Graphical Abstract

NOTE and T-NOTE an efficient transformer encoder for IoT task offloading in edge-fog-cloud computing environment

Highlights

NOTE and T-NOTE an efficient transformer encoder for IoT task offloading in edge-fog-cloud computing environment

- Research highlight 1
- Research highlight 2

NOTE and T-NOTE an efficient transformer encoder for IoT task offloading in edge-fog-cloud computing environment

, , , , ,

Abstract

The Internet of Things (IoT) is a rapidly expanding field, with increasing applications in various sectors, including healthcare, industry, and the connected home. This necessitates the offloading of processes to facilitate the orchestration and computation of tasks generated by these devices. In light of the challenges associated with cloud offloading, including delays, costs and energy consumption, the fog alternative has emerged as a novel paradigm for offloading, addressing these issues. This expansion, coupled with the advent of new environments, has given rise to numerous offloading strategies, encompassing a range of approaches from heuristics to deep learning and metaheuristics. The necessity to evaluate these algorithms is the catalyst for the exploration of a realistic hybrid edge, fog and cloud environment scenario in this paper, which is based on publicly available data. This facilitates a comparison between multi-objective genetic-algorithm (GA) and QRL-based optimization techniques for a multilayer perceptron (MLP). The following paper puts forward a proposal for two architectures, the basis of which is the most recent developments in a range of fields, including NLP, vision, and so forth. These are known as Transformers, and they are adapted to the cloud, fog, edge, and MEC using Deep Q Renforcement learnig (DQRL) algorithms. The enhancement of the...

TaskOflloding fomer TOFormer NodeOffloadingFormer NOFormer

Node-based Offloading Tranformer Encoder (NOTE)

T

Keywords:

1. Introduction

1.1. Background and motivation

The Internet of Things (IoT) has become a cornerstone of modern technology, enabling innovative applications in smart cities, healthcare, autonomous systems, and more. The proliferation of IoT devices has been rapid and significant, with Cisco reporting a global total exceeding 30 billion [**benaboura_comprehensive_noda**
These devices generate a substantial volume of data, approximately 2 exabytes on a daily basis. Efficient processing and analysis are required to fully realise their potential. However, the proliferation of IoT has given rise to considerable challenges in the domains of data management, resource allocation, and system scalability.

The inherent limitations of IoT devices, such as their small batteries, limited processing power, and minimal storage capacity, render them ill-suited to manage the substantial volumes of data they generate. It is evident that tasks such as real-time processing of sensor data or computation-intensive applications frequently exceed the capabilities of local devices. Conventional approaches entail the delegation of these tasks to centralized cloud servers. However, network limitations and latency sensitivity frequently render this approach inefficient, particularly for real-time applications [**benaboura_comprehensive_nodat**

In order to address these challenges, fog computing has emerged as a distributed computing paradigm that extends the capabilities of cloud computing to the edge of the network. By facilitating the execution of storage, computation, and data management operations in close proximity to the data source, fog computing contributes to the reduction of latency, power consumption, and network traffic. It is evident that contemporary applications, including but not limited to smart homes, autonomous vehicles, smart agriculture, and healthcare, are contingent on this paradigm in order to satisfy their real-time and location-aware processing requirements [**das_review_2023**]. Fog computing, first introduced in 2012, provides a hierarchical architecture that serves to bridge the gap between cloud servers and IoT devices, thereby allowing for seamless data flow and operational efficiency [**fahimullah_review_2022**].

Notwithstanding the advantages inherent in task offloading in fog computing, this process is encumbered by numerous challenges. Optimizing resource utilization, reducing energy consumption, and maintaining Quality of Service (QoS) require robust strategies due to the heterogeneous and dynamic nature of fog networks. Factors such as fluctuating workloads, mo-

bility, and task diversity have been identified as contributing to an increase in the complexity of the situation. In order to address these challenges, innovative resource allocation techniques are required, including machine learning (ML)-based methods, auction models, and heuristic optimization [**fahimullah_review_2022**].

Conventional resource management techniques frequently employ static heuristic approaches, which prove ineffective when confronted with the diverse and dynamic workloads that are characteristic of fog environments. These methods, when configured offline for specific scenarios, lack the scalability and flexibility required for real-time task offloading and resource optimization. Consequently, a substantial decline in performance is experienced as system demands increase [**iftikhar_ai-based_2023**].

