Deep Meta Q-Learning Based Multi-Task Offloading in Edge-Cloud Systems

Nelson Sharma, Aswini Ghosh, Rajiv Misra, *Senior Member, IEEE*, and Sajal K. Das, *Fellow, IEEE*

***Abstract*—Resource-constrained edge devices can not efficiently handle the explosive growth of mobile data and the increasing computational demand of modern-day user applications. Task offloading allows the migration of complex tasks from user devices to the remote edge-cloud servers thereby reducing their computational burden and energy consumption while also improving the efficiency of task processing. However, obtaining the optimal offloading strategy in a multi-task offloading decision-making process is an NP-hard problem. Existing Deep learning techniques with slow learning rates and weak adaptability are not suitable for dynamic multi-user scenarios. In this article, we propose a novel deep meta-reinforcement learning-based approach to the multi-task offloading problem using a combination of first-order meta-learning and deep Q-learning methods. We establish the meta-generalization bounds for the proposed algorithm and demonstrate that it can reduce the time and energy consumption of IoT applications by up to 15%. Through rigorous simulations, we show that our method achieves near-optimal offloading solutions while also being able to adapt to dynamic edge-cloud environments.**

***Index Terms*—Deep q-learning, directed acyclic graph, edge-cloud computing, Internet of Things, meta-learning, multi-task offloading.**

I. Introduction

With the advent of *Internet of Things* (IoT) and rapid advancements in communication technologies, there has been an unprecedented growth in the volume of data generated by IoT and end-user devices. AI-enabled Intelligent IoT devices have increased demand for computational resources while also trying to operate under stringent latency and capacity constraints which cannot be adequately addressed by a conventional centralized cloud computing architecture [1]. Edge computing emerges as an extension of cloud computing which shifts the function of cloud services to the proximity of end users. Edge caching, edge training, edge inference, and edge offloading are four fundamental components of edge intelligence [2]. Edge offloading helps to migrate complex and computation-intensive tasks to nearby cloudlets by utilizing powerful decision-making capabilities [3] of edge intelligence.

Making optimal offloading decisions in *multi access edge computing* (MEC) is generally affected by the quality of network connection, wireless communication channels, preferences of application users, mobility of IoT devices, and availability of cloud servers. A downside of edge offloading is that it increases the amount of data communication between the IoT and edge devices which may cause network congestion and affects the latency of user devices. In addition, adjusting the offloading decisions in rapidly changing MEC environments is usually a difficult task because it requires recomputing optimal offloading solutions every time the environment changes. It creates huge service delays [4] and increases resource consumption of edge devices.

By extending the offloading process to cloud servers, users can essentially overcome any resource limitations imposed by their devices while also consuming less overall energy. In [35], the authors considered offloading decisions for a series of dependent tasks considering the delay and energy consumption of the concerned devices. It is assumed that the application sub-tasks are *atomic* i.e., a task can be either performed locally or completely offloaded to a remote location. Being able to execute multiple sub-tasks in parallel can reduce the overall execution time of user applications, which is highly desirable for every user application today. Multi-threaded processing models [28], [29], [30] and *directed acyclic graphs* (DAG) [31], [32], [33], [34] are some of the known techniques for modeling applications in a task offloading scenario.

Deep Reinforcement Learning (DRL) is a promising way to address task offloading issues but the main disadvantage of DRL is slower learning speed and weak inductive bias, making it generally less efficient [6]. Recent applications of DRL to various MEC task offloading problems are discussed in [2], [24], [25], [27], where the models learn the offloading policy by interacting with the MEC environment but strictly consider the MEC host, wireless channel and user equipment to be stationary. All these methods have the shortcoming of weak adaptability in dynamic environments as they require full re-training in new environments, making it a very time-consuming process.

Researchers have introduced meta-reinforcement learning [26] to address the adaptability issue by leveraging knowledge learned from a range of training tasks to perform significantly better on new or unseen tasks. *Meta reinforcement learning* (MRL) tries to learn a *meta-policy* which, when trained,

Manuscript received 8 February 2022; revised 13 March 2023; accepted 23 March 2023. Date of publication 5 April 2023; date of current version 6 March 2024. The work of Sajal K. Das was supported by the NSF under Grants CNS-2008878 (FLINT: Robust Federated Learning for Internet of Things), OAC-1916084 (SANDY), and OAC-2104078 (CANDY). Recommended for acceptance by A. Capone. *(Corresponding author: Aswini Ghosh.)*

Nelson Sharma, Aswini Ghosh, and Rajiv Misra are with the Department of Computer Science, Engineering, Indian Institute of Technology, Patna 801106, India (e-mail: nelson_2121cs07@iitp.ac.in; aswini_2121cs02@iitp.ac.in; rajivm@iitp.ac.in).

Sajal K. Das is with the Department of Computer Science, Missouri University of Science, Technology, Rolla, MO 65409 USA (e-mail: sdas@mst.edu).

Digital Object Identifier 10.1109/TMC.2023.3264901

can learn a new or unseen task, using a small number of interactions with the environment. MRL algorithms usually have two learning procedures, the *inner loop* and the *outer loop*. The inner loop learns a new task, while the outer loop uses its experiences over different task contexts in random environments to gradually adjust the parameters of meta-policy such that new tasks can be learned in a fewer number of learning steps [27]. It should be noted that the term *task* used in the context of meta-learning represents a *meta-task* and is different from that used in the context of *task* offloading.

In this research, we introduce a novel meta-learning method that accounts for multi-task dependency and quick adaptability while optimizing offloading decisions, ensuring that total execution time and energy consumption are kept to a minimum in a dynamic edge-cloud environment. The major contribution of this work can be summarized as follows:

- We propose a framework for *multi-task* offloading where we use directed acyclic graphs (DAGs) for modeling applications and consider that the offloading decisions are to be made for multiple dependent tasks across multiple remote edge or cloud servers.
- Taking into account the dynamic nature of edge-cloud environments, we formulate the task offloading problem as a few-shot meta-learning [47] problem.
- We propose a meta-learning algorithm, called *Deep Meta Q-learning based task-offloading* (DMQTO), which uses first-order meta-learning and deep Q-learning to address the multi-task offloading problem.
- We evaluate the performance of our algorithm through rigorous testing under a variety of dynamic edge-cloud environments. We observe that the proposed method can achieve up to 15% improvements in time and energy consumption of applications over four baseline methods.

The rest of the paper is organized as follows. We have reviewed the related work in Section II. The system model and problem formulation are presented in Section III. In Section IV, we have presented the proposed Deep Meta Q-learning based offloading framework and algorithm. Section V contains experimental results and performance evaluation. Finally, Section VI concludes the paper.

II. Related Work

In the existing literature, the task-offloading methods used in MEC or similar environments can be broadly classified into two categories – *Traditional* and *Deep Learning* methods.

Traditional methods mainly used *heuristics* and *meta-heuristic* based solution and *numerical optimization* based methods. The authors in [7] modeled the computation offloading decision problem as a multi-user computation offloading game for mobile-edge cloud computing in a multi-channel wireless context and proposed a distributed computation offloading algorithm that can successfully achieve the game's Nash equilibrium. In [8], the computation offloading problem is formulated in a multi-cell mobile edge-computing scenario. In the single-user case, the authors computed the global optimal solution of the resulting non-convex optimization problem in closed form. In the more general multi-cell multi-user scenario, they developed centralized and distributed *Successive Convex Approximation* based algorithm with provable convergence to local optimal solutions.

A *Lyapunov optimization* method was proposed in [9] with a goal to optimize the long-term problems on a slot-by-slot basis. The authors utilized an open Jackson queuing network to optimize caching, jointly with task offloading. Based on the Lyapunov optimization novel eTime strategy is presented in [10] for energy-efficient data transmission between cloud and mobile devices. eTime aggressively and adaptively seizes the timing of good connectivity to pre-fetch frequently used data while deferring delay-tolerant data in bad connectivity. In [11] is proposed the low complexity *Lyapunov optimization-based dynamic computation offloading* algorithm for green MEC systems with *energy harvesting* (EH) devices which jointly decides on the offloading decision, the CPU-cycle frequencies for mobile execution, and the transmit power for computation offloading. In [12], the authors investigated the trade-off between energy consumption and execution delay for dynamic offloading tasks in an MEC system with EH capabilities and proposed an *online dynamic Lyapunov optimization-based offloading* algorithm.

Key challenges of task offloading in blockchain-enabled heterogeneous IoT-edge-cloud computing architecture is addressed in [13] that proposed an *energy-efficient dynamic task offloading* algorithm which can dynamically offload tasks by choosing the optimal computing location in an online way. In [14], the authors investigated two types of delayed offloading policies – the partial offloading model and the full offloading model. Both models minimize the *energy-response time-weighted product* metric. The authors have shown that for delay-sensitive applications, the partial offloading model is mostly preferred and if applications are delay-tolerant, the full offloading model outperforms the other offloading models when using long deadlines.

A computing offloading game for mobile devices and edge-cloud servers is proposed in [15] which showed the existence of Stackelberg equilibrium in that game. In [51] is proposed distributed task offloading algorithm by formulating the offloading problem as a multi-user potential game. In [16], the authors proposed a novel *memetic algorithm* based application placement technique to solve the task offloading problems in computing environments with multiple IoT devices, multiple fog/edge servers, and cloud servers. The multi-user computation offloading problem is modeled in [43] as a *mixed-integer linear programming* problem and an iterative heuristic algorithm is propsoed to make offloading decisions dynamically. In [33], a heuristic algorithm is proposed for the dependency-aware task offloading problem without considering parallelism among the sub-tasks. The traditional approaches cannot handle large and complex decision-making processes and require a huge amount of computation time, making them impractical for many real-world scenarios.

In recent years, *artificial intelligence* (AI) and *deep learning* (DL) have gained a significant amount of popularity among researchers. Although reinforcement learning (RL) shows promising results, it usually suffers from slow learning and weak adaptability in unseen environments. In such cases, conventional

RL algorithms perform poorly since the learned parameters cannot make accurate decisions in changing environments and require retraining from scratch, making the learning process very slow. Repetitive training consumes large amount of computation resources and increases the overall time consumption of the offloading algorithm.

A *deep reinforcement learning-based online offloading* algorithm is proposed in [5] to maximize the weighted sum computation rate in wireless-powered MEC networks with binary computation offloading. In [45], the authors applied deep-imitation-learning based algorithms for fine-grained computation offloading for a single mobile device within MEC networks. In [46], a *distributed deep learning-based offloading* algorithm is proposed for MEC networks to minimize the overall system utility including both the total energy consumption and the delay in finishing the task. In [18] is proposed a *distributed deep learning-driven task offloading* algorithm where multiple parallel deep neural networks (DNNs) are adopted to effectively and efficiently generate offloading decisions over the *mobile device* (MDs), edge-cloud server, and central cloud server. To obtain the optimal offloading policy in a dynamic blockchain network with MEC for multiple users, the authors in [20] proposed reinforcement learning-based task offloading algorithms but did not consider dynamic environment changes in MEC.

A *meta reinforcement learning-based task offloading* framework for edge-cloud computing is proposed in [35] that considered a single task at a time for offloading decision-making. Since this method considers a single task, it is not suitable for dependent multi-task application models where the offloading decisions for multiple tasks would be taken at a time. In [36], a DRL-based methodology is developed for the MEC system's DAG-based multi-task computation offloading strategy, but this approach has very limited adaptability in MEC environments that change often.

