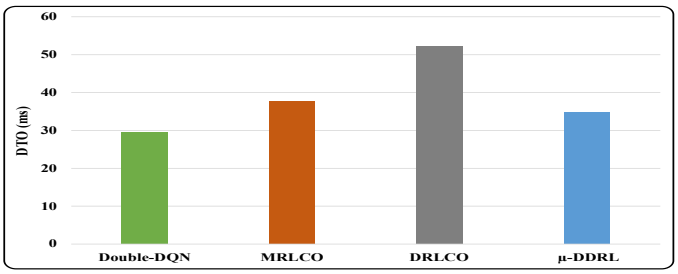


Figure 8: Decision Time Overhead Analysis

6.2.4 Speedup Analysis

The speedup analysis studies how long it takes for DRL agents to obtain a predefined number of experience trajectories. The faster interactions with the computing environment, the more experience trajectories, and the higher potential for faster training. Similarly to [5], [48], we define the speedup as follows:

$$SP = \frac{Time_R}{Time_T} \tag{26}$$

where $Time_R$ is the reference time defined as the required time that the APPO technique with 1 worker reaches 150,000 sample steps in the environment. Also, $Time_T$ is the technique time within which each technique reaches the specified sample steps.

Fig 7 presents the speedup results of μ-DDRL with 8 workers, MRLCO, DRLCO, and Double DQN techniques. The results depict that μ-DDRL works roughly 8 to 12 times faster than Double DQN and MRLCO techniques in terms of SP. Moreover, the speedup of μ-DDRL is about 40% higher than the DRLCO technique. Therefore, the learning time of the μ-DDRL agent is significantly lower than other techniques, which helps reduce the startup time of DRL agents (i.e., the time DRL agents require to converge) and also provides greater adaptability to highly dynamic and stochastic computing environments, such as fog computing.

6.2.5 Decision Time Overhead Analysis

In this experiment, we study the average Decision Time Overhead (DTO) of all service offloading techniques. DTO represents the average required amount of time that techniques require to make an offload decision for each service with several tasks. For evaluation, we consider services that contain 40 tasks (i.e., $L = 40$). As Fig. 8 depicts, while μ-DDRL outperforms MRLCO and DRLCO, it has a higher DTO compared to Double DQN technique. However, because μ-DDRL converges faster to more suitable solutions and its higher efficiency in the training and experience trajectory generation, the negligible increase of DTO in worst-case scenarios is acceptable.

6.2.6 Optimality Analysis

In this experiment, we are evaluating the effectiveness of our proposed solution by comparing its performance against the optimal results. To achieve the optimal results, we employed MiniZinc[3], which allows the integration of various optimization solvers, to explore all possible candidate configurations for service offloading. Due to the extensive time required to find the optimal solution, in this experiment, we reduce the number of candidate servers to 8 (i.e., $|\mathcal{M}| = 8$) and the number of tasks for the server to 10 (i.e., $L = 10$) to reduce the search space of the problem.

[3]https://www.minizinc.org/

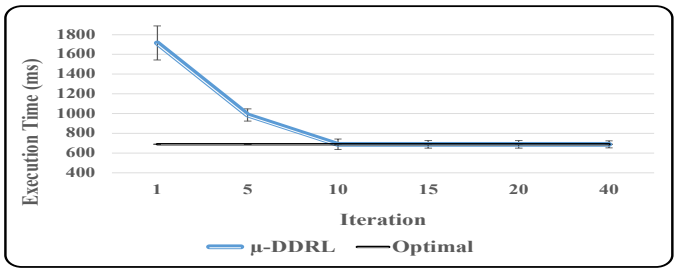


Figure 9: Optimality Analysis

Fig. 9 shows the performance of μ-DDRL compared to the optimal results, obtained from Minizinc, in terms of execution time. The results illustrate that μ -DDRL can reach the optimal solution after 10 training iterations. As the search space for the problem grows, μ-DDRL requires extra exploration and training to converge to optimal solutions.