To address this problem, metaheuristics such as GA have emerged as powerful optimization techniques capable of handling the multi-objective nature of fog computing task offloading. GAs excel in exploring complex solution spaces and finding near-optimal trade-offs between conflicting objectives such as latency minimization, energy efficiency, and resource utilization. Unlike traditional optimization methods that often focus on single objectives, multi-objective genetic algorithms can simultaneously optimize multiple performance criteria, making them particularly suitable for fog computing environments where various QoS parameters must be balanced. The evolutionary nature of GAs allows them to adapt to dynamic network conditions and heterogeneous resource availability, providing robust solutions for real-time task offloading decisions.

Recent advancements in the field of Artificial Intelligence (AI), particularly deep reinforcement learning (DRL), have demonstrated considerable potential in addressing the intricacies of task offloading in fog environments. Research has demonstrated the efficacy of AI-driven approaches in reducing latency, energy consumption, and operational costs. As [**fahimullah_review_2022**] demonstrates in their review, techniques such as centralized dueling deep Q-networks (DDQNs), decentralized learning models, and multi-agent reinforcement learning have been employed to optimize offloading policies and resource allocation. Furthermore, hybrid strategies that integrate artificial intelligence with conventional methods have demonstrated efficacy in heterogeneous fog environments [**mishra_collaborative_2023**].

In order to maintain currency with the latest advancements in this domain, the present study investigates innovative methodologies that integrate the strengths of evolutionary computation and deep reinforcement learning

for the purpose of optimizing IoT task offloading. The integration of these approaches addresses the limitations of individual techniques while leveraging their complementary advantages.

Metric	Description
C: Cost	Refers to the expenses incurred in task offloading, including computation, storage, and data transfer costs.
L: Latency	The time delay between the initiation of a request and the reception of the response. Crucial for real-time applications.
E: Energy	The total energy consumed during task offloading, including device and network-level energy usage.
ET: Execution Time	The total time taken to execute a task from start to finish, including computation and communication time.
D: Delay	The time difference between task submission and the start of its processing. Includes network and processing delays.
M: Makespan	The total time required to complete all tasks in a batch or workflow. Indicates overall system efficiency.
QWT: Queue Waiting Time	The duration a task spends waiting in the queue before being processed. Impacts response time and throughput.
TTR: Task Throw Rate	The rate at which tasks are successfully processed and completed in the system, indicating system throughput.
DLV: Dead Line Violation	The percentage or rate of tasks that succeed to complete within their specified deadlines, indicating system reliability and quality of service.

Table 1: Explanation of QoS Metrics in Edge-Fog-Cloud Computing

1.2. Contributions

To address these gaps, this work makes the following key contributions:

- The proposal of a two-transformers-based architecture, utilizing DQRL for the purpose of task offloading within a cloud-fog-edge environment, is hereby presented. The selection of the node (edge, fog or cloud) to execute each incoming task is determined by these models, representing n different ctions. The NOTE system is oriented towards node-level features, such as CPU, buffer, and bandwidth. In contrast, the T-NOTE system incorporates additional task attributes, including size, deadline, and CPU-cycle requirements. This capability facilitates a more precise depiction of the interplay between node resources and task demands within a fog environment.

- A transformation of the conventional static genome to a dynamic one is achieved by adapting the Niched Pareto Genetic Algorithm (NPGA) and the Non-dominated Sorting Genetic Algorithm II (NSGA-II), two multi-objective genetic algorithms (GAs), to a MLP as the genome. Subsequently, a comparative analysis was conducted between the GAs and a Deep Q-Learning (DQRL) approach. The implementation of these algorithms in the training of a MLP constitutes a pivotal element of the study. The GA approach is employed in a dynamic setting, thereby enabling the MLP to be optimized for multi-objective offloading in real time.

- This work also puts forth a model for offloading in hybrid environments that considers a substantial number of parameters for the QoS. To

further expand upon the existing body of knowledge, an adaptation of a scenario with a real dataset was implemented within the framework of RayClousSim, which is available at https://github.com/tutur90/Task-Offloading-Fog.

1.3. Paper organization

The reminder of this paper is structured as follows: Section **??** reviews related work;.....

2. Related Work

Task offloading in Fog/Cloud environments is a relatively recent research area compared to well-studied domains like image classification or time series forecasting. However, the decision-making process for task offloading is inherently complex due to its combinatorial nature.