IoT applications are usually composed of multiple tasks that depend on each other. Therefore, while making offloading decisions, it is important to take task dependencies into account. In [44], a multi-user partial computation offloading is formulated in the MEC system to minimize the weighted sum of energy consumption of *smart mobile devices* (SMDs). In [21], the authors developed a successful offloading strategy to achieve the lowest possible total latency and used meta-reinforcement learning for fast adaptability. The authors use directed acyclic graphs (DAGs) to model mobile applications and use policy optimization to train a sequence-to-sequence (seq2seq) neural network. In [48], the dependent task offloading is approached using both deep Q-learning and proximal policy optimization techniques. In [50], the authors used first-order meta-RL methods to obtain fine-tuned policies in heterogeneous edge/cloud computing environments with multiple mobile terminal users, varying data volumes, and varying task workloads. However, they do not consider inter-task dependency. In [49], a meta-learning for computation offloading in dynamic MEC is proposed, without ctaking task dependency into account.

The deep learning methods discussed so far can easily outperform traditional methods in complex decision-making scenarios. However, ensuring adaptability and minimizing total execution time as well as energy consumption in dynamic environments for highly complex, multi-task systems is very challenging. Traditional optimization algorithms require iteratively adjusting the offloading decisions towards the optimum which is often infeasible for real-time system optimization under a fast-changing environment. Therefore, improved intelligent *meta-learning* algorithms are well suited to address the aforementioned objective.

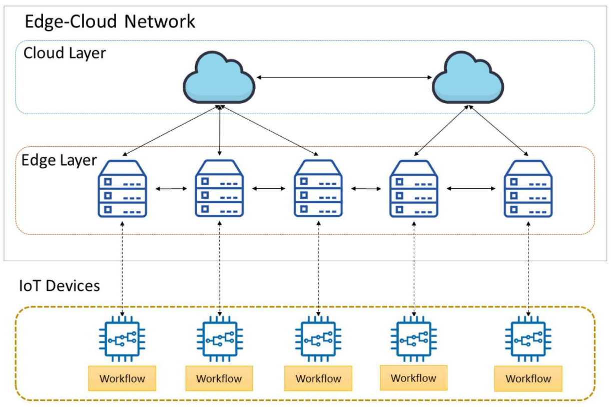

Fig. 1. Edge-cloud network with multiple IoT devices.

III. System Model

As shown in Fig. 1, the edge-cloud network model consists of multiple cloud and edge servers interconnected via a network with the IoT devices directly connecting to nearby high-bandwidth edge servers. For the remainder of this paper, we shall refer to all end-user devices as *IoT devices* while the edge and cloud servers shall be referred to as *remote servers*.

A. Environment Model

The environment is designed to model multi-user edge-cloud scenarios like MECs. It is based on an underlying edge-cloud network topology where each server or device is referred to as a location (l). An offloading process that optimizes the offloading decision based on parameters like task execution time and energy consumption, requires access to underlying environment parameters such as available bandwidth, transmission delays in the network, computation power of cloud servers and energy requirement of the user and edge servers to compute offloading costs. It is assumed that this information is available through appropriate monitoring tools deployed in the environment. We characterize such an environment model using the following parameters:

- $D_R(l_i, l_j)$ represents the *Data Time Consumption* i.e., the time taken to move a unit of data between locations l_i and l_j over the network.
- $D_E(l_i)$ represents the *Data Energy Consumption* i.e., the energy consumed for processing a unit of data at the location l_i, which includes energy consumption for transmitting and receiving data through appropriate networking hardware.

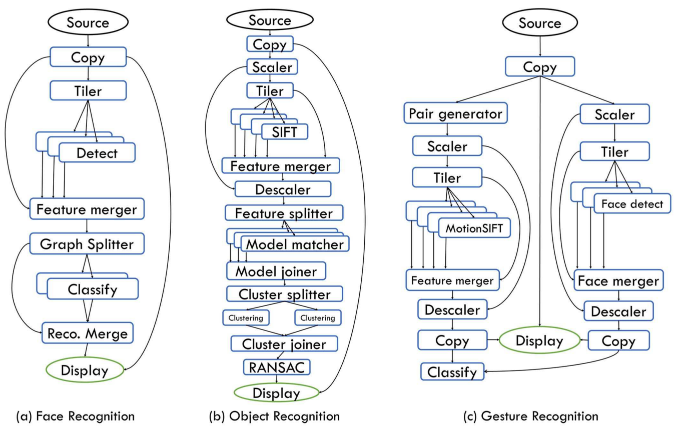

Fig. 2. Example of DAGs for computer vision applications.

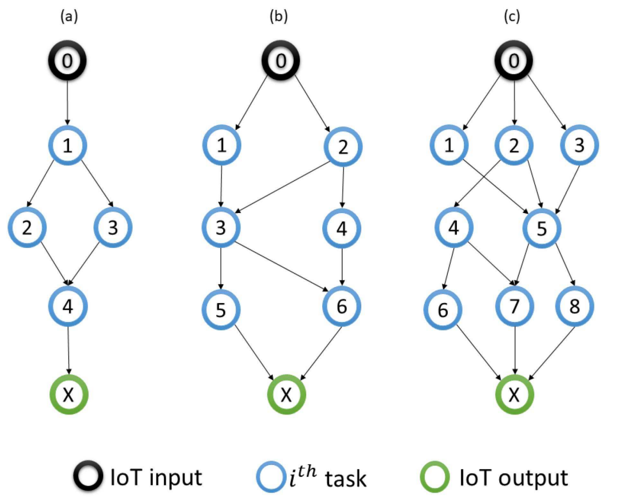

Fig. 3. IoT application workflows modeled as directed acyclic graphs (DAGs).

- $V_R(l_i)$ represents the *Task Time Consumption* i.e., the time taken to execute a unit task at the location l_i.
- $V_E(l_i)$ represents the *Task Energy Consumption* i.e., the energy consumed for executing a unit task at the location l_i.

B. Application Model

As described in [21], [35], [36], [48], [53], [54], IoT devices can have applications composed of multiple tasks with inner dependencies among them. We refer to such a set of dependent tasks as an *application workflow*. A workflow can be interpreted as one large set of programs or tasks to be executed in a certain order as defined by the data dependencies between them [35], [53]. An application may generate a variety of workflows that are essentially *task graphs* modeled using directed acyclic graphs (DAGs). Some examples of computer vision application DAGs [54] are shown in Fig. 2. In such a workflow DAG, the nodes represent the *tasks* in a workflow while the edges represent the *data* dependencies i.e., communication required between tasks. It is assumed that each task requires some input data and produces certain output data for the next task. After the execution of a certain task in the workflow, the output data produced, has to be moved over the network to the location where the next task is to be executed. Also, the initial data required for the first set of tasks is assumed to be located on the IoT device. At the end of completing all tasks, the final output data must be sent back to the IoT device which generated the workflow. Formally, a workflow containing N tasks is represented by a DAG (w), defined as:

$$w = \{V, D\}$$

where, V represents the vertex set and D represents the edge set, defined as:

$$V = \{(v_i) : 0 \leq i \leq (N+1)\}$$

$$D = \{(d_{i,j}) : 0 \leq i, j \leq (N+1)\}$$

Accordingly, for the ith task, v_i represents the task size (in CPU cycles) and $d_{i,j}$ represents the size of data dependency (in bytes) to the jth task. Note that the *entry task* (v_0) and the *exit task* (v_{N+1}) are added to the workflow-DAG as a proxy for IoT input and output respectively. Both the entry and the exit task have unit task size which represents some initial and final processing delay at the IoT devices. These tasks cannot be offloaded and their sizes will not affect the offloading decision. Also, the set J_i represents the set of all parent tasks of the ith task, and the set K_i represents the set of all tasks that are dependent on the ith task. Fig. 3 shows a few DAG topologies used to represent an application workflow.

In order to leverage the computation resource of the edge servers, the application would require to offload its workflow consisting of multiple dependent tasks, across the available remote locations using an appropriate *offloading scheme*. Such a scheme should be able to produce offloading solutions of the form p, represented by a sequence of N offloading locations as:

$$p = \{l_1, l_2, l_3, \ldots, l_N\}$$

where l_i represents the offloading location for the ith task of the workflow. It should be noted that the IoT devices are referenced by a special location number $l = 0$. Hence, $l_i = 0$ represents no offloading i.e., the task is not to be offloaded to a remote location but executed locally on the IoT device. Since only IoT devices can generate workflows, it is assumed that $l_0 = l_{N+1} = 0$. For example, Fig. 4 depicts the placement solution $p = \{1, 4, 2, 5\}$ for the given workflow containing four tasks. In the solution vector p, the locations 1 and 2 refer to edge servers whereas the locations 4 and 5 refer to cloud servers.

C. Cost Model

The *offloading cost* (U) captures the total weighted cost of offloading a workflow (w) which includes task execution time, delay in moving data within the network, and the energy consumed in processing tasks and data by remote servers and IoT devices. Specifically, the offloading cost for a given workflow is

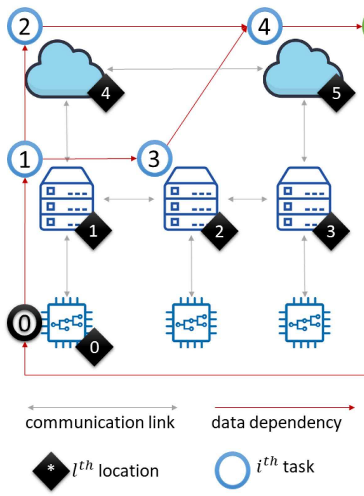

Fig. 4. Task placement across the edge-cloud network using offloading solution $p = \{1, 4, 2, 5\}$ for a 3-level DAG.

the sum of the costs of the total energy consumed and the total delay incurred in executing the workflow i.e., from the time of the request, till the final output data is received back at the requesting IoT device.