7 Conclusions and Future Work

In this paper, we proposed a distributed DRL technique, called μ-DDRL, for efficient DAG-based service offloading for complex IoT services in highly stochastic and dynamic fog computing environments. We designed an MDP model for service offloading and defined a reward function to minimize the execution time of services while meeting their deadlines. Next, we proposed μ-DDRL as a service offloading technique working based on actor-critic architecture and Asynchronous Proximal Policy Optimization (APPO). The designed DDRL framework enables multiple parallel actors (i.e., brokers that make service offloading decisions) to work simultaneously in a computing environment and share their experience trajectories with the learner for more efficient and accurate training of the target policy. Moreover, we used two off-policy correction techniques, called PPO Clipping and V-trace, to further improve the convergence speed of μ-DDRL towards the optimal service offloading solution. Based on extensive experiments, we demonstrate that μ DDRL outperforms its counterparts in terms of scalability and adaptability to highly dynamic and stochastic computing environments and different service models. Furthermore, experiments show that μ-DDRL can obtain service

offloading solutions to reduce the execution of IoT services by up to 60% compared to its counterparts.

As part of future work, we plan to consider DDRL techniques for IoT services with dynamic mobility scenarios. Moreover, we plan to update the reward function to enable dynamic service migration among different surrogate servers.

REFERENCES

[1] Q. Luo, C. Li, T. H. Luan, and W. Shi, "Minimizing the delay and cost of computation offloading for vehicular edge computing," *IEEE Transactions on Services Computing*, vol. 15, no. 5, pp. 2897–2909, 2021.

[2] Q. Qi, J. Wang, Z. Ma, H. Sun, Y. Cao, L. Zhang, and J. Liao, "Knowledge-driven service offloading decision for vehicular edge computing: A deep reinforcement learning approach," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 5, pp. 4192–4203, 2019.

[3] A. Pekar, J. Mocnej, W. K. Seah, and I. Zolotova, "Application domain-based overview of iot network traffic characteristics," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–33, 2020.

[4] C. Liu and L. Ke, "Cloud assisted internet of things intelligent transportation system and the traffic control system in the smart city," *Journal of Control and Decision*, pp. 1–14, 2022.

[5] M. Goudarzi, M. S. Palaniswami, and R. Buyya, "A distributed deep reinforcement learning technique for application placement in edge and fog computing environments," *IEEE Transactions on Mobile Computing*, 2021, accepted, in press.

[6] S. Tian, C. Chi, S. Long, S. Oh, Z. Li, and J. Long, "User preference-based hierarchical offloading for collaborative cloud-edge computing," *IEEE Transactions on Services Computing*, 2021.

[7] 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 Transactions on Mobile Computing*, vol. 20, no. 4, pp. 939 – 951, 2021.

[8] 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 Transactions on Mobile Computing*, vol. 20, no. 4, pp. 1298 – 1311, 2021.

[9] 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 *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments*, 2021, pp. 1–8.

[10] M. Goudarzi, M. Palaniswami, and R. Buyya, "A fog-driven dynamic resource allocation technique in ultra dense femtocell networks," *Journal of Network and Computer Applications*, vol. 145, p. 102407, 2019.

[11] F. Al-Doghman, N. Moustafa, I. Khalil, Z. Tari, and A. Zomaya, "Ai-enabled secure microservices in edge computing: Opportunities and challenges," *IEEE Transactions on Services Computing*, 2022.

[12] A. Al-Shuwaili and O. Simeone, "Energy-efficient resource allocation for mobile edge computing-based augmented reality applications," *IEEE Wireless Communications Letters*, vol. 6, no. 3, pp. 398–401, 2017.

[13] Y. Ding, K. Li, C. Liu, Z. Tang, and K. Li, "Budget-constrained service allocation optimization for mobile edge computing," *IEEE Transactions on Services Computing*, 2021.

[14] 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 Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 242–253, 2020.

[15] A. Petrenko, Z. Huang, T. Kumar, G. Sukhatme, and V. Koltun, "Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7652–7662.

[16] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1407–1416.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[18] T. Bahreini, M. Brocanelli, and D. Grosu, "Vecman: A framework for energy-aware resource management in vehicular edge computing systems," *IEEE Transactions on Mobile Computing*, 2021, (in press).

[19] L. Chen, C. Shen, P. Zhou, and J. Xu, "Collaborative service placement for edge computing in dense small cell networks," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 377–390, 2021.

[20] E. F. Maleki, L. Mashayekhy, and S. M. Nabavinejad, "Mobility-aware computation offloading in edge computing using machine learning," *IEEE Transactions on Mobile Computing*, 2021, (in press).

