TPTO: A Transformer-PPO based Task Offloading Solution for Edge Computing Environments

Niloofar Gholipour*, Marcos Dias de Assuncao*, Pranav Agarwal*, Julien Gascon-Samson*, Rajkumar Buyya†
*Dept. of Software Engineering and IT, École de Technologie Supérieure, Univ. of Quebec, Montreal, Canada
{niloofar.gholipour, pranav.agarwal}.1@ens.etsmtl.ca, {marcos.dias-de-assuncao, julien.gascon-samson}@etsmtl.ca
†CLOUDS Lab, School of Computing and Information Systems, The Univ. of Melbourne, Australia
rbuyya@unimelb.edu.au

***Abstract*—Emerging applications in healthcare, autonomous vehicles, and wearable assistance require interactive and low-latency data analysis services. Unfortunately, cloud-centric architectures cannot fulfill the low-latency demands of these applications, as user devices are often distant from cloud data centers. Edge computing aims to reduce the latency by enabling processing tasks to be offloaded to resources located at the network's edge. However, determining which tasks must be offloaded to edge servers to reduce the latency of application requests is not trivial, especially if the tasks present dependencies. This paper proposes a Deep Reinforcement Learning (DRL) approach called TPTO, which leverages Transformer Networks and Proximal Policy Optimization (PPO) to offload dependent tasks of IoT applications in edge computing. We consider users with various preferences, where devices can offload computation to an edge server via wireless channels. Performance evaluation results demonstrate that under fat application graphs, TPTO is more effective than state-of-the-art methods, such as Greedy, HEFT, and MRLCO, by reducing latency by 30.24%, 29.61%, and 12.41%, respectively. In addition, TPTO presents a training time approximately 2.5 times faster than an existing DRL approach.**

***Index Terms*— Edge computing, reinforcement learning, Transformers, task offloading**

1. Introduction

Edge computing, by complementing the cloud, can enable an increasing range of IoT applications that produce vast amounts of time-sensitive data requiring prompt analysis, such as in autonomous driving, healthcare, online video processing, and wearable assistance [1], [2]. In autonomous driving, for instance, latency is a critical factor in ensuring the safety of passengers and pedestrians. A minor delay in processing sensor data or making control decisions can not only degrade the users' quality of experience but also result in accidents or compromised safety. Edge computing provides computing services (e.g., base stations, access points, and edge routers) that are closer to end-users, contributing to lower the latency of application requests, their energy consumption, and the amount of data transferred to the cloud for processing [3].

Reducing the latency of IoT applications requires offloading data processing tasks to edge servers, an activity that often poses significant challenges. Offloading tasks can free constrained resources of user devices, but on the other hand, transferring data between the user devices and remote edge computing servers can impact the application latency [4]. Moreover, according to research conducted by Alibaba, around 75% of real-world applications have interdependent tasks, commonly structured as a Directed Acyclic Graph (DAG), where the vertices represent data sources, data sinks, end-users, and operators, and the edges represent data streaming from one operator to another [5], [6]. Trying to devise efficient offloading decisions for these applications can often result in NP-hard problems, which require sophisticated algorithms to address them effectively.

Several heuristics, meta-heuristics, and model-based approaches exist for offloading in edge computing, most of which are unsuitable to stochastic environments where resource availability is continuously evolving [7], [8]. Edge computing is also stochastic when considering the number of applications, the number of tasks in an application, their arrival rate, their dependencies, and their resource requirements [9]. DRL with policy optimization is a promising approach to address these challenges and design agents interacting with the environment to learn an optimal policy, enhanced over time through trial and error [10]. DRL agents can learn a stochastic policy without having preliminary information about the environment, making them suitable for stochastic and complex systems like edge environments [7], [8], [11], [12], [13].

We formulate the task offloading decision as a binary optimization problem and propose a solution, Transformer-PPO based Task-Offloading (TPTO), which utilizes a combination of Markov Decision Process (MDP), Reinforcement Learning (RL), and Transformers [14]. While RL provides a learning mechanism to optimize offloading decisions over time, the Transformer model enhances the solution's performance by enabling it to learn from previous tasks and apply the knowledge to future offloading decisions. TPTO trains Transformers for various edge computing tasks and quickly adapts to new ones with less training time and shorter latency. Our proposed approach features Bidirectional Encoder Representations from Transformers (BERT) architecture incorporating multi-head attention, layer normalization, and feed-forward fully con-

nected layers. The predictions made by the Transformer, provided to a Softmax function, act as the actions that guide the training process in collaboration with the PPO algorithm. This results in a more efficient and effective solution. To our knowledge, this paper presents the first work that applies BERT for offloading decisions in edge environments. To validate our approach, we carry out simulations using synthetic DAGs that reflect real-world applications with dependent tasks and network topologies with multiple wireless transmission rates. Experimental results demonstrate our approach's effectiveness in optimizing the offloading problem.

The main contributions of this work are: A novel latency-aware task offloading approach, TPTO, that leverages the Transformer model that quickly adapts to stochastic edge environments; and a new policy that jointly uses Transformers and an actor-critic framework to determine the best action for task offloading – i.e., offloaded to the edge or processed locally to minimize end-to-end latency.