2.1. Deterministic Approaches

Deterministic algorithms provide a direct method for solving the task offloading problem. For instance, the optimal task assignment algorithm proposed by Yan *et al.* [**yan_optimal_2020**] employs a three-step approach: (i) assuming the offloading decisions are pre-determined, (ii) deriving closed-form expressions for optimal offloading, and (iii) implementing bisection search and a one-climb policy. This approach effectively reduces energy consumption and execution time for IoT tasks in Mobile Edge Computing (MEC) systems.

Nevertheless, the task offloading problem is considered NP-hard, as demonstrated in [**guo_algorithmics_2024**, **jin_task_2024**, **sarkar_deep_2022**]. Consequently, deterministic methods face scalability limitations, prompting the exploration of heuristic, metaheuristic, and AI-based approaches.

2.2. Heuristic and Metaheuristic Methods

Heuristic methods are well-suited for scaling in complex edge/cloud environments, as they avoid the exponential computational growth of deterministic algorithms [**zhang_survey_2024**]. For example, the Deadline and Priority-aware Task Offloading (DPTO) algorithm [**adhikari_dpto_2020**] schedules tasks by prioritizing delays and task priorities, selecting optimal devices to minimize overall offloading time.

Metaheuristic algorithms, known for their adaptability, have also shown strong performance. Bernard *et al.* [**bernard_d-npga_2024**] introduced the Drafting Niched Pareto Genetic Algorithm (D-NPGA), which optimizes task offloading decisions and improves makespan and cost efficiency for IoT tasks in fog/cloud systems.

More sophisticated methods use multiple approaches. For example, Energy-Efficient and Deadline-Aware Task Scheduling in Fog Computing (ETFC) [**pakmehr_etfc_2024**] employs a Support Vector Machine (SVM) to predict traffic on fog nodes and classify them as low- or high-traffic. Then, they use reinforcement learning (RL) on the low-traffic group and a non-dominated sorting genetic algorithm III (NSGA-III) on the high-traffic group to make the offloading decision. This allows both algorithms to perform better on adequate tasks.

Metaheuristic approaches have gained attention for their ability to address the NP-hard nature of the task offloading problem with flexibility and adaptability. A recent and comprehensive survey by Rahmani *et al.* [**rahmani_optimizing_2025**] reviewed a wide range of metaheuristic algorithms—including Genetic Algorithms (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), and Grey Wolf Optimizer (GWO)—applied to task offloading in IoT environments. The review identified key strengths of these methods in balancing latency, energy consumption, and cost efficiency across heterogeneous fog-edge-cloud infrastructures. Their taxonomy also distinguishes between evolutionary and swarm-based algorithms, showing that hybrid models are increasingly used to adapt to dynamic network conditions. Despite their strengths, the authors noted limitations in scalability and real-time adaptability—areas where AI and deep learning techniques, such as DRL or transformers, may provide an edge.

2.3. AI Approaches

Machine learning (ML) has recently demonstrated its efficacy in task offloading. For instance, decision tree classifiers [**suryadevara_energy_2021**] determine whether tasks should be offloaded to the fog or cloud, resulting in reduced latency and energy consumption. Logistic regression models [**bukhari_intelligent_2022**] have also been applied, estimating the probability of successful offloading by leveraging maximum likelihood estimation.

Deep learning (DL), a subset of ML, deserves special attention due to its notable advances across various fields.

Basic deep neural networks (DNNs) have been applied to IoT offloading tasks, such as in [**sarkar_deep_2022**], where parallel DNNs optimize cost, energy consumption, and latency. However, deep reinforcement learning (DRL) is often preferred for its decision-making capabilities in dynamic environments.

For example, Jiang *et al.* [**jiang_reinforcement_2021**] utilized Deep Q-Learning (DQN) to identify optimal offloading policies and resource allocation strategies for user equipment in fog/cloud systems. Their approach combines a dueling deep Q-network for model pre-processing and a distributed deep Q-network for efficient task allocation.

Moreover, multi-agent DRL methods [**ren_deep_2021**] have been employed to offload IoT tasks. Each IoT device trains its DRL model to select fog access points, followed by a greedy algorithm to determine cloud offloading. This approach demonstrates competitive performance in energy efficiency compared to exhaustive search and genetic algorithms.