To capture the proper trade-off between execution time and energy consumption, one should agree on an appropriate cost per unit of both time and energy. Hence a trade-off is decided where we assume equivalence between some amount of time and energy. The trade-off is implementation specific and should be decided based on the energy consumption and execution time of hardware components used by the participating IoT devices and remote servers. We assume that $\mathbb{T}_t$ units of time (milliseconds) are equivalent to $\mathbb{T}_e$ units of energy (millijoules) and both the terms have unit cost. Hence, the cost per unit of execution time (C_T) and the cost per unit of energy consumption (C_E) can be defined as:

$$C_T = \frac{1}{\mathbb{T}_t} \tag{1}$$

$$C_E = \frac{1}{\mathbb{T}_e} \tag{2}$$

The offloading cost depends on the parameters that define the underlying edge-cloud network model which may vary in the number of edge servers $(\mathcal{E})$ and the number of cloud servers $(\mathcal{C})$ that are available. To model the environment, we assume a set of four parameters $\{D_R, D_E, V_R, V_E\}$ as defined in Section III-A. The offloading cost for a workflow consists of two components namely, *energy consumption cost* (E) and *time consumption cost* (T). These components are defined as follows:

- *Energy Consumption Cost* (E), is the combined energy cost incurred in completing all the tasks in a workflow at their respective offloading location in the edge-cloud network, defined as:

$$E = C_E \cdot (E_D + E_V) \tag{3}$$

where, E_D represents the total energy consumed for data communication at the respective locations of task execution, defined as:

$$E_D = \sum_{i=1}^{N} \left[D_E(l_i) \times \left(\sum_{j \in J_i} d_{j,i} + \sum_{k \in K_i} d_{i,k} \right) \right] \tag{4}$$

and E_V represents the total energy consumed in executing all the tasks at their respective offloading locations, defined as:

$$E_V = \sum_{i=1}^{N} v_i \times V_E(l_i) \tag{5}$$

- *Time Consumption Cost* (T), is the cost equivalent of the total time required to complete the execution of a workflow including task execution time and data communication delay. The total execution time of all the tasks in a workflow-DAG is given by the cost of the critical path i.e., the path of maximum delay through the corresponding delay-DAG. To obtain a delay-DAG $(w_\Delta = \{V_\Delta, D_\Delta\})$, we first define a DAG which is isomorphic to the workflow-DAG $(w = \{V, D\})$ and then set its weights as:

$$D_\Delta(i, j) = d_{i,j} \times D_R(l_i, l_j) + v_i \times V_R(l_i) \tag{6}$$

The delay-DAG combines the time consumption for task execution and data transfer. As opposed to workflow-DAG, the delay-DAG has no weights assigned to its nodes. The edge weights in the delay-DAG represent the time delay between dependent tasks i.e., the sum of task execution time and the data transfer time to the next dependent task. The total time consumption cost can then be obtained using:

$$T = C_T \cdot \Delta_{max}(w_\Delta) \tag{7}$$

where Δ_{max} is a function that computes the total delay of the critical path (longest path) in a given delay-DAG (w_Δ). Finding the longest path is an NP-hard problem for an arbitrary graph. However, for directed acyclic graphs, finding the longest path is the same as finding the shortest path on an equivalent graph with negative weights. Hence, the critical path problem for the delay-DAG can be solved in linear time $\mathcal{O}(E + V)$ using a topological sorting-based algorithm [52].

Finally, the total offloading cost (U) for a workflow (w) using an offloading solution (p) can be expressed as:

$$U(w, p) = \delta_t T + \delta_e E \tag{8}$$

where δ_t and δ_e are the weights assigned to the time consumption cost and the energy consumption cost respectively. These weights take binary values $(\delta_t, \delta_e \in \{0, 1\})$ based on the application's *mode of operation*. The applications may run under three major modes of operation as follows:

- *Low Latency Mode:* applications running on low latency mode prefer to improve performance. In this mode, no weight is assigned to the energy consumption cost. Hence, $\delta_e = 0$ and $\delta_t = 1$.
- *Low Power Mode :* applications running on low power mode prefer to reduce the energy consumption of devices.

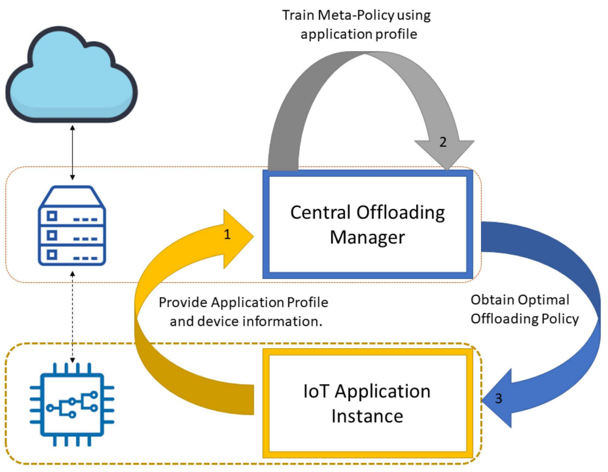
Train Meta-Policy using application profile
2
Central Offloading Manager
1
Provide Application Profile and device information.
Obtain Optimal Offloading Policy
IoT Application Instance
3

Fig. 5. Steps in application-specific task-offloading.

In this mode, no weight is assigned to the time consumption cost. Hence, $\delta_e = 1$ and $\delta_t = 0$.

- *Balanced Mode:* applications running on balanced mode prefer to minimize both energy consumption and execution time. In this mode, equal weights are assigned to both the energy consumption cost and the time consumption cost. Hence, $\delta_e = \delta_t = 1$.

Using an application-specific offloading approach requires each application instance to have its own offloading policy that is fine-tuned to the application-specific workflows. When an application instance is initialized on an IoT device, it communicates with a *central offloading manager* to obtain a refined policy based on the current application profile, user preferences, and device parameters such as transmission rate and energy consumption. The central offloading manager maintains a collection of meta-policies and provides fine-tuned policies to each application instance as and when required.

As shown in Fig. 5, our approach shifts the computation required for training environment-specific policies from IoT devices to edge servers which further reduces the computational burden at the end devices. Another advantage of such an application-specific approach is that it allows the grouping of offloading policies according to the set of applications they apply to. Instead of maintaining a single policy for all types of applications, we can now group the policies according to the similarity in their workflow profiles and the specific IoT devices that they can support. Through our experiments, it is observed that a clever grouping of applications can greatly improve adaptability and allows the use of heterogeneous policies. It should be noted that the terms *offloading scheme* and *offloading policy* can be used interchangeably. A summary of the notation used in this paper is presented in Table I.

D. Problem Formulation

Given an edge-cloud network represented using parameters $\{D_R, D_E, V_R, V_E\}$ containing $\mathcal{E}$ number of edge servers and $\mathcal{C}$ number of cloud servers with the IoT applications generating workflows according to their respective profiles and mode of operation, our objective is to find an optimal offloading scheme that can generate minimum cost offloading solution for any given workflow. Formally, the problem can be expressed as:

$$\min_{p}\{U(w,p)\} \ \ \forall w \in W \tag{9}$$

where W represents the set of all possible workflows that can be generated by an application running on an IoT device under different profiles or modes of operation. The workflows, represented by DAGs, can vary according to the IoT application and user preferences. We consider a variety of workflow DAGs characterized by the following attributes:

- *Node-density* - the number of nodes in the DAG that represent the total number of tasks in the workflow.
- *Height* - the maximum path length starting from the entry task and ending at the exit task.
- *Maximum branch-factor* - represents the maximum number of dependent tasks that any given task can have.

We consider a variety of applications with randomly generated workflow DAGs having the node density, height, and maximum branch factor limited to a maximum of 20. These parameters are described in detail in Section V. The applications are allowed to run in three modes of operation i.e., low latency, low power, and balanced mode. In the following section, we formalize the offloading cost optimization problem using the reinforcement learning framework.

TABLE I
SUMMARY OF NOTATION

Parameter	Description
l_i	Offloading location for i^{th} task.
v_i	Task size for i^{th} task.
$d_{i,j}$	Data dependency size between i^{th} and j^{th} task.
$V_E(l_i)$	Task energy consumption at location l_i
$D_E(l_i)$	Data energy consumption at location l_i
$V_R(l_i)$	Task time consumption at location l_i
$D_R(l_i, l_j)$	Data time consumption between location l_i and l_j
C_E	Cost per unit of energy consumption
C_T	Cost per unit of time consumption
E_D	Total data energy consumption for workflow
E_V	Total task energy consumption for workflow
E	Total energy consumption for workflow
T	Total time consumption for workflow
δ_t	Weight assigned to task time consumption
δ_e	Weight assigned to task energy consumption
U	Total offloading cost
$\mathcal{E}$	Number of edge offloading locations
$\mathcal{C}$	Number of cloud offloading locations
G_n	Node-density for workflow DAG
G_h	Maximum height of workflow DAG
G_b	Maximum branch-factor for workflow DAG

E. Reinforcement Learning

In a reinforcement learning setup, we consider agents that are able to take a sequence of actions in an environment by following a policy. The environment state dynamics are described by a Markov Decision Process (MDP) which can be either stochastic or fully deterministic. The agents build up their experience by taking actions and obtaining a single scalar reward per action. In an episodic setting with fixed horizon (H), we consider the

MDP to have a finite and constant number of state transitions (steps) for each episode i.e., agents can reach a final state from any given initial state, by taking a fixed and constant number of actions. The task of an agent is to learn a policy that maximizes the cumulative reward over every possible episode. Formally, consider an agent operating over an episode of T time-steps where at time-step t the agent encounters state s_t and chooses an action a_t according to its policy (π) and receives a numerical reward r_t. Throughout the paper, it is assumed that the actions are discrete and finite as well.

Q-learning [42] is a well-established technique that uses an action-value function, also called the Q-function: $Q(s_t, a_t)$, to estimate the expected future reward for taking an action a_t in a state s_t. At its core, Q-learning uses the Bellman Optimality equation to characterize the optimal expected future reward function via a state-action value function as:

$$Q^*(s_t, a) = \mathbb{E}[r_t + \gamma \cdot \underset{\acute{a}}{max}\ (s_{t+1}, \acute{a})] \tag{10}$$

where, the expectation is taken w.r.t the distribution of state s_{t+1} and reward r_t obtained by taking action a, and γ represents the discount factor. Unlike classical approaches that use a linear or tabular representation of the Q-function, modern approaches employ the use of non-linear deep neural networks for automatic feature extraction and the parameterized representation of the Q-function. However, it requires large quantities of data and computation resources for the neural network algorithms to learn appropriate feature representation. Even though data collection is quite straightforward, it is crucial for the algorithm to operate on uncorrelated samples of data for stability. To mitigate this problem, the technique of experience replay [22] has proven to be highly successful. The agent uses a replay buffer to store its experience as a data set of transitions of the form $\{s_t, a_t, r_t, s_{t+1}\}$. During the learning process, the agent uses mini-batches of experience $(\mathcal{B})$ drawn randomly from the replay buffer to calculate the Q-value updates and uses stochastic gradient-based methods to optimize the parameterized Q-function as:

$$\underset{\theta}{min} \sum_{i \in \mathcal{B}} [\, Q_\theta(s_i, a_i) - (r_i + \gamma \cdot \underset{a}{max}\ Q^T_\theta(s_i, a))\,]^2 \tag{11}$$

where Q^T_θ represents a target Q-network [40] which is often combined with a deep Q-learning algorithm to avoid any rapid changes to the target values as the parameters of the base Q-network change during learning.

We formalize our problem in terms of a fully deterministic markov decision process (MDP) which is described by the tuple (H, S, A, T_S, R) defined as follows:

- Horizon (H): the number of time steps to reach a final state from any given initial state. In this formulation, we assume the horizon to be the same as the number of tasks in the workflow. The agent makes an offloading decision for one task of the workflow at each time step. Hence,

$$H = \{1, 2, \ldots, N\} \tag{12}$$

- State Space (S): the state vectors carry information about the time-step, the workflow and the partial offloading solution. Assuming that the set W represents all possible workflows and the set P represents the set of all possible placement solutions, we define the state space as

$$S = H \times W \times P \tag{13}$$

Hence, a state is defined by the tuple, $s = (h, w, p)$. The initial state is chosen uniformly from the set of all possible initial states defined as (S_0)

$$S_0 = \{1\} \times W \times \{p_0\} \tag{14}$$

where p_0 is the zero vector in the space of all possible offloading solutions (P). It refers to a solution where none of the tasks are offloaded and are instead executed locally on the IoT device. Hence,

$$p_0 = \{0, 0, \ldots, 0\} \tag{15}$$

- Action Space (A): the action space is a discrete space containing $(1 + \mathcal{E} + \mathcal{C})$ actions with each action referring to an offloading location available on the underlying edge-cloud network. The zero location is representative of no offloading. We define action space as

$$A = \{0, 1, \ldots, \mathcal{E}, \mathcal{E} + 1, \ldots, \mathcal{E} + \mathcal{C}\} \tag{16}$$

- State-Transition function (T_S): since we assume a fully deterministic MDP, the transition functions directly map the current state to the next state instead of defining a transition probability. Hence, $T_S : S \times A \rightarrow S$

$$T_S((h, w, p), a) = (h + 1, w, M(h, p, a)) \tag{17}$$

where, $M(h, p, a)$ represents an action transformation function,

$$M : H \times P \times A \rightarrow P \tag{18}$$

The action transformation function describes how the partial offloading solution (p) changes by applying action a at time-step h.