The paper is structured as follows: Section 2 describes the problem and presents a formulation. Section 3 presents TPTO, whereas Section 4 analyzes and compares its efficiency against state-of-the-art techniques. Section 5 reviews related work, and Section 6 concludes the paper and discusses future work.

2. Problem Description and Formulation

A real-time object detection system presents a typical example of an application that can benefit from computation offloading to edge computing servers (Figure 1). In this scenario, a user device often captures a video stream from a camera and aims to detect and recognize objects from the video feed in real-time. This scenario reflects, for instance, applications in facial recognition [7] and pest bird detection systems [15]. The user device can carry out data pre-processing and execute a lightweight object detection model locally (e.g., identifying some features), but the type of computations it can perform will largely depend on the system status, the available resources, and their constraints. Alternatively, some of the computations can be offloaded to an edge server.

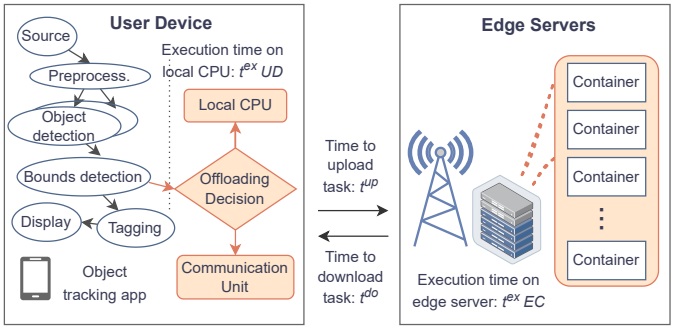


Figure 1. System model with sample application.

An application A is a DAG $G = (V, E)$ where each vertex $v_i \in V$ represents a task and each directed edge $e(v_i, v_j) \in E$ is a dependence constraint in which task v_i must complete before task v_j starts. *Entry tasks* are tasks without parent tasks, whereas *exit tasks* or *sinks* are tasks without children. The computation of task v_i corresponds to the number of CPU cycles needed for its execution, given by c_i. Moreover, we define as $data_i^{up}$ and $data_i^{do}$ the amount of data required to upload and download, respectively, task v_i to/from an edge server.

The computing capacity of a resource m_j (user device or edge server), denoted as cs_j, reflects its clock speed times the number of cores available in the system. A user device is associated with a container providing the computing and network resources that an application requires. We consider that the containers share the computing resources equally, such that the capacity of a container on an edge server m_j is $cs_{ct} = cs_j/k$, where k is the number of users in m_j. This approach of equal resource allocation ensures a fair distribution of the computing capacity to each user, thereby promoting efficient utilization of the resources within the edge computing environment. The user device can execute a task locally or offload its computation to an edge server via *wireless channels*. A wireless channel's uplink and downlink transmission rates are r_{up} and r_{do}. Three steps are required to offload a task v_i to an edge server m_j: first, the user device sends the task to the edge server via a wireless channel. Second, the edge server executes the task. Finally, the edge server sends the execution results back to the user's device. The overall latency for a task is influenced by both the task's requirements and the current system status. Hence, the total time involved in offloading task v_i to edge server m_j encompasses the time to upload the task (t_i^{up}), the time to execute the task on the edge server (t_i^{ex}), and the time to download the resulting data back to the user device (t_i^{do}). This can be mathematically expressed as:

$$t_i^{up} = data_i^{up}/r_{up}, \quad t_i^{ex} = c_i/cs_{ct}, \quad t_i^{do} = data_i^{do}/r_{do} \qquad (1)$$

When offloaded to an edge server, the overall end-to-end latency of task v_i represents the sum of the above times in (1). On the other hand, if a user device executes task v_i locally, hence using resource m_k (the user device), its latency consists only of the task execution time (*i.e.* $t_i^{ex} = c_i/cs_k$). In addition, for a task v_i scheduled for execution, we establish four task finish times, namely FT_i^{ud}, FT_i^{up}, FT_i^{ec}, and FT_i^{do}, to denote the task finish time on the user device, on the upload link, on the edge server and the download link. If task v_i runs locally on the user device, then $FT_i^{up} = FT_i^{ec} = FT_i^{do} = 0$. Otherwise, $FT_i^{ud} = 0$ if v_i is offloaded to an edge server.

Before scheduling a task v_i, all preceding tasks (*i.e.*, its parent tasks) must have been scheduled. In this way, we denote RT_i^{ud}, RT_i^{up}, RT_i^{ec}, and RT_i^{do} as the ready time, the earliest time that task v_i can be executed on a resource (user device, upload link, edge server, download link) so that the precedence constraints are maintained. Hence, for task v_i scheduled on the user device, we can calculate its ready time as:

$$RT_i^{ud} = \max_{j \in parent(v_i)} max\left\{FT_j^{ud}, FT_j^{do}\right\} \qquad (2)$$

where ***parent***(v_i) is the set of parent tasks immediately before task v_i. RT_i^{ud} is the earliest time at which all the tasks preceding v_i will have completed and produced the results that

v_i requires. When a task v_j preceding v_i is scheduled locally, then $max\{FT_j^{ud}, FT_j^{do}\} = FT_j^{ud}$; otherwise, when offloaded to the edge server, $max\{FT_j^{ud}, FT_j^{do}\} = FT_j^{do}$. Task v_i can only start executing once v_j has freed the wireless download channel.