Long Short-Term Memory (LSTM) networks, a type of recurrent neural network, are particularly suitable for the temporal dimensions of task offloading. Tu *et al.* [**tu_task_2022**] combined LSTM with DQN to predict task dynamics in real-time, leveraging observed edge network conditions and server load. This hybrid model significantly improved latency, offloading cost, and task throughput.

Transformers, known for their revolutionary impact on natural language processing (NLP), have also been adapted for task offloading. Gholipour *et al.* [**gholipour_tpto_2023**] proposed TPTO, a transformer-based framework with Proximal Policy Optimization (PPO). Their model encodes task dependencies using a transformer encoder and employs an actor-critic framework trained with PPO to generate probability distributions for offloading actions, achieving state-of-the-art results in edge computing environments.

Algorithms are referenced in the Table **??** with the method, the dataset type, the QOS evaluated, and the job type.

While many existing offloading solutions rely on synthetic datasets and focus on Multi-access or Vehicular Edge Computing (MEC/VEC) scenarios [**fahimullah_review_2022**, **tu_task_2022**, **gholipour_tpto_2023**], research specifically targeting fog computing remains limited. In particular, models such as Deep Neural Networks (DNNs) [**sarkar_deep_2022**] and Deep Q-Learning (DQL) [**jiang_reinforcement_2021**] have rarely been evaluated in realistic fog environments using real-world data.

Genetic Algorithms (GAs) are often static in nature, such as in [**bernard_d-npga_2024**,

pakmehr_etfc_2024], meaning that they cannot be employed in real-time applications and can only converge through iterations on the same simulation environment where tasks and their order remain static. This characteristic makes them particularly complex to deploy in real-world applications where dynamic adaptation is required.

Transformer-based deep learning techniques remain poorly explored in this domain, despite being state-of-the-art in many fields. Some exploration efforts show promise; however, existing approaches often implement basic action spaces, such as TPTO [**gholipour_tpto_2023**], which is designed with only two actions: offloading the task to the cloud or processing it on the edge.

Paper	Method	Algorithm	Dataset	QoS	Type of Job	Environment	Tool
[**yan_optimal_2020**]	Deterministic	Optimal Task Assignment	Real	E, ET	IoT tasks	MEC	NA
[**adhikari_dpto_2020**]	Heuristic	DPTO	Synthetic	QWT, D	IoT tasks	Cloud & Fog	NA
[**bernard_d-npga_2024**]	Metaheuristic	D-NPGA	Synthetic	C, M	IoT tasks	Cloud & Fog	Python
This work	Metaheuristic	NPGA/NSGA-II+MLP	Real	C, L, TTR	IoT tasks	Cloud & Fog	Python, PyTorch
[**pakmehr_etfc_2024**]	AI, Metaheuristics	ETFC	C, E, L, DLV	Synthetic	IoT tasks	Fog	NA
[**bukhari_intelligent_2022**]	AI	Logistic Regression (LR)	Real	C, E, L	IoT tasks	Cloud & Fog	Python, Matlab
[**suryadevara_energy_2021**]	AI	Decision Tree (DT)	Synthetic	E, L	IoT tasks	Cloud & Fog	iFogSim
[**sarkar_deep_2022**]	AI	Deep Neural Network (DNN)	Synthetic	C, E, L	Mobile	Cloud & Fog	iFogSim
[**jiang_reinforcement_2021**]	AI	Deep Q-Network (DQN)	Synthetic	E, L	Mobile	Cloud & Fog	Python, Adam optimizer
[**ren_deep_2021**]	AI	Multi-agent DRL	Synthetic	E	IoT tasks	Cloud & Fog	NA
[**tu_task_2022**]	AI	DRL+LSTM	Real	C, L, TTR	Mobile	MEC	NA
[**gholipour_tpto_2023**]	AI	Transformers PPO	Synthetic	L	IoT tasks	Edge	Python, TensorFlow
This work	AI	DQRL+MLP	Real	C, L, TTR	IoT tasks	Cloud & Fog	Python, PyTorch
This work	AI	Transformers	Real	C, L, TTR	IoT tasks	Cloud & Fog	Python, PyTorch

Table 2: Summary of Task Offloading Methods in Fog and Edge Computing

3. Problem Modeling

3.1. Scenario Modeling

This scenario is modeled according to foundational concepts presented in [**aazam_cloud_2022**, **bukhari_intelligent_2022**, **jazayeri_autonomous_2021**], following a three-tier offloading approach that involves *edge*, *fog*, and *cloud* nodes. In Figure **??**, a high-level cloud-enabled architecture is shown, where a **Global Gateway (GG)** collects tasks from various IoT devices before determining whether to process them locally or offload them to fog or cloud resources.