- Reward function (R): for the deterministic MDP, the reward function defines a real-valued reward for each state-action pair. Hence $R : S \times A \rightarrow \mathbb{R}$

$$R((h, w, p), a) = U(w, p) - U(w, M(h, p, a)) \tag{19}$$

For any given workflow (w), an action (a) at a time-step (h), defines a transformation on the solution $p_{(h-1)} \rightarrow p_{(h)}$. The reward function captures the amount of improvement made to the solution. Since the MDP always starts in the initial state $s_0 = (1, w, p_0)$, the summation of reward over N time steps by following a policy π can be calculated as: $R_N(\pi) = \sum_{i=1}^{N} R((i, w, p_{i-1}), \pi(i, w, p_{i-1})) = \sum_{i=1}^{N} [U(w, p_{i-1}) - U(w, M(i, p_{i-1}, \pi(i, w, p_{i-1})))]$ Assuming $a_i = \pi(i, w, p_{i-1})$, and $p_i = M(i, p_{i-1}, a_i)$ It follows that,

$$R_N(\pi) = \sum_{i=1}^{N} [U(w, p_{i-1}) - U(w, p_i)] \tag{20}$$

Finally,

$$R_N(\pi) = U(w, p_0) - U(w, p_N) \tag{21}$$

Hence,

$$\underset{\pi}{max}[R_T(\pi)] \equiv \underset{\pi}{max}[U(w, p_0) - U(w, p_N)] \tag{22}$$

Since $U(w, p_0)$ is a known constant for a given workflow, maximizing the total reward R_N is the same as minimizing the offloading cost $U(w, p_N)$. It follows,

$$\underset{\pi}{max}[R_T(\pi)] \equiv \underset{\pi}{min}[U(w, p_N)] \tag{23}$$

Therefore, the offloading cost (U) can be minimized by solving the MDP formulated above as long as the set of action transformations guarantees that the minimum cost solution is reachable from initial state $(1, w, p_0)$ in at most N time-steps. The optimal policy (π^*) defines a set of N actions that transform the initial solution (p_0) to the optimal solution (p_N) after N time-steps.

The action transformation function (M) decides how the current solution changes by applying an action on the current state of the MDP. A variety of transformation functions can be used that allow the solution to reach its minimum in at least N time steps. We define transformation function (M) as:

$$M(h, p, a) = \acute{p} \tag{24}$$

where, $\acute{p}$ represents a new solution in which all the tasks following the hth task are offloaded to the location corresponding to the action a, while locations of the tasks prior to the hth task remain unchanged.

As shown in Fig. 6, the MDP always starts from a fixed initial solution (p_0), and M is applied at each time step with some chosen value of action (a). It can be seen that the choice of action transformation function allows the solution vector to be modified in every dimension so that all possible solutions are reachable by at least one unique sequence of N actions. It should be noted that the tasks are offloaded only after the final solution has been produced by the trained agent i.e., after all time steps have been completed. None of the tasks are offloaded on any intermediate time step.

IV. Meta-Learning Based on Deep Q-Learning

In a dynamic edge-cloud environment, any changes to the application profile or mode of operation may cause the topology of underlying workflow-DAGs to be changed. The network bandwidth and execution speed of remote devices are also subjected to dynamic changes causing the environment parameter (D_R, V_R) to change as well, causing a direct change in the reward signal of the underlying MDP. This causes the estimated Q-values to become invalid in the current context and requires retraining in the changed environment. A meta-learning algorithm aims to find the optimal meta-parameters (θ) that can adapt to a new environment i.e., a new set of optimal Q-value estimates, in a fewer number of learning steps. Such a meta-learning setup is often referred to as a few-shot learning [37] problem.

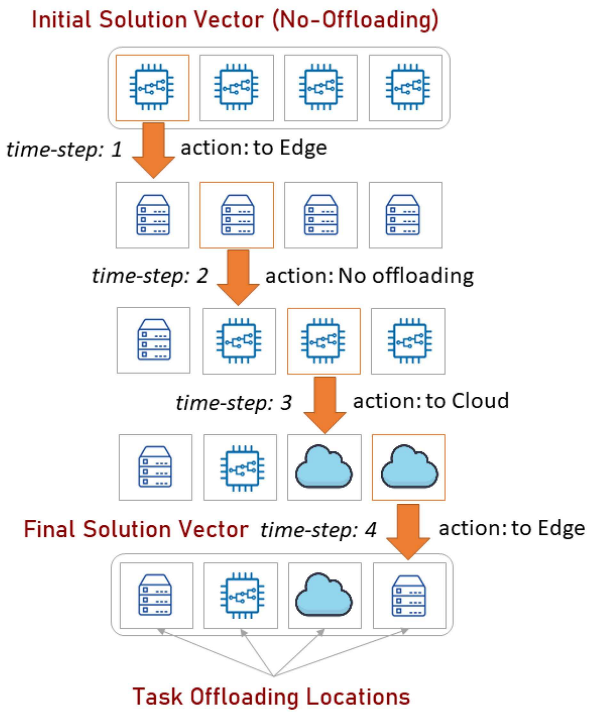

Fig. 6. Transformation of the offloading solution vector $p \in P$, over a single episode of $N = 4$ time-steps.

To address the aforementioned meta-learning problem, we propose a meta-learning algorithm, called the *Deep Meta Q-learning based task-offloading (DMQTO)* which combines the concepts from first-order meta-learning [37] with the deep Q-learning [38] approach. We employ the deep Q-learning method for better sample efficiency as well as having the advantage of off-policy learning for better exploration. The proposed meta-learning algorithm is described in the algorithm [1]. The algorithm consists of two phases, the task-learning phase, and the meta-learning phase.

A. Task-Learning Phase

The task-learning phase represents the inner loop of the meta-algorithm. In this phase, a new DQN $(Q_i : \theta_i)$, initialized using the meta-DQN $(Q_\theta : \theta)$, is trained using trajectories or episodes from the current environment until the loss stabilizes below a minimum value $\mathcal{L}_{min}$. The agent first explores the environment using the ϵ-greedy strategy for a fixed number of episodes and stores its experience in a replay buffer as a set of transitions (D). After enough experience has been collected, a random sample of transitions (D_i^X) is selected for learning.

$$D_i^X = [S_x, A_x, R_x, S'_x] \tag{25}$$

The updated Q-values $(\tilde{Q}_i)$ are calculated using:

$$\tilde{Q}_i(S_x, A_x) = R_x + \gamma \times max[Q_i(S'_x)] \tag{26}$$

where, γ represents the discount factor and Q_i represents the Q-function estimated by the current DQN $(Q_i : \theta_i)$.

Using Mean Squared Error (MSE) as the loss function, we obtain the training loss $\mathcal{L}_X^i$ as:

$$\mathcal{L}_X^i = \frac{1}{X}\sum_{x=1}^{X}(Q_i(S_x, A_x) - \tilde{Q}_i(S_x, A_x))^2 \tag{27}$$

Using a learning rate of α, we update the parameters θ_i using Adam-based stochastic gradient descent as:

$$\theta_i \leftarrow \theta_i - \alpha m/(\sqrt{v} + \mu) \tag{28}$$

where, μ is small constant ($\mu \leq 10^{-8}$), m and v represent the first and second moment vector respectively. We use β_1 and β_2 as the exponential decay rates for our moment estimates thus updating moment vectors at learning step (t) as:

$$m \leftarrow \frac{\beta_1 m + (1-\beta_1)\nabla_{\theta_i}\mathcal{L}_X^i}{1-\beta_1^t} \tag{29}$$

$$v \leftarrow \frac{\beta_2 v + (1-\beta_2)(\nabla_{\theta_i}\mathcal{L}_X^i)^2}{1-\beta_2^t} \tag{30}$$

The process of gradient descent is repeated with random batches of experience (D_i^X) until the training loss reaches the minimum loss value $(\mathcal{L}_{min})$. After the learning iterations have been completed, we obtain the learned parameters θ_i and store them for later use in the meta-learning phase.

After running the task-learning phase for multiple random environments, we obtain a set of learned parameters $\mathcal{B}$

$$\mathcal{B} = \{\theta_i : i \in [1, 2, \ldots, B]\} \tag{31}$$

B. Meta-Learning Phase

The meta-learning phase represents the outer loop of the meta-algorithm. In this phase, the set of parameters, $\mathcal{B}$ learned from multiple random environments in the task-learning phase, is used to update the meta parameters. We want the meta-parameters (θ) to be as *close* as possible, to all the learned parameters $\{\theta_i\}$ so that it takes less number of learning steps for the meta parameters to converge to new parameters. In other words, we directly minimize the *mean squared error* $(\mathcal{L}_\mathcal{B})$ between the meta-parameters and the learned parameters, expressed as:

$$\mathcal{L}_\mathcal{B} = \frac{1}{B}\sum_{i\in\mathcal{B}}\frac{1}{2}(\theta - \theta_i)^2 \tag{32}$$

We again use gradient descent to perform the minimization of the MSE Loss $(\mathcal{L}_\mathcal{B})$. The gradient of $\mathcal{L}_\mathcal{B}$ w.r.t meta-parameters (θ) can be obtained as follows:

$$\nabla_\theta\mathcal{L}_\mathcal{B} = \nabla_\theta\frac{1}{B}\sum_{i\in\mathcal{B}}\frac{1}{2}(\theta-\theta_i)^2 = \frac{1}{2B}\sum_{i\in\mathcal{B}}\nabla_\theta(\theta-\theta_i)^2$$

In second-order meta-learning algorithms like MAML [21], the parameters learned during the inner loop are assumed to be dependent on the meta parameters through updates made in the inner loop. In such a case, during the meta-learning step, second-order back-propagation through multiple gradient update steps, made in the inner loop, becomes a high computation and memory-intensive task. Hence, in first-order meta-learning, we assume the learned parameters (θ_i) to be independent of the meta-parameters so as to avoid the second-order gradient computation. Hence, the above gradient reduces to:

$$\nabla_\theta\mathcal{L}_\mathcal{B} = \frac{1}{B}\sum_{i\in\mathcal{B}}(\theta - \theta_i) \tag{33}$$

Finally, using a learning rate of β, we update the meta-parameters as:

$$\theta \leftarrow \theta - \beta\nabla_\theta\mathcal{L}_\mathcal{B} \tag{34}$$

In the implementation of DMQTO, we consider the batch of learned parameters $(\mathcal{B})$ to be estimated from a uniformly sampled batch of random environments $(\mathcal{T})$. To avoid fine-tuning DQN in the early phases of meta-learning, we start with a larger value of minimum loss $(\mathcal{L}_{min})$ for the task-learning phase and reduced it over time. This improves the training process by avoiding the meta-parameters to move too close to any particular batch of environment parameters.