On the other hand, if that task v_i is to be offloaded to the edge server, then its ready time on the upload channel (RT_i^{up}) is given by:

$$RT_i^{up} = \max_{j \in parent(v_i)} max\left\{FT_j^{ud}, FT_j^{up}\right\} \tag{3}$$

where RT_i^{up} is the earliest time when v_i can use the upload channel while meeting precedence constraints. When a task v_j preceding v_i is scheduled locally, then $max\{FT_j^{ud}, FT_j^{up}\} = FT_j^{ud}$; otherwise, when offloaded to the edge server, then $max\{FT_j^{ud}, FT_j^{up}\} = FT_j^{up}$. Task v_i can only start execution once v_j has freed the wireless download channel.

The ready time of a task v_i on an edge server is:

$$RT_i^{ec} = max\left\{FT_i^{up}, \max_{j \in parent(v_i)} FT_j^{ec}\right\} \tag{4}$$

where RT_i^{ec} is the earliest time v_i can execute on the edge server while respecting precedence constraints. If a task v_j preceding v_i is scheduled locally, then $FT_j^{ec} = 0$. Hence, $\max_{j \in parent(v_i)} FT_j^{ec}$ is the earliest time when all offloaded tasks preceding v_i have finished execution.

The earliest time for sending the results of task v_i back to the user device is:

$$RT_i^{do} = FT_i^{ec} \tag{5}$$

The offloading goal is to compute an offloading plan $O_n = (o_1, o_2, \ldots, o_n)$ that minimizes the latency of an application DAG $G(V, E)$, where $n = |V|$. Here, o_i denotes the offloading decision for task v_i, where o_i can be either 0 for local computation or 1 for remote computation. Before offloading, tasks are sorted by priority, as discussed later, so that O_{n-1}, for example, represents the partial offloading plan comprising all tasks from v_1 to v_{n-1}. The optimization goal is, hence, to minimize the overall *Application Latency*:

$$AL_{O_n} = max\left[max_{v_e \in \mathcal{E}}(FT_e^{ud}, FT_e^{do})\right] \tag{6}$$

where $\mathcal{E}$ is the set of exit tasks (*i.e.* tasks with no children). The equation considers the maximum task latency within a DAG to compute the overall application latency. This maximum time represents the duration of the critical path of the DAG, which is the longest path from a start task to any of the exit tasks. Table 2 summarizes the main notations used in this paper.

3. Transformer-Based Offloading Solution

This section presents our Transformer-PPO-based task offloading solution.

TABLE 1. NOTATION USED IN THIS PAPER.

Notation	Description
$G(V, E)$	Application DAG where V is the set of tasks and E the task precedence constraints
$v_i \in V$	Computing task v_i
$e(v_j, v_i) \in E$	Precedence constraint, task v_j must execute before v_i can start
$data_i^{up}$, $data_i^{do}$	Number of bytes to upload/download to/from an edge server when offloading task v_i
r_{up}, r_{do}	Transmission rates of wireless uplink and downlink channels
cs_k, cs_{ct}	Computing capacity of resource m_k, and of a container
t_i^{up}, t_i^{ex}, t_i^{do}	Time required for uploading, executing and downloading task v_i to edge server m_k
FT_i^{ud}, FT_i^{up}, FT_i^{ec}, FT_i^{do}	Finish time of task v_i on user device, uplink channel, edge server, and downlink channel
RT_i^{ud}, RT_i^{up}, RT_i^{ec}, RT_i^{do}	Earliest time when task v_i can use the user device, uplink channel, edge server, and downlink channel

3.1. Transformer-PPO based Task Offloading

In RL, an agent interacts with an environment, trying to learn a policy to take actions that maximize the accumulated reward. An MDP, commonly used to represent RL problems [16], consists of a tuple (S, A, P, R, γ), where S represents the set of possible states; A represents the action space; $P(s'|s, a)$ denotes the probability of transitioning to state s' when taking action a under the current state s; $R(s, a, s')$ represents the immediate reward received when transitioning from s to s' by taking action a; γ is a discount factor. The goal is to find a policy $\pi(s)$ that maximizes the expected cumulative reward over time. A policy network $\pi(a|s, \theta)$ takes the state s as input and outputs a probability distribution over the actions a, where θ represents the neural network parameters. Training the policy network involves finding the optimal parameters θ^* that maximize the expected cumulative reward, a process typically performed using policy gradient algorithms that seek to maximize the expected return. TPTO optimizes the policy network parameters using PPO [17]. During training, PPO uses a batch of sampled trajectories to update the network weights. The following describes the main elements of our MDP:

State S: A state comprises the task profile (CPU cycle requirements and data sizes), the DAG topologies, the wireless transmission rates, and the status of edge resources. The status of an edge resource depends on the offloading decisions for tasks preceding v_i. Hence, we can express the state combining the encoded DAG and the partial offloading plan as:

$$S = \{s_i | s_i = (G(V, E), O_i)\} \tag{7}$$

where $i \in [1, |V|]$, $G(V, E)$ represents the sequence of embedding tasks and O_i is the partial offloading plan of task v_i. We use the approach outlined in [7] to convert a DAG into a sequence of embedding tasks.