A real-world dataset of IoT-generated tasks, is employed to study and evaluate these offloading decisions. The tasks in this dataset vary in size, deadline constraints, and computational complexity, thereby reflecting the heterogeneous nature of practical IoT workloads.

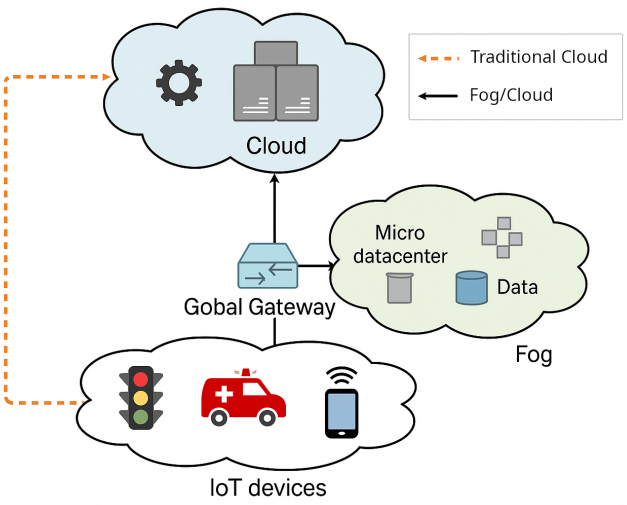

Figure 1: System Architecture

3.1.1. Architecture

In Figure **??**, a schematic overview of the **proposed three-tier architecture** is illustrated. The system is composed of IoT devices (sensors, mobile nodes, and other edge devices) that generate tasks. These tasks subsequently reach the Global Gateway, which processes them locally if resources are available, or decides to offload them to the fog or the cloud based on resource availability and energy considerations.

Global Gateway. The GG layer is located at the edge and aggregates tasks from local IoT devices. In addition to routing capabilities, the GG has limited computational power, enabling it to handle *smaller tasks* locally and thus reduce network traffic. This approach is particularly advantageous for time-sensitive tasks that can be processed quickly on-site.

Upon receiving a task, several factors are evaluated:

- Node state: The current load on the GG, along with the availability of fog and cloud nodes.
- Network conditions: Bandwidth, potential congestion, and latency to fog or cloud nodes.
- Energy usage: A predictive assessment of energy costs if the task is executed locally versus offloaded.

- Task requirements: Size, computational complexity, and deadline or QoS constraints.

A subsequent decision is made to determine the optimal execution strategy for the task. The system evaluates three primary options: processing the task locally if it is computationally feasible and resource-efficient, offloading the task to a nearby Fog Node to leverage edge computing capabilities, or offloading the task directly to a Cloud Node for intensive computational requirements that exceed local and fog node capacities.

Fog Nodes. Fog nodes provide moderate computational resources that exceed those of edge devices but remain below the capacity of cloud data centers. These intermediate computing nodes offer lower latency than distant cloud nodes due to their closer physical proximity to the network edge. Fog nodes serve as an essential intermediate tier for tasks that cannot be processed locally at the Gateway (GG) yet do not require the extensive resources of the cloud infrastructure. This positioning makes them ideal for applications requiring balanced performance between computational capability and response time.

Cloud Nodes. Cloud nodes represent large-scale data centers with substantial computational and storage capabilities. Although they typically have higher baseline latencies because of greater network distances, they are well suited for computationally intensive applications. Cloud nodes excel in handling large tasks that demand significant CPU and memory resources, making them optimal for complex analytical workloads. They are particularly effective for batch processing scenarios with relaxed real-time constraints, where throughput is prioritized over immediate response. Furthermore, cloud data centers can accommodate high concurrency situations through dynamic resource scaling, allowing them to adapt to varying computational demands efficiently. concurrency situations, as cloud data centers can scale resources dynamically.

CPU capacity: Cloud servers can support a high volume of CPU cycles, facilitating efficient offloading for tasks with elevated computational demands.