Algorithm 1: Deep Meta Q-Learning Based Task-Offloading.

```
Require: Hyper-Parameters $(\alpha, \beta, B, X)$
 1: Randomly initialize a new meta-DQN $(Q_\theta : \theta)$
 2: Set $\mathcal{L}_{min} = \infty$
 3: while $\mathcal{L}_{min} > \mathcal{L}_\mu$ do
 4:   Sample a new batch $(\mathcal{T})$ containing $B$ number of
      random environments
 5:   Initialize an empty set $\mathcal{B}$ for storing learned parameters
 6:   for $i = 1$ to $B$ do
 7:     Set the current environment to $\mathcal{T}_i$
 8:     Initialize a new DQN $(Q_i : \theta_i) \leftarrow (Q_\theta : \theta)$
 9:     Initialize moment vectors $m \leftarrow 0$ and $v \leftarrow 0$
10:     while not $(\mathcal{L}_X \leq \mathcal{L}_{min})$ do
11:       Explore the current environment using $\epsilon$-greedy
          strategy and store the experience in the replay
          buffer
12:       Sample a batch of $X$ transitions $(D_i^X)$ from the
          replay buffer for learning
13:       Calculate Q-Updates $\tilde{Q}_i(S_x, A_x)$ from experience
          $(D_i^X)$
14:       Obtain inner loss $\mathcal{L}_X^i$ and use its gradient to update
          moment vectors $m$ and $v$
15:       Updated parameters, $\theta_i \leftarrow \theta_i - \alpha m/(\sqrt{v} + \mu)$
16:     end while
17:     Add $\theta_i$ to set of learnt parameters $\mathcal{B}$
18:   end for
19:   Update loss thresholds as: $\mathcal{L}_{min} \leftarrow \frac{1}{B}\sum_{i=1}^{B}\mathcal{L}_X^i$
20:   Obtain outer loss $\mathcal{L}_\mathcal{B}$ and use its gradient to update
      meta-parameters, $\theta \leftarrow \theta - \beta\nabla_\theta\mathcal{L}_\mathcal{B}$
21: end while
22: output DQN $(Q_\theta : \theta)$ as meta-DQN
```

C. Generalization Bounds

For meta-learning operating in a multi-task setting, the goal is to produce a meta-algorithm $(\mathbb{A})$ that can take data from multiple tasks $(\mathcal{T}_i)$ and output a task-specific learning algorithm $(\mathcal{A})$.

The meta-learning algorithm is said to be data efficient if it can produce a task-specific algorithm using a small number of tasks (n) with fewer data points (m) that are assumed to originate from a common underlying distribution $(\mathbb{P})$. In such a case, the quantity of interest is the *meta-generalization* of $\mathbb{A}$ which describes a relation between the performance of a task-specific algorithm $(\mathcal{A})$ produced by a meta-algorithm $(\mathbb{A})$, on any future tasks sampled from $\mathbb{P}$; and the number of training-tasks (n) and data points per tasks (m). We assume that a learning task $\mathcal{T}_i$ is defined using an independent and identically distributed sample of m data points, called the support set (S), defined as:

$$S_i = \{\, (x_j, x_j) : \forall j \in [1, m] \,\} \sim \mathcal{D}_i^m$$

where, $\mathcal{D}_i \sim \mathbb{P}$. The *transfer risk* $(\mathcal{R})$, as defined in [38], can be expressed as:

$$\mathcal{R}(\mathcal{A}, \mathbb{P}) = \mathbb{E}_{D\sim\mathbb{P}}[\mathbb{E}_{S\sim\mathcal{D}^m}[\mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathcal{L}(f_{\mathcal{A}}(x), y)]]] \tag{35}$$

The transfer risk expresses the expected loss of models $(f_{\mathcal{A}})$ produced using algorithm $\mathcal{A}$, on new tasks $\mathcal{T}_i$ sampled from a distribution $\mathbb{P}$ and hence, characterizes the generalization of algorithm $\mathcal{A}$ over task distribution $\mathbb{P}$. For a meta-learning algorithm $(\mathbb{A})$ that produces learning algorithm $(\mathcal{A})$ by optimizing loss $(\mathcal{L})$, we express the *meta-generalization error bounds* using the definition given in [39] as follows:

Definition 1. For a meta-learning algorithm $\mathbb{A}$ that produces a learning algorithm $\mathcal{A}$ using a meta task sample set $\mathbb{S}$ of n tasks $(\mathbb{S} = \{S_i \ : i \in [1, n]\})$, sampled from a distribution $\mathbb{P}$, the meta-generalization error bound can expressed by the function $\mathbb{B}(\delta, \mathbb{S})$ if for any task distribution $\mathbb{P}$ and $0 < \delta \leq 1$ the following holds with a probability at least $(1 - \delta)$:

$$\mathcal{R}(\mathcal{A}, \mathbb{P}) - \mathcal{L}(\mathcal{A}, \mathbb{S}) < \mathbb{B}(\delta, \mathbb{S}) \tag{36}$$

where $\mathcal{A}$ is obtained from $\mathbb{A}$ using a set of tasks $\mathbb{S}$, as:

$$\mathcal{A} = \underset{A\in\{\mathbb{A}(\mathbb{S})\}}{argmin}\ \mathcal{L}(A, \mathbb{S})$$

The bounds are obtained in [39] using results from algorithmic stability in [40] and [41] for the first-order algorithm in a Few-Shot Classification setting as:

$$\mathcal{R}(\mathcal{A}, \mathbb{P}(\mathcal{T})) - \mathcal{L}(\mathcal{A}, \mathbb{S}) \leq \mathcal{O}\left(\acute{L}^2\acute{T}\sqrt{\frac{\ln(1/\delta)}{n}} + L^2 T\frac{1}{m}\right)$$

where, the loss minimized in the outer loop $\mathcal{L}(\mathcal{A}, \mathbb{S})$ is assumed to be an empirical estimator of the true transfer risk. The quantities L and $\acute{L}$ refer to the Lipschitz constant of the inner and outer loss respectively whereas T and $\acute{T}$ represent the number of learning steps in the inner loop and outer loop respectively.

In the reinforcement learning setup of the DMQTO algorithm, we note that the inner loop may use multiple batches of experience replay for updating Q-values until the loss reaches $\mathcal{L}_{min}$ at the ith iteration of the inner loop. Considering the batch size of X, the data points per task (m) can be bounded by $\mathcal{O}(X)$. Also, the number of tasks used in the learning phase (n) is the same as the size of the learned-parameter set $|\mathcal{B}| = B$. Hence, we can express the bounds for the DMQTO algorithm using *Definition (1)* as follows:

Corollary 1. Let T_α and T_β represent the number of learning steps of step-size α and β respectively and let B and X represent the number of meta-learning environments and size of replay batch (D_i^X) used per task respectively, the meta-generalization bounds for DMQTO can be expressed as:

$$\mathbb{B} \subseteq \mathcal{O}\left(T_\beta\sqrt{\frac{\ln(1/\delta)}{B}} + \frac{T_\alpha}{X}\right) \tag{37}$$

It is evident from the bounds expressed in (37) that a larger sample (B) of meta-learning tasks should result in better generalization with lower transfer risk. It can also be noted that as $B \to \infty$, the generalization bound does not reduce to zero, instead, it is bounded by the batch-size (X) used by the inner-learning algorithm (inner loop). This implies that the DMQTO algorithm will always have a non-zero gap that arises due to within-task sample complexity in the task-learning phase.

While using a decremented loss threshold in the task-learning phase, it should be noted that as $\mathcal{L}_{min} \to 0$, the agent will require more batches of experience to fine-tune its Q-value estimates, causing $X \to \infty$. Although this suggests better generalization, it might be not possible to reduce the loss all the way to zero. Hence, we assume that the loss threshold tends to move towards an arbitrarily small value $(\mathcal{L}_{min} \to \mathcal{L}_\mu)$.

D. Deployment Phase

In the deployment phase, an application instance first communicates the application profile, user preferences, and device information to the central offloading manager which in turn initializes a new DQN with meta parameters θ and train it in the current environment to obtain optimal parameters θ_i. The fine-tuned DQN $(Q_i : \theta_i)$ is then sent to the user device and used for inference until any changes in the user preferences or device parameters are detected, after which the application requests a new policy in the changed environment. The deployment algorithm is demonstrated in the algorithm [2].

Algorithm 2: Deployment Algorithm.

```
Require: Hyper-Parameters $(\alpha, X)$
1: Initialize a new DQN $(Q_i : \theta_i) \leftarrow (Q_\theta : \theta)$
2: Initialize moment vectors $m \leftarrow 0$ and $v \leftarrow 0$
3: while not converged do
4:    Explore the current environment using $\epsilon$-greedy
      strategy and store the experience in the replay buffer
5:    Sample a batch of $X$ transitions $(D_i^X)$ from the replay
      buffer for learning
6:    Calculate Q-Updates $\tilde{Q}_i(S_x, A_x)$ from experience
      $(D_i^X)$ using (22)
7:    Obtain loss $\mathcal{L}_X^i$ using (27) and update moment vectors
      $m$ and $v$ using (29) and (30)
8:    Updated parameters, $\theta_i \leftarrow \theta_i - \alpha m/(\sqrt{v} + \mu)$
9: end while
10: Output $\theta_i$ as the learnt parameters
```

TABLE II
ENVIRONMENT PARAMETERS

Parameter	Range of Values
$V_E(l_{iot})$	2.0 mJ
$V_E(l_{edge})$	1.0 mJ
$V_E(l_{cloud})$	0.5 mJ
$D_E(l_{iot})$	0.1 mJ
$D_E(l_{edge})$	0.05 mJ
$D_E(l_{cloud})$	0.02 mJ
$V_R(l_{iot})$	$(0.010, 0.015)$ mS
$V_R(l_{edge})$	$(0.005, 0.009)$ mS
$V_R(l_{cloud})$	$(0.002, 0.005)$ mS
$D_R(l_{iot}, l_{edge})$	$(2.0, 2.5)$ mS
$D_R(l_{edge}, l_{edge})$	$(1.0, 1.5)$ mS
$D_R(l_{edge}, l_{cloud})$	$(1.0, 1.5)$ mS
$D_R(l_{cloud}, l_{cloud})$	$(0.5, 1.0)$ mS

During the deployment phase, we start with a learning rate $\alpha = 0.01$ and gradually reduce it as the loss becomes smaller. We also decay the exploration probability ϵ as the training progresses causing the agent to take greedy actions during later phases of training. Reducing exploration probability in such a way causes the Q-value estimates of greedy paths to improve and stabilize as the training process nears its completion. We use a linearly decaying exploration probability (ϵ) and an exponentially decaying learning rate (α) in the deployment phase.

V. EXPERIMENTAL RESULTS

In this section, we present the experimental results and performance evaluation of the proposed DMQTO algorithm. All the experiments were conducted on an Intel i7-6700 processor with 16 GB of memory and were implemented using python-3.8 with PyTorch-1.10.1+cu102 and TensorFlow-2.9.1 for deep learning.