For efficient offloading, tasks receive a "rank" based on their completion time and dependencies, sequenced from lowest to highest rank. Each task is embedded with information on

its attributes and its parent-child relationships. This approach ensures optimized task scheduling, enhancing efficiency and reducing latency. Task embeddings use three vectors: one for the task's profile, one for parent tasks, and another for child tasks, with padding if the number of tasks is below the vector's length.

Action A: As the scheduling for each task is a binary choice, executing the task either on the user device or on an edge server, the action space is $A := 0, 1$, where 0 represents execution on the user device and, 1 represents offloading.

Reward function R: The objective is to minimize the total application latency, defined in Equation 6. Hence, the reward function estimates the negative increase in latency resulting from an offloading decision for a particular task: $\Delta AL_{O_i} = AL_{O_i} - AL_{O_{i-1}}$, where AL_{O_i} represents the total latency when taking a given action for task v_i and $AL_{O_{i-1}}$ represents the total latency of the partial offloading plan for the previous task.

Assume that $\pi(a_i|G(V,E), O_{i-1})$ represents the likelihood of the offloading plan O_{i-1} given the graph $G(V,E)$, we can compute $\pi(O_n|G(V,E))$ by using the chain rule of probability on each $\pi(a_i|O_{i-1}, G(V,E))$ as follows:

$$\pi(O_n|G(V,E)) = \prod_{i=1}^{n} \pi(a_i|O_{i-1}, G(V,E)) \tag{8}$$

We employ Transformers to devise our policy. Distinct from traditional Recurrent Neural Networks (RNNs), Transformers use an encoder-decoder structure, effectively addressing various RNN limitations. Typically, the encoder integrates features like embedding, multi-head attention, residual connections with normalization, feed-forward networks, and softmax. A distinguishing feature of Transformers is the incorporation of a self-attention mechanism, enhancing data dependency extraction [14]. In the context of TPTO, a Transformer processes the task embeddings from a sequence $(v_1, v_2, ..., v_n)$ of a DAG and formulates a refined representation through successive Transformer layers. Based on the output, the actor makes offloading decisions for each task. Meanwhile, the critic assesses each task's value function, with fully connected layers producing these outputs.

3.2. Implementing TPTO

As Figure 2 outlines, TPTO employs the Transformer model and PPO to update the policy network. First, the Transformer receives an observation of the environment and produces two results: the policy logits and the value function. The policy logits are passed through a softmax function to obtain a proper probability distribution of the available actions. Next, the actor network takes the Transformer's output and produces the final policy, which provides a probability distribution for the available actions. Finally, the critic network takes the Transformer's output and generates the estimated value of the current state. The advantage function captures the difference between the actual and estimated return and the estimated value of the current state.

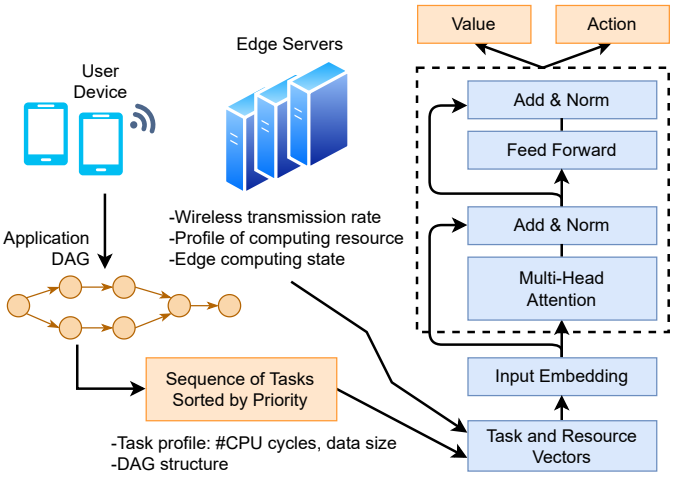


Figure 2. Overview of TPTO.

We use PPO as the policy optimization method. For a given learning task $\mathcal{T}$, PPO creates trajectories using a sample policy $\pi_{\theta_{sam}}$ and updates the target policy π_θ over multiple epochs, where θ and θ_{sam} are the parameters of the target and sample policies, respectively. At the initial epoch, $\theta = \theta_{sam}$. Then the probability ratio $r_t(\theta)$ at a time step t is:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{sam}}(a_t|s_t)} \tag{9}$$

where $s_t = G(V,E), O_t$. To update the actor's policy, PPO uses a clipped surrogate objective to avoid extensive policy updates:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{10}$$

where $\hat{A}_t$ is the advantage function at time step t, and $\hat{\mathbb{E}}$ is the average expectation over a set of samples in an algorithm that alternates between sampling and optimization [17]. As the policy and value functions share most of their parameters, facilitating mutual training, we also employ the entropy coefficient to compute the entropy bonus, added to the policy loss, to encourage exploration in the policy space. The combined objective is, therefore:

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{11}$$

where c_1 and c_2 are coefficients, $S[\pi_\theta](s_t)$ represents the entropy bonus, and $L_t^{VF}(\theta)$ is the squared-error loss: $(V_\theta(s_t) - V_t^{targ})^2$, where V is a state-value function.