3.2. QoS Modeling

Task offloading decisions are evaluated within the RayCloudSim framework [**zhang2022osttd**], a comprehensive simulation platform for modeling and assessing cloud-edge-IoT computing environments based on [**WiesnerThamsen_LEAF_202**

3.2.1. Task Throw Rate

In distributed environments, task execution failures can occur due to issues such as *network disconnections*, *node isolation*, or *buffer overflows*.

The *task throw rate* τ is defined as:

$$\tau = \frac{\text{Number of failed tasks}}{\text{Total tasks generated}}. \tag{1}$$

A lower throw rate is indicative of a more robust and efficient offloading strategy.

3.2.2. Latency

The latency metric for task offloading is exclusively defined for tasks that achieve successful offloading, as the underlying assumption requires that tasks both arrive at the destination and undergo computational processing. For tasks meeting this criterion, the total latency encompasses three distinct components and is formally expressed as:

$$L_{\text{total}} = L_{\text{transmission}} + L_{\text{processing}} + L_{\text{queuing}} \tag{2}$$

Conversely, for tasks that fail to achieve successful offloading, whether due to transmission failures, processing errors, or other system constraints, the total latency is assigned a null value:

$$L_{\text{total}} = 0 \tag{3}$$

Transmission Delay. Transmission delay is composed of transfer delay and propagation delay:

$$L_{\text{transmission}} = L_{\text{transfer}} + L_{\text{propagation}}. \tag{4}$$

1. Transfer Delay: The duration required to send all bits of a packet onto the transmission medium:

$$L_{\text{transfer}} = \frac{S}{B}, \tag{5}$$

   where S is the data size (in bits) and B is the allocated bandwidth for the task (in bits per second).

2. Propagation Delay: The time taken for a signal to traverse the medium:

$$L_{\text{propagation}} = \frac{d}{v}, \tag{6}$$

   where d is the total transmission distance (in meters) and $v \approx 2 \times 10^8$ m/s in optical fiber.

Processing Delay. Processing delay is the time spent by a computing node on executing a task:

$$L_{\text{processing}} = \frac{C \cdot S}{f}, \tag{7}$$

where C is the number of CPU cycles required to process the task, and f is the CPU frequency (in cycles per second).

Queuing Delay. Queuing delay captures the waiting time a task experiences when the node is busy executing other tasks:

$$L_{\text{queuing}} = \sum_{i=1}^{N} L_{\text{processing},i}, \tag{8}$$

where N is the number of tasks that arrived before the current task, and $L_{\text{processing},i}$ is the processing time of each task in the queue.

3.2.3. Energy Consumption Model

Similar to the latency metric, energy consumption is exclusively defined for tasks that achieve successful offloading. The simulation framework categorizes energy consumption into three distinct components:

- **Idle Energy** (E^i_{idle}): Energy drawn by node i when it remains idle, estimated through an idle power coefficient P_{idle}.
- **Execution Energy** ($E^{i,k}_{\text{exe}}$): Energy consumed by node i during the execution of task k, based on execution power P_{exe}.
- **Transmission Energy** ($E^{i,k}_{\text{trans}}$): Energy utilized to transmit task k from source node j to destination node i, determined by a per-bit cost $C^{m,n}$ on the communication link.

All energy terms are expressed in Joules (J), while power (P) is measured in Watts (W). Let N denote the number of nodes and T represent the total number of tasks successfully offloaded. The overall energy consumption in the system is calculated as:

$$E_{\text{total}} = \sum_{i=1}^{N} \Big(E^i_{\text{idle}} + \sum_{k=1}^{T} \big(E^{i,k}_{\text{exe}} + E^{i,k}_{\text{trans}} \big) \Big) \tag{9}$$

This formulation ensures that energy consumption measurements reflect only the computational and communication activities associated with successful task offloading, maintaining consistency with the latency metric definition and providing a meaningful performance evaluation.

Node Power Consumption Model. Based on **[ismail_computing_2021]**, the power consumption of a computing node can be approximated by:

$$P(u_{\text{cpu}}) = \alpha + \beta \cdot u_{\text{cpu}}, \tag{10}$$

where:

- α represents the idle power (P_{idle}).
- β is the incremental power coefficient for executing tasks, defined by $\beta = (P_{\text{exe}} - P_{\text{idle}})$.
- $u_{\text{cpu}} \in [0, 1]$ is the CPU utilization ratio.