A. Experimental Setup

We set up three experiments to measure the overall performance of the DMQTO algorithm, each having a different edge-cloud network topology and a group of application profiles. To simulate a dynamic environment, we first define a joint uniform distribution $(\mathcal{P})$ over all the environment parameters that can be randomized.

$$\mathcal{P} \sim \{D_R, V_R, W, \delta_e, \delta_t\}$$

A set of random environments $(\mathcal{T})$ is sampled from a uniform distribution of environments $(\mathcal{P})$ for each task-learning phase. In such a distribution, each of the parameters in $\mathcal{P}$ is chosen uniformly within a predefined range of values. The range of each environment parameter is presented in Table II.

The set of workflow DAGs (W) is generated randomly for each application based on its profile and user preference while δ_e and δ_t are chosen based on all the modes of operation that the application can support. We consider different values of node-density (G_n), height (G_h) and maximum branch factor (G_b) across multiple applications and assume that the task sizes (v_i) take values in range $[10^6, 10^7]$ CPU cycles, whereas data dependencies $(d_{i,j})$ take values within the range of $[10, 30]$ MB. The cost per unit energy C_E is set to 1.34 and the cost per unit time C_T is set to 0.2. A summary of experimental parameters is presented in Table III.

TABLE III
EXPERIMENTAL PARAMETERS

Experiment	$\mathcal{E}$	$\mathcal{C}$	G_n	G_h	G_b
A	1	1	$(5, 10)$	$(2, 10)$	3
B	3	2	$(10, 15)$	$(3, 15)$	6
C	5	2	$(15, 20)$	$(4, 20)$	8

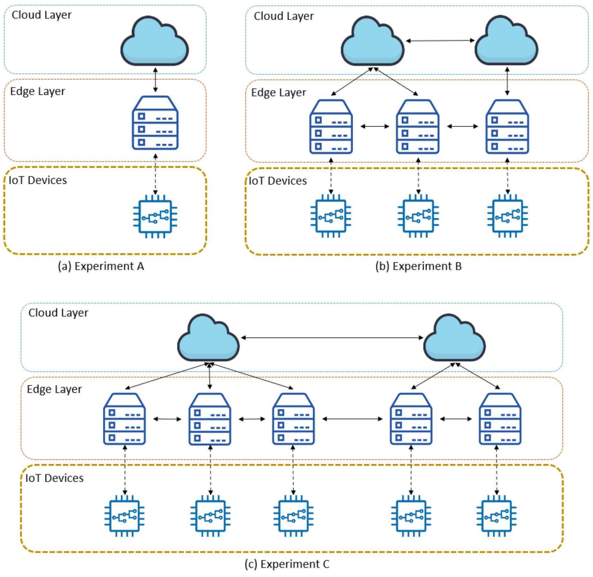

Fig. 7. Edge-cloud network architecture used in experiments.

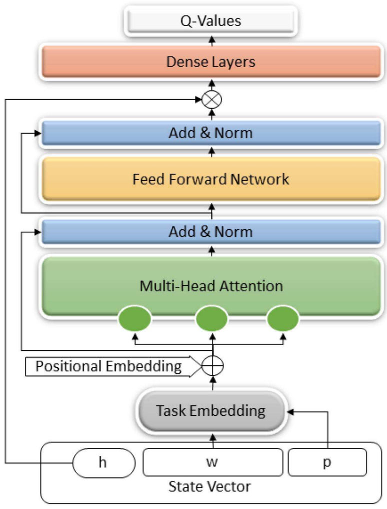

Fig. 8. Multi-head attention-based DQN architecture.

As shown in Fig. 7, we consider three different edge-cloud network architectures across experiments A, B, and C with varying numbers of Edge $(\mathcal{E})$ and Cloud $(\mathcal{C})$ servers.

To learn the Q-function, we employ an ensemble of multi-head attention-based encoders followed by a fully-connected dense neural network containing four layers consisting of 256 neurons each, with ELU (Exponential Linear Unit) activation at each layer as shown in Fig. 8. The state vector is transformed into a sequence using an appropriate embedding before it is fed to the

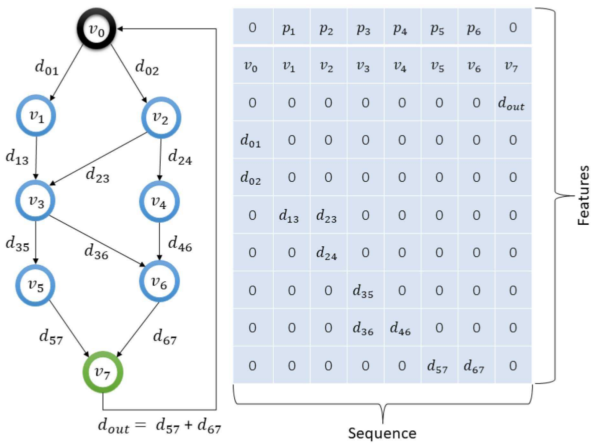

Fig. 9. Task-Embedding for a sample workflow DAG: each column represents a single time-step in the sequence while the rows represent the features: v_i represents the task size, $d_{i,j}$ represents the data-dependency and p_i represents the offloading location.

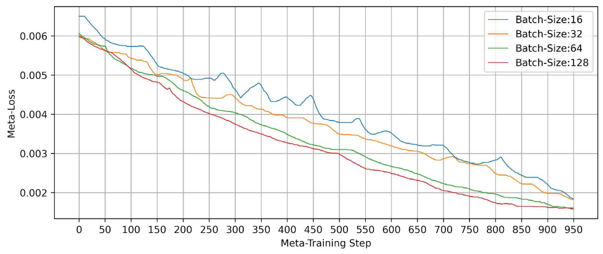

Fig. 10. Meta-Training Loss in experiment A for various batch sizes (B).

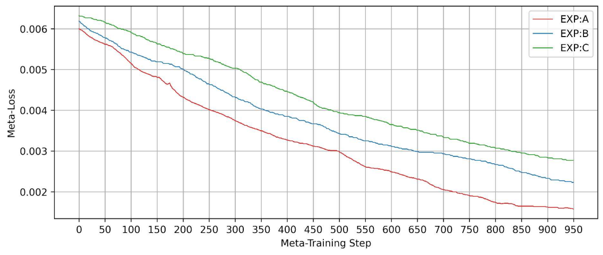

Fig. 11. Effect of scaling on meta-training: loss across experiments of increasing scale with fixed meta batch size $(\mathcal{B} = 128)$.

DQN. This is achieved by defining a topological ordering on the nodes of the workflow DAG. If two or more tasks are at the same height, the task with higher task size (V_i) is ordered before tasks with a lower task size. As shown in Fig. 9, the features of each task in the sequence include its task size (v_i), data dependency to other tasks $(d_{i,j})$ and its partial offloading location (p_i). The time-step information (h) is appended to the encoder output i.e., after the feed forward and normalization layers.

B. Performance Evaluation

The convergence of the meta-training phase with four different batch sizes (B) is shown in Fig. 10. It is noted that using a larger batch size in the meta-learning phase results in better minimization of the meta-loss and takes a lower number of learning

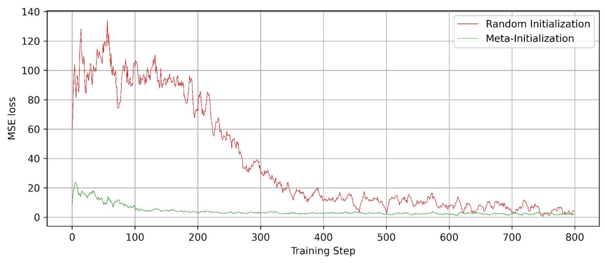

Fig. 12. Training loss comparison for meta-initialized and randomly initialized parameters during deployment phase.

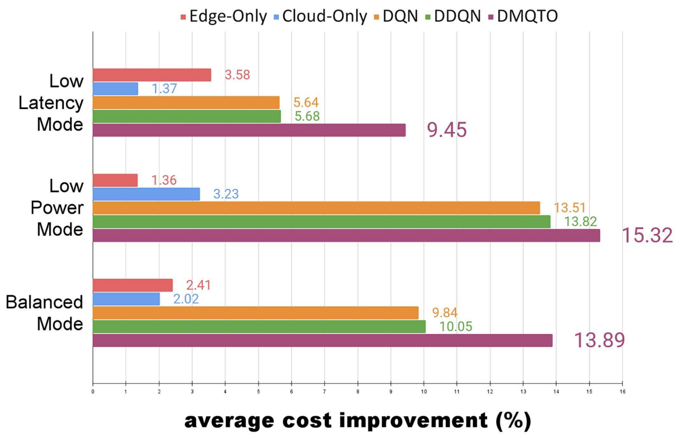

Fig. 13. Comparison of average cost improvement percentage $(\mathcal{J})$ for different offloading schemes under 3 modes of operation for the experiment (C).

steps to converge. Fig. 11 shows the effect of environment scale on meta-convergence with fixed meta batch size $\mathcal{B}$.

The convergence performance of the meta-initialization method is shown in Fig. 14. We compare the convergence performance of *meta-initialization* method with two other initialization methods:

- *Pre-Trained:* a DQN trained on random environments
- *Random Initialization:* a DQN initialized randomly

For validation, we use a set of *frozen* environments containing fixed workflow-DAG with topology similar to that in the corresponding training environments. We define the *average validation cost improvement* $(\mathcal{V})$ in the total offloading cost for the currently trained scheme $(\mathcal{S}_\theta)$ over no-offloading scheme $(\mathcal{S}_0)$ for the set of validation workflows $(\mathcal{W}_v)$, defined as:

$$\mathcal{V}_{(\theta)} = \frac{1}{|\mathcal{W}_v|} \sum_{w \in \mathcal{W}_v} \frac{U(w, p_0) - U(w, p_\theta)}{U(w, p_0)} \tag{38}$$

where, $U(w, p_0)$ represents the cost achieved by no-offloading scheme $(\mathcal{S}_0)$ and $U(w, p_\theta)$ represents the cost achieved for the offloading solution produced by currently trained scheme $(\mathcal{S}_\theta)$.

The validation cost improvement captures the reduction in offloading cost achieved by offloading solutions produced by the trained scheme over the no-offloading scheme. We use a single validation set for each experiment which consists of 20 different random environments with frozen parameters $\mathcal{P}$ where each environment has 100 unique validation workflows.