The advantage function at time step t, denoted by $\hat{A}_t$, is calculated using General Advantage Estimator (GAE) [18]. GAE is a specific type of advantage function estimated as follows:

$$\hat{A}_t = \sum_{l=0}^{n-t+1} (\gamma\lambda)^k \left[r_t + \gamma V(s_{t+k+1}) - V(s_{t+k}) \right] \tag{12}$$

where λ is in the interval $(0, 1)$ and determines the equation's balance between bias and variance. We can then use gradient ascent to maximize $L^{CLIP+VF+S}(\theta)$.

Algorithm 1 Transformer-PPO based task offloading

Require: Task distribution $r(\mathcal{T})$, learning rate α
Ensure: Updated policy parameters θ
1: Randomly initialize the parameters of the policy, θ;
2: **for** iterations $k \in \{1, 2, \ldots, K\}$ **do**
3: Sample n learning tasks $\{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n\}$ from $r(\mathcal{T})$;
4: **for** each task $\mathcal{T}_i$ **do**
5: Initialize $\theta_{\text{sam}} \leftarrow \theta$
6: Sample trajectory set $S = (\tau_0, \tau_1, \ldots, \tau_n)$ from $\mathcal{T}_i$ using policy $\pi(\theta_{\text{sam}})$;
7: Calculate the advantage estimates $\hat{A}_1, \hat{A}_2, .., \hat{A}_T$;
8: Compute the policy gradient:
9: $L^{\text{TPTO}}_{\tau_{\text{sam}}}(\theta_{\text{sam}}) = \nabla_{\theta_{\text{sam}}} L^{\text{CLIP+VF+S}}(\theta_{\text{sam}})$
10: **end for**
11: Update the policy network parameters θ using Adagrad optimizer with gradients computed by the TPTO loss function with trajectory set S for m steps:
12: $\theta \leftarrow \theta + \alpha L^{\text{TPTO}}_{\tau_{\text{sam}}}(\theta_{\text{sam}})$
13: **end for**

Algorithm 1 outlines how TPTO performs the offloading decision and generates trajectories. First, the algorithm samples an n sized batch of learning tasks τ and performs the training loop for each sampled learning task. Following the completion of the training loop, the algorithm then updates the policy parameters θ using gradient ascent $\theta \leftarrow \theta + \alpha L^{TPTO}$ using Adagrad optimizer [19], where α is the learning rate of training loop.

4. Performance Evaluation

This section presents the experimental setup, the baseline algorithms, and performance evaluation results.

4.1. Experimental Setup

We evaluated the performance of TPTO by developing an event-driven simulation environment in Python using the OpenAIGym [20], similar to [7]. This approach ensured a controllable and repeatable evaluation process. We consider a cellular network whose data transmission rate varies based on the user devices' position. Also, a user device's CPU clock speed is $1GHz$, denoted by f_1. In contrast, each container in an edge server has a quota of four cores, each core running at $2.5GHz$, represented by f_s. Consequently, offloaded tasks can simultaneously use all cores, resulting in a combined CPU clock speed of $10GHz$ for each container.

We consider latency under multiple scenarios to evaluate TPTO's efficiency comprehensively in dynamic environments. We use a synthetic DAG generator tool[1] to create diverse heterogeneous DAGs that emulate real-world applications. These DAGs encompass a broad spectrum of topologies and

1. https://github.com/frs69wq/daggen

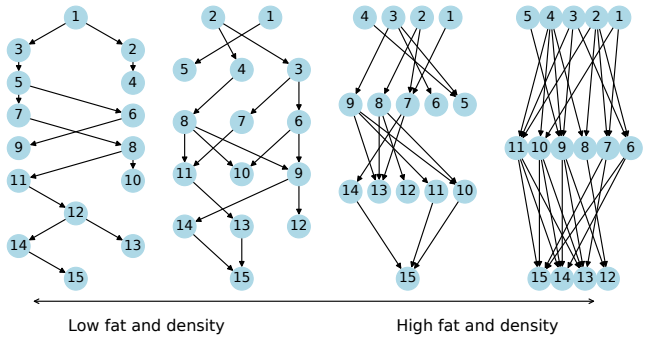


Figure 3. Examples of application DAGs with different fats and densities.

transmission rates encountered in practical scenarios. The generator receives four parameters: n, fat, $density$, and ccr. The n represents the number of tasks; fat determines the DAG's width and height; $density$ sets the number of edges between two levels of the DAG; and computation to communication ratio, ccr, specifies the ratio between tasks' communication and computation cost.

TABLE 2. TPTO'S HYPERPARAMETERS.