This model has a Standard Error of Estimation (SEE) of 12.9%, indicating a reasonably accurate fit to empirical data.

Idle Energy. The idle energy for node i over the entire simulation duration T can be calculated as:

$$E^i_{\text{idle}} = \int_0^T P^i_{\text{idle}}\, dt = P^i_{\text{idle}} \cdot T. \tag{11}$$

Execution Energy. Execution energy can be expressed by integrating the CPU utilization over time:

$$E^i_{\text{exe}} = \int_0^T \left(P^i_{\text{exe}} - P^i_{\text{idle}}\right) \cdot u_{\text{cpu}}(t)\, dt. \tag{12}$$

In practical simulations, full CPU utilization ($u_{\text{cpu}} = 1$) is often assumed while tasks are running, yielding:

$$E^{i,k}_{\text{exe}} = T^{i,k}_{\text{exe}} \cdot \left(P^i_{\text{exe}} - P^i_{\text{idle}}\right), \tag{13}$$

where $T^{i,k}_{\text{exe}}$ represents the execution time of task k on node i.

Transmission Energy. The transmission energy required to move task k from node j (source) to node i (destination) depends on data size and per-bit link costs:

$$E_{\text{trans}}^{i,k} = s^k \sum_{(m,n)\in I^{j,i}} C^{m,n}, \tag{14}$$

where:

- s^k is the size of task k in bits.
- $C^{m,n}$ is the energy cost per bit (J/bit) on the link from node m to node n.
- $I^{j,i}$ is the set of links used along the path from node j to node i.

This model offers a structured, interpretable framework for quantifying energy consumption in offloading scenarios, thereby facilitating meaningful comparisons of energy efficiency under different scheduling and resource-allocation strategies.

4. Proposed Offloading Strategies

A range of offloading strategies was proposed to explore different decision-making paradigms, including NPGA, NSGA , and (DQRL). Each approach balances complexity and global optimality to varying degrees.

4.1. Metaheuristics

Metaheuristic algorithms, such as evolutionary methods, excel at exploring large, dynamic search spaces in IoT offloading. Although they have been applied in more static contexts [**bernard_d-npga_2024**], this work extends them to a *dynamic* genome representation namely, an MLP mapping real-time states to offloading actions, following ideas from [**such2018deepneuroevolutiongenetic**

4.1.1. GA

Genome Representation (MLP Encoding). A candidate solution (or individual) is encoded as the set of weight matrices in a multi-layer perceptron (MLP). Each layer's trainable parameters form one matrix, and the full network $\{\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_L\}$ defines how input features (e.g., CPU, buffer, bandwidth) are transformed into node-specific scores for offloading. Evolutionary operators act directly on these matrices over multiple generations.

!!RAJOUTER WL

Crossover and Mutation. **Crossover:** Two parent genomes $\{\mathbf{W}_1^{(p1)}, \dots\}$ and $\{\mathbf{W}_1^{(p2)}, \dots\}$ are combined by arithmetic crossover:

$$\mathbf{W}_\ell^{(\text{child})} = \alpha\, \mathbf{W}_\ell^{(p1)} + (1-\alpha)\, \mathbf{W}_\ell^{(p2)},$$

where $\alpha \in [0, 1]$ is chosen at random, ensuring offspring inherit traits from both parents.

Mutation: With probability p_{mutation}, each element of a weight matrix is perturbed by Gaussian noise, $\mathbf{W}[i, j] \leftarrow \mathbf{W}[i, j] + \epsilon_{ij}$, where $\epsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$. Clipping may be applied to maintain valid parameter ranges.

Overall Workflow of Genetic Algorithms. The genetic algorithm process begins with population initialization, where an initial population of candidate solutions (chromosomes) is randomly generated using an appropriate representation format. Each individual in the population is then evaluated using multiple objective functions to determine fitness values across all optimization criteria. The selection phase follows, where parent individuals are chosen for reproduction based on their performance scores. The core evolutionary operations of crossover and mutation are then applied to generate offspring by recombining selected parents through crossover operations and introducing random variations through mutation. Subsequently, the replacement phase evaluates new offspring and forms the next generation by combining or replacing individuals from the parent and offspring populations using multi-objective replacement strategies. The algorithm terminates after reaching a maximum number of generations, achieving convergence criteria, or meeting other predefined stopping conditions. The final Pareto front represents the set of non-dominated solutions, effectively demonstrating the trade-offs among competing objectives. The pseudo code of the GA is referenced in Algorithm **??**.