As shown in Fig. 14, the DMQTO meta-initialization method allows for faster convergence in a fewer number of steps

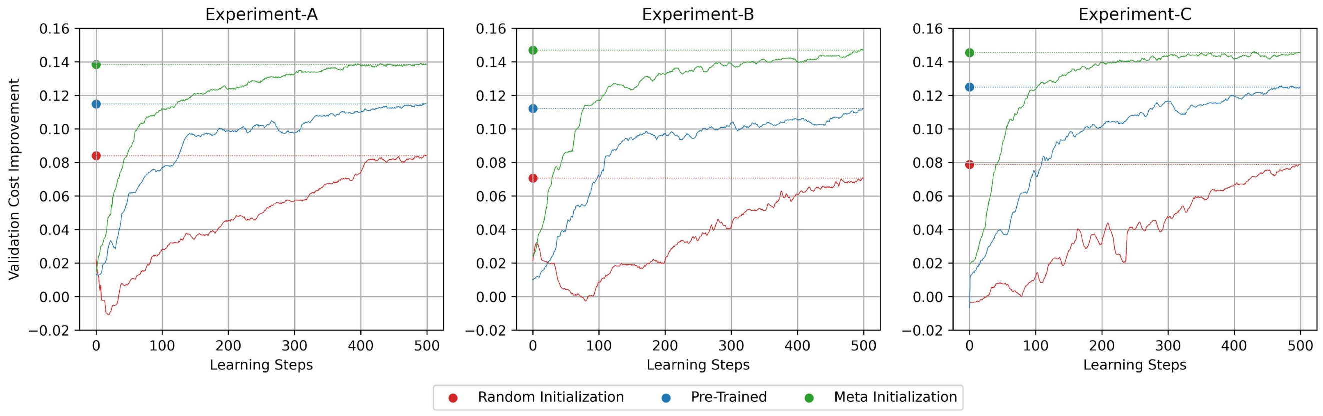

Fig. 14. Validation cost improvement for policy initialization methods.

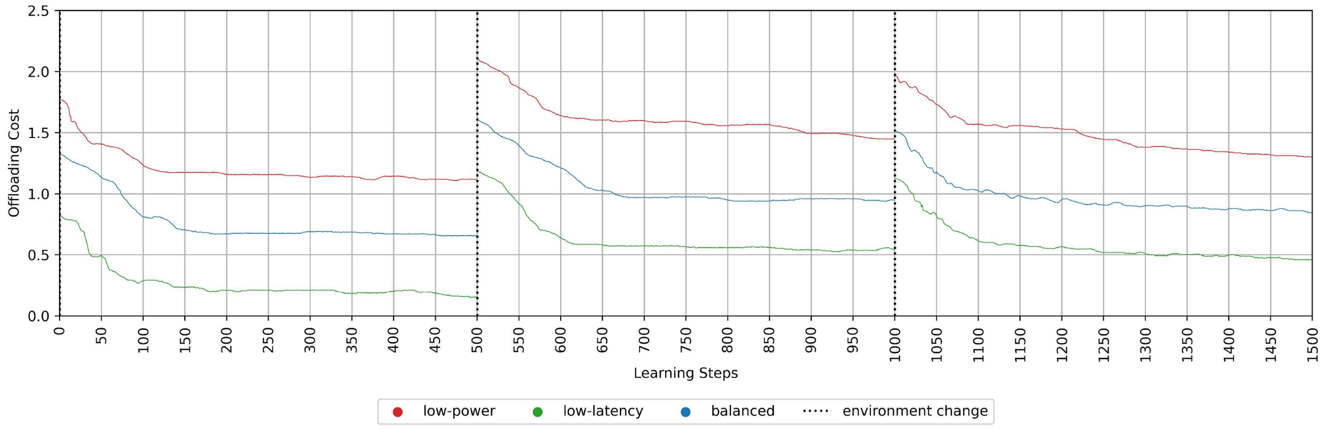

Fig. 15. Offloading cost reduction for an application running in 3 different modes of operation in changing environments.

as compared to the other two initialization methods. It can be noted from Fig. 12 that the loss observed by DQNs using the meta-initialization method is quite low as compared to random-initialization. This allows the meta-DQNs to converge faster, requiring fewer learning steps to reach the same level of performance when compared to other initialization methods.

To observe the effect of environment changes on offloading cost, we consider a single application running in three different modes of operation and subject it to random environment changes. The environment parameters are chosen randomly from within the range as defined in Table II. Fig. 15 shows the offloading cost as the DQN is being trained using the deployment algorithm over three episodes of environment randomization.

To measure the performance of trained DQNs, we use baselines obtained using the fine-tuned multi-layer perceptron (MLP) based DQN [22] and Double-DQN [23] algorithms. We also compare the results with two other non-intelligent offloading schemes namely, *Edge-Only offloading* and *Cloud-Only offloading*. To compare the performance of different offloading schemes under each mode of operation, we define the *average cost improvement percentage* $(\mathcal{J})$ in the total offloading cost for a given scheme $(\mathcal{S})$ over No-Offloading scheme $(\mathcal{S}_0)$ for a set of workflows $(\mathcal{W})$, defined as:

$$\mathcal{J}_{(\mathcal{S})} = \frac{100}{|\mathcal{W}|} \sum_{w \in \mathcal{W}} \frac{U(w, p_0) - U(w, p)}{U(w, p_0)} \tag{39}$$

where, $U(w, p_0)$ represents the cost achieved by no-offloading scheme $(\mathcal{S}_0)$ and $U(w, p)$ represents the cost achieved for the offloading solution produced by current scheme $(\mathcal{S})$.

As seen in Fig. 13, the average cost improvement (%) in total offloading cost over the no-offloading scheme is the largest in the DMQTO algorithm which suggests that DMQTO achieves lower cost than any other scheme which is a direct consequence of faster convergence.

We have also compared our algorithm against compatible meta-learning methods in the existing literature such as MRLCO [21], MELO [49], and MR-DRO [50], all of which use first-order meta-learning. The comparison is based on a restricted environment model since the existing schemes do not capture the scale of the environment and/or the application task model (DAG-based) as used in our experiments. Table IV captures the difference between various existing meta-learning-based methods.

TABLE IV
DIFFERENCES IN EXISTING META-LEARNING BASED METHODS

Method	MRLCO	MELO	MR-DRO	DMQTO (ours)
Optimization parameter	execution time	execution time	execution time, energy consumption	execution time, energy consumption
RL-Algorithm (inner loop)	Proximal Policy Optimization	Q-Learning	Q-Learning	Q-Learning
No. of offloading locations	fixed (2)	fixed (2)	fixed (3)	variable ($\geq$3)
Task-model	DAG	Sequence	Sequence	DAG
Neural Network architecture	Seq2Seq LSTM	MLP	MLP	Multi-Head attention encoder

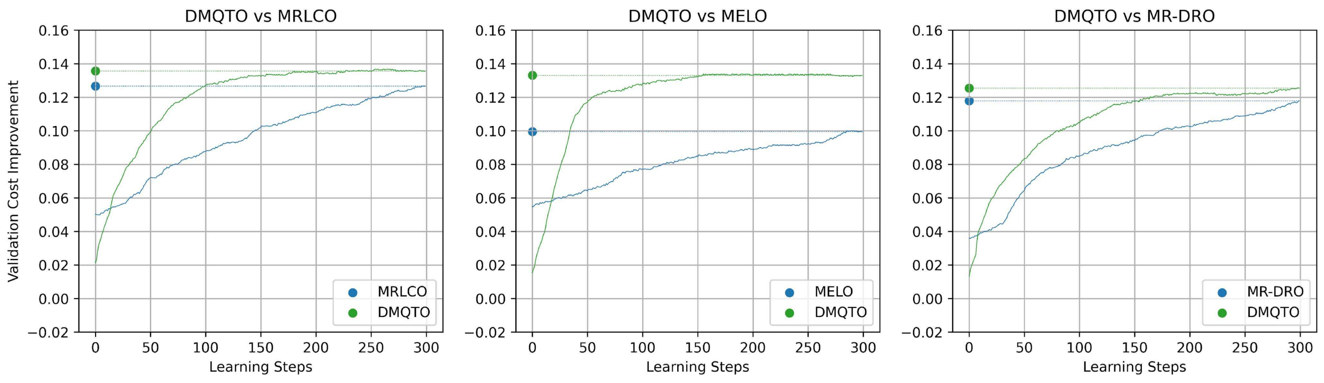

Fig. 16. Convergence performance under different meta-initialized methods.

For comparison against the MRLCO and MELO schemes, we consider only the low-latency mode with a specialized environment that contains only one edge server since both of these schemes consider binary offloading and optimize only the execution time of tasks. Furthermore, the MELO scheme considers sequential tasks only which can be represented by DAGs with a maximum branch factor of 1. For comparison with the MR-DRO scheme, we consider a single edge and a single cloud server with sequential tasks only. Fig. 16 compares the convergence performance of meta-initialized networks using the MRLCO, MELO, and MR-DRO schemes over the DMQTO scheme. The DMQTO scheme performs better than the MRLCO scheme in the early stages of fine-tuning which can be attributed to the smaller size of the multi-head attention-based DQN used in the DMQTO scheme as opposed to a larger seq2seq LSTM encoder-decoder architecture used in the MRLCO scheme. However, given enough training time, both schemes can achieve equivalent performance. The MELO scheme is designed for binary-offloading for the sequential task models instead of the DAG-based task model and uses simple multi-layer perceptron (MLP) based DQN. Our scheme performs better than MELO because of the use of multi-head attention-based architecture that can better capture sequential data. The same can be stated for MR-DRO which also uses MLP-based DQN and is designed for sequential task models instead of DAG-based task models.

C. Computational Complexity

Due to inherent randomness in Deep RL algorithms, it is difficult to capture the time complexity of such algorithms accurately. Table V shows the average time for our deployment algorithm (Algorithm [2]) for random initialized, pre-trained initialized, and meta-initialized DQN for all three experiments.

TABLE V
AVERAGE TRAINING TIME (IN MINUTES) FOR THE DEPLOYMENT ALGORITHM

Experiment	Random-Initialization	Pre-trained	Meta-Initialization
A	12.66	8.95	5.06
B	20.32	13.33	8.41
C	31.87	20.52	12.42

The experiments are conducted on an Intel i7-6700 processor with 16 GB of memory. It can be noted that the CPU time is comparatively less for meta-initialization as compared to other initialization methods.

Modern RL libraries provide vectorized environments and parallel implementation of RL algorithms that can significantly improve training time. However, we do not employ parallelism or use vectorized environments for training. The results presented in this paper are based on simple non-vectorized environments without employing any parallelism.

VI. CONCLUSION

In this article, we proposed an improved framework for formulating multi-task offloading problems in dynamic edge-cloud systems. In our formulation, we used directed acyclic graphs (DAGs) to model the application workflows and consider that the offloading locations span across multiple edge and cloud servers. We presented the DMQTO algorithm that uses meta-learning techniques to improve the learning process in dynamic edge-cloud environments by reducing the time required to train an optimal offloading policy. Through the experiments, it was observed that the proposed DMQTO algorithm consistently achieved the lowest offloading cost in a variety of dynamic scenarios as compared to other methods.

In recent years, resource scheduling in edge computing has attracted widespread interest from industry and academia. In the future, we plan to expand the DMQTO framework to include resource scheduling as well. In addition, we shall try to improve the meta-learning process so that it can further reduce the training time and improve overall efficiency.

REFERENCES

[1] Y. Ai, M. Peng, and K. Zhang, "Edge computing technologies for the Internet of Things: A primer," *Digit. Commun. Netw.*, vol. 4, no. 2, pp. 77–86, 2018.

[2] D. Xu, T. Li, Y. Li, X. Su, S. Tarkoma, and P. Hui, "A survey on edge intelligence," 2003, *arXiv:2003.12172*.

[3] H. Wu, Y. Sun, and K. Wolter, "Energy-efficient decision making for mobile cloud offloading," *IEEE Trans. Cloud Comput.*, vol. 8, no. 2, pp. 570–584, Second Quarter 2020.