Hyperparameter	Value
Number of Layers	3
Num Attention Head	8
Dimension of Key Vector	1024
Dimension of Value Vector	1024
Dimension of FF network	512
Hidden Size	512
Dropout Rate	0.4
Policy Learning Rate	0.1
Valuefunc Learning Rate	0.01
Batch Size	100
Clip ratio	0.2
Activation Function	Relu
Optimization Method	Adagrad
Discount Factor	0.99
Entropy coefficient	0.5

To model the mobile network users' diverse preferences, we generated 25 DAG datasets, each dataset comprising 100 DAGs with various fat, densities, and ccr – key parameters impacting the DAG topology. Each DAG has 20 tasks, and fat and density values for each DAG are selected randomly from the set $\{0.4, 0.5, 0.6, 0.7, 0.8\}$, while ccr is chosen randomly within the range of 0.3 to 0.5. This range is representative of the computation sensitivity observed in a majority of IoT applications. The DAGs simulate diverse application preferences of a mobile user: for instance, a fatter DAG suggests a preference for more parallel tasks, while a denser DAG indicates a higher dependency between tasks, all under varying data transmission speeds. We randomly select 22 DAG sets as "training datasets" and the remaining three as "unseen testing datasets" with different DAG topologies. Figure 3 illustrates DAGs generated by the synthetic DAG generator with varying fat and density.

TPTO is implemented using Tensorflow, with 3 layers of Transformer encoders having 512 hidden units per layer and layer normalization included. Table 2 summarizes the

hyperparameters for training TPTO. To ensure the robustness of the TPTO policy, we trained it using a range of transmission rates between 4Mbps to 22Mbps, with a step size of 3Mbps. To evaluate its performance on previously unseen transmission rates and topologies, we tested the trained policy on data rates of 8.5Mbps and 11.5Mbps, not seen during training, following a similar methodology as in [7] with sampling 20 trajectories for a DAG on the dataset. In addition, as we aim to assess how TPTO performs in different dynamic scenarios, the task data size varies from 5KB to 50KB, while the CPU cycle requirements range from 10^7 to 10^8 cycles per task, as reported in [21]. Furthermore, the length of the parent/child task indices vector is 12. By testing TPTO's performance on these diverse sets of DAGs, we aim to gain insights into its ability to effectively provision network resources and meet the varying needs of mobile users.

4.2. Baseline Algorithms

We assess TPTO's performance against three state-of-the-art algorithms:

MRLCO: this algorithm, proposed by Wang *et al.* [7], integrates meta reinforcement learning and a Seq2Seq neural network. The approach focuses on modeling task offloading using meta-reinforcement learning and an offloading policy based on a custom Seq2Seq neural network.

HEFT based: this algorithm, based on the work by Lin *et al.* [22], involves prioritizing tasks using the HEFT method and scheduling each task according to its earliest estimated finish time.

Greedy Heuristic: a greedy approach considers the estimated finish time of each task to decide whether to assign a task to the user device or an edge server.

4.3. Result Analysis

Figures 4(a) and 4(b) depict the average latency of simulation results during training for TPTO and MRLCO. The results demonstrate that TPTO converges faster than MRLCO while being more stable and general, mainly due to TPTO's ability to effectively capture the diverse preferences of mobile users through its training on a wide range of network topologies and transmission rates. Figure 4(c) and 4(d) show the performance of HEFT and Greedy algorithms.

Table 3 summarizes the average latency of TPTO and the baseline algorithms. TPTO outperforms heuristic and meta-learning algorithms for the various wireless transmission rates. Overall, the Greedy algorithm has the highest latency, while TPTO achieves lower latency under various network conditions, indicating its effectiveness in provisioning network resources to meet the needs of mobile users. Moreover, distinct topologies reflect the diverse preferences of user requests in terms of dependency and parallel computing of tasks. Increasing the transmission rate can further reduce latency as offloaded tasks traverse the wireless channels faster. Overall, the results show that TPTO is a promising solution for optimizing network performance and enhancing user experience in mobile networks. TPTO achieves a training time 2.5 faster than MRLCO. The Transformer architecture of TPTO is mainly responsible for this training time difference. Transformers are known for their parallel execution and efficient utilization of self-attention mechanisms, which can exploit the parallel processing capabilities of modern hardware architectures, resulting in a faster training process. These results underscore the potential benefits of employing Transformer-based models for optimizing offloading decisions in the edge computing environment.

TABLE 3. COMPARATIVE ANALYSIS OF TPTO AND BASELINE METHODS: AVERAGE LATENCY (MS) ACROSS DIVERSE TEST DATASETS.

Testing Topology Sets	Algorithm	Wireless Transmission Rate of r_{up} and r_{do}	
		8.5Mbps	11.5Mbps
fat = 0.8 density = 0.6 ccr = 0.5	HEFT	1033	835
	Greedy	1064	837
	MRLCO	846	760
	TPTO	741	581
fat = 0.5 density = 0.7 ccr = 0.3	HEFT	1157	849
	Greedy	1462	952
	MRLCO	989	869
	TPTO	1022	811
fat = 0.6 density = 0.8 ccr = 0.4	HEFT	1521	943
	Greedy	1009	822
	MRLCO	894	810
	TPTO	900	719