[21] H. Badri, T. Bahreini, D. Grosu, and K. Yang, "Energy-aware application placement in mobile edge computing: A stochastic optimization approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 909–922, 2020.

[22] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.

[23] C. Zhang and Z. Zheng, "Task migration for mobile edge computing using deep reinforcement learning," *Future Generation Computer Systems*, vol. 96, pp. 111–118, 2019.

[24] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Transactions on Mobile Computing*, vol. 19, no. 11, pp. 2581–2593, 2019.

[25] 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 of Things Journal*, vol. 6, no. 3, pp. 4005–4018, 2018.

[26] R. Garaali, C. Chaieb, W. Ajib, and M. Afif, "Learning-based task offloading for mobile edge computing," in *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE, 2022, pp. 1659–1664.

[27] P. Li, W. Xie, Y. Yuan, C. Chen, and S. Wan, "Deep reinforcement learning for load balancing of edge servers in iov," *Mobile Networks and Applications*, vol. 27, no. 4, pp. 1461–1474, 2022.

[28] L. Liao, Y. Lai, F. Yang, and W. Zeng, "Online computation offloading with double reinforcement learning algorithm in mobile edge computing," *Journal of Parallel and Distributed Computing*, vol. 171, pp. 28–39, 2023.

[29] F. Ramezani Shahidani, A. Ghasemi, A. Toroghi Haghighat, and A. Keshavarzi, "Task scheduling in edge-fog-cloud architecture: a multi-objective load balancing approach using reinforcement learning algorithm," *Computing*, vol. 105, no. 6, pp. 1337–1359, 2023.

[30] H. Zhou, Z. Wang, H. Zheng, S. He, and M. Dong, "Cost minimization-oriented computation offloading and service caching in mobile cloud-edge computing: An a3c-based approach," *IEEE Transactions on Network Science and Engineering*, 2023.

[31] D. Kimovski, N. Mehran, C. E. Kerth, and R. Prodan, "Mobility-aware iot applications placement in the cloud edge continuum," *IEEE Transactions on Services Computing*, 2021, (in press).

[32] M. Goudarzi, M. Palaniswami, and R. Buyya, "A distributed application placement and migration management techniques for edge and fog computing environments," in *Proceedings of the 16th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE, 2021, pp. 37–56.

[33] S. Shekhar, A. Chhokra, H. Sun, A. Gokhale, A. Dubey, and X. Koutsoukos, "Urmila: A performance and mobility-aware fog/edge resource management middleware," in *Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 118–125.

[34] C. Mouradian, S. Kianpisheh, M. Abu-Lebdeh, F. Ebrahimnezhad, N. T. Jahromi, and R. H. Glitho, "Application component placement in nfv-based hybrid cloud/fog systems with mobile fog nodes," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 5, pp. 1130–1143, 2019.

[35] H. Sami, A. Mourad, and W. El-Hajj, "Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 778–790, 2020.

[36] M. Bansal, I. Chana, and S. Clarke, "Urbanenqosplace: A deep reinforcement learning model for service placement of real-time smart city iot applications," *IEEE Transactions on Services Computing*, 2023.

[37] X. Xu, Q. Liu, Y. Luo, K. Peng, X. Zhang, S. Meng, and L. Qi, "A computation offloading method over big data for iot-enabled cloud-edge computing," *Future Generation Computer Systems*, vol. 95, pp. 522–533, 2019.
[38] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1085–1101, 2020.
[39] M. Luo, J. Yao, R. Liaw, E. Liang, and I. Stoica, "Impact: Importance weighted asynchronous architectures with clipped target networks," *arXiv preprint arXiv:1912.00167*, 2019.
[40] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.
[41] H. Wu, W. Knottenbelt, and K. Wolter, "An efficient application partitioning algorithm in mobile environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 7, pp. 1464–1480, 2019.
[42] T. Q. Dinh, J. Tang, Q. D. La, and T. Q. Quek, "Offloading in mobile edge computing: Task allocation and computational frequency scaling," *IEEE Transactions on Communications*, vol. 65, no. 8, pp. 3571–3584, 2017.
[43] M. Goudarzi, Q. Deng, and R. Buyya, "Resource management in edge and fog computing using fogbus2 framework," *arXiv preprint arXiv:2108.00591*, 2021.
[44] Z. Cheng, M. Min, M. Liwang, L. Huang, and Z. Gao, "Multi-agent ddpg-based joint task partitioning and power control in fog computing networks," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 104–116, 2021.
[45] T. Xie and X. Qin, "Scheduling security-critical real-time applications on clusters," *IEEE transactions on computers*, vol. 55, no. 7, pp. 864–879, 2006.
[46] 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 Generation Computer Systems*, vol. 110, pp. 1098–1115, 2020.
[47] Y. Chen, S. Zhang, Y. Jin, Z. Qian, M. Xiao, J. Ge, and S. Lu, "Locus: User-perceived delay-aware service placement and user allocation in mec environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1581–1592, 2022.
[48] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, 2020.