[4] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surv. Tut.*, vol. 19, no. 4, pp. 2322–2358, Fourth Quarter 2017.

[5] L. Huang, S. Bi, and Y. J. 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.

[6] M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, and D. Hassabis, "Reinforcement learning, fast and slow," *Trends Cogn. Sci.*, vol. 23, no. 5, pp. 408–422, 2019.

[7] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[8] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Trans. Signal Inf. Process. Netw.*, vol. 1, no. 2, pp. 89–103, Jun. 2015.

[9] K. Peng et al., "Joint optimization of service chain caching and task offloading in mobile edge computing," *Appl. Soft Comput.*, vol. 103, 2021, Art. no. 107142.

[10] P. Shu et al., "eTime: Energy-efficient transmission between cloud and mobile devices," in *Proc. IEEE INFOCOM*, 2013, pp. 195–199.

[11] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 12, pp. 3590–3605, Dec. 2016.

[12] G. Zhang, W. Zhang, Y. Cao, D. Li, and L. Wang, "Energy-delay tradeoff for dynamic offloading in a mobile-edge computing system with energy harvesting devices," *IEEE Trans. Ind. Inform.*, vol. 14, no. 10, pp. 4642–4655, Oct. 2018.

[13] H. Wu, K. Wolter, P. Jiao, Y. Deng, Y. Zhao, and M. Xu, "EEDTO: An energy-efficient dynamic task offloading algorithm for blockchain-enabled IoT-edge-cloud orchestrated computing," *Internet Things J.*, vol. 8, no. 4, pp. 2163–2176, Feb. 2021.

[14] H. Wu and K. Wolter, "Stochastic analysis of delayed mobile offloading in heterogeneous networks," *IEEE Trans. Mobile Comput.*, vol. 17, no. 2, pp. 461–474, Feb. 2018.

[15] M. Li, Q. Wu, J. Zhu, R. Zheng, and M. Zhang, "A computing offloading game for mobile devices and edge cloud servers," *Wireless Commun. Mobile Comput.*, vol. 2018, p. 10, 2018.

[16] M. Goudarzi, H. Wu, M. 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.

[17] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," *IEEE Commun. Surv. Tut.*, vol. 22, no. 2, pp. 869–904, Feb. 2020.

[18] H. Wu, Z. Zhang, C. Guan, K. Wolter, and M. Xu, "Collaborate edge and cloud computing with distributed deep learning for smart city Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 9, pp. 8099–8110, Sep. 2020.

[19] Y. Kang et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.

[20] Z. Zhang, F. R. Yu, F. Fu, Q. Yan, and Z. Wang, "Joint offloading and resource allocation in mobile edge computing systems: An actor-critic approach," in *Proc. IEEE Glob. Commun. Conf.*, 2018, pp. 1–6.

[21] J. J. Wang, G. Hu, A. Y. MinZomaya, 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.

[22] V. Mnih et al., "Playing atari with deep reinforcement learning," 2023, *arXiv:1312.5602*.

[23] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 2094–2100.

[24] T. Q. Dinh, Q. D. La, T. Q. Quek, and H. Shin, "Learning for computation offloading in mobile edge computing," *IEEE Trans. Commun.*, vol. 66, no. 12, pp. 6353–6367, Dec. 2018.

[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 Things J.*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019.

[26] J. Wang, J. Hu, G. Min, W. Zhan, Q. Ni, and N. Georgalas, "Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning," *IEEE Commun. Mag.*, vol. 57, no. 5, pp. 64–69, May 2019.

[27] B. Cao, L. Zhang, Y. Li, D. Feng, and W. Cao, "Intelligent offloading in multi-access edge computing: A state-of-the-art review and framework," *IEEE Commun. Mag.*, vol. 57, no. 3, pp. 56–62, Mar. 2019.

[28] Z. Wang, Z. Zhao, G. Min, X. Huang, Q. Ni, and R. Wang, "User mobility aware task assignment for mobile edge computing," *Future Gener. Comput. Syst.*, vol. 85, pp. 1–8, 2018.

[29] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[30] W. Sun, J. Liu, and H. Zhang, "When smart wearables meet intelligent vehicles: Challenges and future directions," *IEEE Wireless Commun.*, vol. 24, no. 3, pp. 58–65, Jun. 2017.

[31] J. Lee, H. Ko, J. Kim, and S. Pack, "Data: Dependency-aware task allocation scheme in distributed edge clouds," *IEEE Trans. Ind. Informat.*, vol. 16, no. 12, pp. 7782–7790, Dec. 2020.

[32] V. De Maio and I. Brandic, "First hop mobile offloading of dag computations," in *Proc. 18th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2018, pp. 83–92.

[33] Y. Han, Z. Zhao, J. Mo, C. Shu, and G. Min, "Efficient task offloading with dependency guarantees in ultra-dense edge networks," in *Proc. IEEE Glob. Commun. Conf.*, 2019, pp. 1–6.

[34] C. Shu, Z. Zhao, Y. Han, G. Min, and H. Duan, "Multi-user offloading for edge computing networks: A dependency-aware and latency-optimal approach," *IEEE Internet Things J.*, vol. 7, no. 3, pp. 1678–1689, Mar. 2019.

[35] G. Qu, H. Wu, R. Li, and P. Jiao, "DMRO: ADeep meta reinforcement learning-based task offloading framework for edge-cloud computing," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 3, pp. 3448–3459, Sep. 2021.

[36] J. Chen, Y. Yang, C. Wang, H. Zhang, C. Qiu, and X. Wang, "Multitask offloading strategy optimization based on directed acyclic graphs for edge computing," *IEEE Internet Things J.*, vol. 9, no. 12, pp. 9367–9378, Jun. 2022.

[37] A. Nichol, J. Achiam, and J. Schulman, "On first-order meta-learning algorithms", 2018, *arXiv:1803.02999*.

[38] J. Baxter, "A model of inductive bias learning," *J. Artif. Intell. Res.*, vol. 12, pp. 149–198, 2000.

[39] M. Al-Shedivat, L. Li, E. Xing, and A. Talwalkar, "On data efficiency of meta-learning," 2021, *arXiv:2102.00127*.

[40] O. Bousquet and A. Elisseeff, "Stability and generalization," *J. Mach. Learn. Res.*, vol. 2, pp. 499–526, 2002.

[41] A. Maurer, "Algorithmic stability and meta-learning," *J. Mach. Learn. Res.*, vol. 6, pp. pp. 967–994, 2005.

[42] , "Christopher JCH Watkins and Peter Dayan," *Q-Learn. Mach. Learn.*, vol. 6, pp. 967–994, 2005.

[43] Z. Ning, P. Dong, X. Kong, and F. Xia, "A cooperative partial computation offloading scheme for mobile edge computing enabled Internet of Things," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4804–4814, Jun. 2019.

[44] Y. Wang, M. Sheng, X. Wang, and J. Li, "Cooperative dynamic voltage scaling and radio resource allocation for energy-efficient multiuser mobile edge computing," in *Proc. IEEE Int. Conf. Commun.*, 2018, pp. 1–6.

[45] S. Yu, X. Chen, L. Yang, D. Wu, M. Bennis, and J. Zhang, "Intelligent edge: Leveraging deep imitation learning for mobile edge computation offloading," *IEEE Wireless Commun.*, vol. 27, no. 1, pp. 92–99, Feb. 2020.

[46] 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.*, vol. 27, pp. 1123–1130, 2022.

[47] M. Ren et al., "Meta-learning for semi-supervised few-shot classification," 2018, *arXiv:1803.00676*.

[48] J. Wang, J. Hu, G. Min, W. Zhan, A. Y. Zomaya, and N. Georgalas, "Dependent task offloading for edge computing based on deep reinforcement learning," *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2449–2461, Oct. 2022.

[49] L. Huang, L. Zhang, S. Yang, L. P. Qian, and Y. Wu, "Meta-learning based dynamic computation task offloading for mobile edge computing networks," *IEEE Commun. Lett.*, vol. 25, no. 5, pp. 1568–1572, May 2021.

[50] Z. Zhang, N. Wang, H. Wu, C. Tang, and R. Li, "MR-DRO: A fast and efficient task offloading algorithm in heterogeneous edge/cloud computing environments," *IEEE Internet Things J.*, vol. 10, no. 4, pp. 3165–3178, Feb. 2023.

[51] E. Wang, P. Dong, Y. Xu, D. Li, L. Wang, and Y. Yang, "Distributed game-theoretical task offloading for mobile edge computing," in *Proc. IEEE 18th Int. Conf. Mobile Ad Hoc Smart Syst.*, 2021, pp. 216–224.

[52] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., London, U.K.: Pearson Education, 2011.

[53] I. Gupta, A. Choudhary, and P. K. Jana, "Generation and proliferation of random directed acyclic graphs for workflow scheduling problem," in *Proc. 7th Int. Conf. Comput. Commun. Technol.*2017 pp. 123–127.

[54] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *Proc. 11th 9th Int. Conf. Mobile Syst. Appl. Serv.*, 2011, pp. 43–56.

Nelson Sharma received the MTech degree in computer science and engineering from IIEST, Shibpur, India, in 2020. He is currently working toward the PhD degree from the Department of Computer Science and Engineering, Indian Institute of Technology(IIT) Patna, India. His current research interests include Reinforcement-learning, Cloud, and Edge computing.

Aswini Ghosh received the BTech and MTech degrees from Maulana Abul Kalam Azad University in Computer Science and Engineering, in 2005 and 2008 respectively. He is currently working toward the PhD degree in the department of Computer Science and Engineering from the Indian Institute of Technology(IIT) Patna, India. His current research interests include Cloud computing, Edge computing, 5G Slicing, and Wireless Networks. He has ten years experience of working in a 3G data center in India.

Rajiv Misra (Senior Member, IEEE) received the MTech degree in computer science and engineering from the Indian Institute of Technology (IIT) Bombay and the PhD degree in mobile computing from the Indian Institute of Technology (IIT) Kharagpur. He presently has a position as professor and CSE department head at the Indian Institute of Technology (IIT) Patna in India. His current research focuses on reinforcement learning for distributed cloud and edge computing, designing distributed algorithms for wireless networks. In addition to publishing more than 60 papers in prestigious journals and conferences, a text book, and edited books, he has made a substantial contribution to these fields. With almost 1241 citations, his h-index is 16, which is high. He has published articles in journals like *IEEE Transactions on Parallel and Distributed Systems* and *IEEE Transactions on Mobile Computing*, among others.

Sajal K. Das (Fellow, IEEE) is a Curators' Distinguished Professor of Computer Science and Daniel St. Clair Endowed Chair with the Missouri University of Science and Technology, Rolla, USA. His research interests include wireless and sensor networks, IoT, cyber-physical systems, smart environments (smart city, smart grid, smart transportation, smart agriculture, and smart health), edge and cloud computing, applied machine learning, data science, and cyber-physical security. He has published extensively in these areas with more than 700 research articles in high-quality journals and refereed conference proceedings. He holds 5 US patents and co-authored 4 books. His h-index is 98 with more than 38,700 citations according to Google Scholar. He is a recipient of 14 Best Paper Awards in prestigious conferences and numerous awards for teaching, mentoring, and research. He has graduated 50 Ph.D. students and mentored more than a dozen postdoctoral fellows. He is a Distinguished alumni of the Indian Institute of Science, Bangalore.