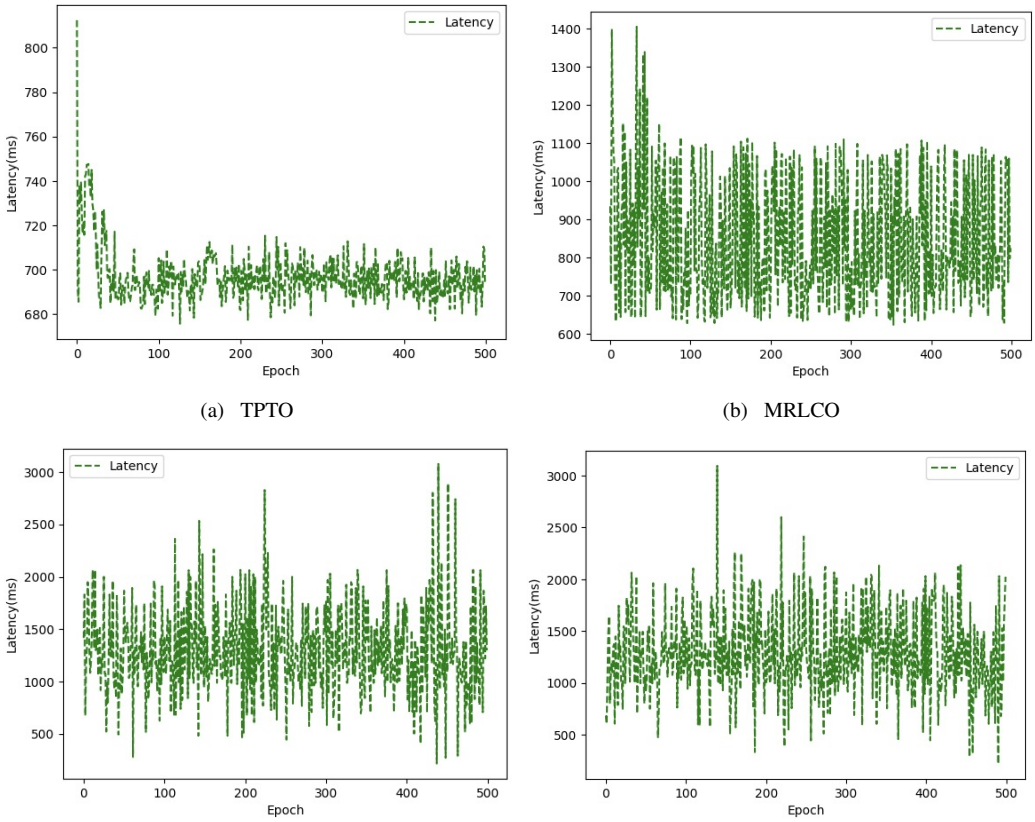
5. Related Work

Approaches for computation offloading to edge computing are a very active topic. Proposed techniques fall mainly into machine-learning and optimization-based methods.

Machine-Learning Offloading Approaches: Qu *et al.* present a framework for IoT devices to offload computing tasks to edge servers [23]. They use deep meta-reinforcement learning to minimize energy consumption, task computation, and transmission delays by dividing applications into sequential workflows. The proposed framework, Deep Meta Reinforcement learning based Offloading (DMRO), includes an inner and outer loop. The former relies on Q-learning, whereas the latter employs a meta-algorithm to learn the initial parameters and adapt to changing environments, quickly converging to optimal offloading solutions.

Huang *et al.* [24] propose MELO, a Meta-Learning-based computation Offloading algorithm for independent tasks in edge computing, which consists of one edge server and N wireless devices, each with a prioritized task to execute. They apply binary offloading, where tasks run locally on a device or the edge server. The approach focuses on minimizing latency, communication, and computation delay.

Yang *et al.* [28] tackle joint offloading optimization and bandwidth allocation, modeled as a mixed-integer programming (MIP) problem for independent tasks. They propose the Deep Supervised Learning-based computational Offloading



(a) TPTO (b) MRLCO

(c) HEFT (d) Greedy

Figure 4. Influence of wireless transmission rate and network topology on latency.

TABLE 4. SUMMARY OF RELATED WORK AND THEIR COMPARISON WITH TPTO.

Techniques	Category	Task Dependency	Task Offloading Engine		
			Main Approach	Task Priority	Network Architecture
Nguyen *et al.* [25]	Optimization	✓	Discrete whale optimization	✗	✗
Abbas *et al.* [26]		✗	Ant colony, whale, grey wolf	✗	✗
Xu *et al.* [27]		✗	Order-preserving policy, bisection search	✗	✗
Qu *et al.* [23]	Machine learning	✓	Deep Meta Learning, Q-learning	✗	DNN
Huang *et al.* [24]		✗	Meta-Learning	✓	DNN
Yang *et al.* [28]		✗	Deep Supervised Learning	✓	CNN, DNN
TPTO		✓	PPO, Actor-Critic	✓	Transformer NN

(DSLO) algorithm that considers task delay and energy consumption. Furthermore, incorporating batch normalization into two classical neural network architectures, CNN and DNN, enhances the convergence speed of DSLO.

Optimization-based Offloading Techniques: Nguyen *et al.* [25] introduce a collaborative scheme for Unmanned Aerial Vehicless (UAVs) to share workloads. They consider the task topology, which involves splitting a task into sub-tasks with dependencies and the power consumption constraints of the UAVs in edge computing. They use the discrete whale optimization algorithm and CVXPY's SCS solver to solve the optimization problem, modeled as a mixed-integer, non-linear, and non-convex problem. Abbas *et al.* [26] present classical approaches for optimal independent task offloading in edge computing environments. They use well-known meta-heuristics such as the ant colony optimization algorithm, whale optimization algorithm, and Grey wolf optimization algorithm, adapting these algorithms to their problem. The goal is to minimize the energy consumption of user devices and IoT and minimize response time for task computation in edge computing. A search-based meta-heuristic model, introduced by Xu *et al.* [27], also focused on task offloading and time allocation in edge computing for independent tasks. Considering computation rate and task execution latency, they formulated

the problem as a Mixed Integer Programming (MIP) and divided it into sub-problems: offloading decision and resource allocation. They proposed an "order-preserving policy generation method", which works well in large networks.