APPENDIX A
PARAMETERS AND RESPECTIVE DEFINITIONS

A summary of the parameters and their respective definitions is presented in table 3.

Table 3: Parameters and respective Definitions

Parameter	Definition	Parameter	Definition
G	A service represented as a DAG.	$\mathcal{V}$	The set of tasks within each service G.
v_i	The ith task in a set of tasks $\mathcal{V}$.	$\mathcal{E}$	The set of edges denoting the data dependencies between tasks for each service G.
$e_{i,j}$	A data flow between v_i (parent) and v_j (child)	L	The maximum number of tasks in a service G
$e^w_{i,j}$	The amount of input data that task v_j receives from task v_i.	v^w_j	The number of CPU cycles required for the processing of the task.
v^{ram}_j	The required amount of RAM required to run the task.	ζ_{v_j}	The maximum tolerable delay for the task.
$\mathcal{P}(v_j)$	The set of all predecessor tasks of v_j.	$\mathcal{M}$	The set of available servers
$m^{y,z}$	A server in which y illustrates the server's type and z presents the index of the corresponding server's type.	x_{v_j}	The offloading configuration of task v_j.
$\mathcal{X}$	The offloading configuration set of a service.	$CP(v_i)$	A function to indicate whether or not each task is on the critical path of the service.
$\mathcal{CP}$	The set of tasks on the critical path of the service.	$\mathscr{T}_{x_{v_j}}$	The execution time of each task v_j.
$\mathscr{T}^{proc}_{x_{v_j}}$	The processing time of the task on the corresponding server.	$\mathscr{T}^{input}_{x_{v_j}}$	The time it takes for the input data to become available for that task.
$f^s_{x_{v_j}}$	The processing speed of the corresponding assigned server.	$b(x_{v_i}, x_{v_j})$	The data rate (i.e., bandwidth) between the selected servers for the execution of v_i and v_j.
$l(x_{v_i}, x_{v_j})$	The communication latency between two servers.	$\eth^c$	The propagation speed for the communication medium.
$d(x_{v_i}, x_{v_j})$	The Euclidean distance between the coordinates of the participating servers in the Cartesian coordinate system.	$SS(x_{v_i}, x_{v_j})$	A function that indicates whether the servers assigned to each pair of tasks are the same (that is, 0 if $x_{v_i} = x_{v_j}$) or different (i.e., 1 if $x_{v_i} \neq x_{v_j}$).
$\mathcal{T}(\mathcal{X})$	The execution time of each service based on its configuration.	$IA(v_j, m^{y,z})$	An indicator function to check if the task v_j is assigned to the server $m^{y,z}$ ($IA = 1$) or not ($IA = 0$).
$\mathbb{S}$	The state space in MDP.	$\mathbb{A}$	The action space in MDP.
$\mathbb{P}$	A state transition between states in MDP.	$\mathbb{R}$	The reward function in MDP.
γ	The discount factor in MDP.	$\pi(a_t\|s_t)$	The agent policy.
s_t	The state of environment at time step t.	a_t	The action at time step t.
r_t	The reward at time step t	$F^{\mathcal{M}}_t$	The feature vector of all M servers at time step t.
$F^{v_j}_t$	The feature vectors of the current task in a service.	$f^{m^{y,z}}_i$	The ith feature corresponding to the server $m^{y,z}$.
$f^{v_j}_i$	The ith feature of the task v_j.	Φ	The failure penalty to penalize actions that violate ζ_{v_j}.