Machine learning approaches often demonstrate superior performance than traditional optimization methods. Still, existing work generally employs conventional DRL with sequential neural networks, resulting in less computation efficiency and extended training time. Moreover, most real-world applications comprise dependent tasks, which prior work generally ignores [27], [26], [28], [24]. TPTO tackles these limitations, exhibiting fast adaptability to new tasks by applying Transformers and effectively minimizing latency – an essential concern for delay-sensitive applications – by strategically considering task dependencies. Table 4 summarizes and compares related works with TPTO.

6. Conclusions and Future Work

This work introduced TPTO, a distributed DRL method for task offloading optimization in edge computing using Transformers to reduce latency in DAG-structured applications. We first introduced a latency model that optimizes the task execution time, communication, and offloading in an edge computing environment. This model serves as the basis for the decision-making process in TPTO. Then, experimental results demonstrated TPTO's effectiveness under various network conditions and topologies. TPTO presents superior performance compared to three baseline algorithms: MRLCO, HEFT, and Greedy. In addition, TPTO consistently achieved the lowest latency, showcasing its ability to make efficient offloading decisions. Future research will focus on TPTO's scalability in larger edge computing setups and explore multi-criteria optimization, including energy consumption and execution cost.

References

[1] M. Chen, T. Wang, S. Zhang, and A. Liu, "Deep reinforcement learning for computation offloading in mobile edge computing environment," *Computer Communications*, vol. 175, pp. 1–12, 2021.

[2] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[3] K. Kirkpatrick, "Software-defined networking," *CACM*, vol. 56, no. 9, pp. 16–19, 2013.

[4] Z. Tong, X. Deng, F. Ye, S. Basodi, X. Xiao, and Y. Pan, "Adaptive computation offloading and resource allocation strategy in a mobile edge computing environment," *Information Sciences*, vol. 537, pp. 116–131, 2020.

[5] M. Goudarzi, H. Wu, M. 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, 2020.

[6] F. R. d. Souza, A. D. Silva Veith, M. Dias de Assunção, and E. Caron, "Scalable joint optimization of placement and parallelism of data stream processing applications on cloud-edge infrastructure," in *ICSOC*, 2020, pp. 149–164.

[7] 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.

[8] 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.

[9] 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 Comm. Magazine*, vol. 57, no. 3, pp. 56–62, 2019.

[10] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[11] M. Faraji-Mehmandar, S. Jabbehdari, and H. H. S. Javadi, "A self-learning approach for proactive resource and service provisioning in fog environment," *The Journal of Supercomputing*, pp. 1–30, 2022.

[12] W. Hashem, R. Attia, H. Nashaat, and R. Rizk, "Advanced deep reinforcement learning protocol to improve task offloading for edge and cloud computing," in *Int. Conf. on Advanced Machine Learning Technologies and Applications*, 2022, pp. 615–628.

[13] T. Zheng, J. Wan, J. Zhang, and C. Jiang, "Deep reinforcement learning-based workload scheduling for edge computing," *Journal of Cloud Computing*, vol. 11, no. 1, pp. 1–13, 2022.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[15] R. Mahmud and A. N. Toosi, "Con-pi: A distributed container-based edge and fog computing framework," *IEEE Internet of Things Journal*, vol. 9, no. 6, pp. 4125–4138, 2022.

[16] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Intr.* MIT press, 2018.

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

[18] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[20] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[21] 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.

[22] X. Lin, Y. Wang, Q. Xie, and M. Pedram, "Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 175–186, 2014.

[23] G. Qu, H. Wu, R. Li, and P. Jiao, "Dmro: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3448–3459, 2021.

[24] 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 Communications Letters*, vol. 25, no. 5, pp. 1568–1572, 2020.

[25] L. X. Nguyen, Y. K. Tun, T. N. Dang, Y. M. Park, Z. Han, and C. S. Hong, "Dependency tasks offloading and communication resource allocation in collaborative uavs networks: A meta-heuristic approach," *IEEE Internet of Things Journal*, 2023.

[26] A. Abbas, A. Raza, F. Aadil, and M. Maqsood, "Meta-heuristic-based offloading task optimization in mobile edge computing," *Int. Journal of Distributed Sensor Networks*, vol. 17, no. 6, 2021.

[27] Y. Xu, Y. Wang, and J. Yang, "Meta-heuristic search based model for task offloading and time allocation in mobile edge computing," in *Proc. the 6th International Conference on Computing and Artificial Intelligence*, 2020, pp. 117–121.

[28] S. Yang, G. Lee, and L. Huang, "Deep learning-based dynamic computation task offloading for mobile edge computing networks," *Sensors*, vol. 22, no. 11, p. 4088, 2022.