Multi-objective GA. As the GA algorithm is based on multiple individuals, it can achieve multiple objectives. During the selection process, the selected individuals were able to select multiple scores, rather than a single score, thereby creating the Pareto front. The present study focuses on two scalable and robust approaches.

NPGA [**horn1994npga**] employs a combination of tournament selection and a niching mechanism to maintain population diversity and prevent premature convergence to a single region of the Pareto front. The selection

process works by conducting tournaments between randomly selected candidate solutions, where the winner is determined based on Pareto dominance relationships within a randomly chosen comparison set. Specifically, two candidates compete by counting how many individuals they dominate from a subset of the population, and the candidate with higher dominance count is selected. This approach naturally maintains diversity by distributing selection pressure across different regions of the Pareto front without requiring explicit front classification.

In contrast, *NSGA-II* [**Deb2002AFA**] uses non-dominated sorting to classify solutions into hierarchical fronts and employs crowding distance calculations to preserve solution diversity within each front and guide selection for the next generation. The selection mechanism first performs non-dominated sorting to organize the population into ranked fronts ($\mathcal{F}_1, \mathcal{F}_2, \ldots$), where $\mathcal{F}_1$ contains the best non-dominated solutions. Within each front, crowding distance is calculated to measure the density of solutions in the objective space, favoring individuals in less crowded regions. Selection prioritizes individuals from better fronts first, and within the same front, those with larger crowding distances are preferred to maintain diversity along the Pareto front.

Algorithm 1 Multi-Objective Genetic Algorithm for IoT Task Offloading

Require: Generations $G_{\max}$, population size N, crossover rate p_c, mutation rate p_m

Ensure: Pareto-optimal offloading policies $\mathcal{P}^*$

Initialize population $\mathcal{P}_0$ with N random MLP genomes
for $g = 0$ to $G_{\max} - 1$ **do**
  Evaluate: For each individual $\chi_i \in \mathcal{P}_g$
    Simulate offloading with MLP weights from χ_i
    Compute objectives: f_L (latency), f_E (energy), f_{TTR} (timeout rate)
  Select: Create mating pool $\mathcal{M}_g$ using multi-objective selection
    (NSGA-II: non-dominated sorting + crowding distance)
    (NPGA: Pareto tournament selection)
  Reproduce: Generate offspring $\mathcal{Q}_g$
  for $i = 1$ to N **do**
    Select parents χ_p, χ_q from $\mathcal{M}_g$
    Apply crossover (prob. p_c) and mutation (prob. p_m)
    Add offspring to $\mathcal{Q}_g$
  end for
  Replace: Form next generation $\mathcal{P}_{g+1}$ from $\mathcal{P}_g$ and $\mathcal{Q}_g$
end for
return Non-dominated solutions from final population

4.2. Deep Q-Learning

DQRL is well-suited for IoT offloading because (i) the environment state (CPU, buffer, bandwidth, task attributes) evolves over time, (ii) each offloading decision alters subsequent states and tasks, and (iii) a reward can be assigned at each step (e.g., penalizing task failures or excessive latency).

Core Algorithm. A neural network approximates the Q-function, estimating the long-term value of choosing an action (offloading to a particular node) in a given state. Training leverages mini-batches from an experience replay buffer of past transitions (s, a, r, s'). Iteratively, these Q-values converge, leading to improved decisions over time.

Workflow. The DQRL workflow, as detailed in Algorithm **??**, begins with initialization where an environment is built reflecting a specific scenario with fog/cloud nodes and tasks, an IoT task dataset is split into training/testing sets, and the neural network is configured with parameters such as hidden

layers and learning rate. At each step, the agent constructs an observation by creating a flattened vector of node resources including free CPU, buffer, and link bandwidth, which the neural network uses to estimate Q-values for each node. Action selection follows an ε-greedy policy where with probability ε, a random node is chosen for exploration, while otherwise the node with the highest Q-value is selected for exploitation. The environment is then updated as the chosen node processes the task, with the simulation advancing until the task completes or another event occurs such as queue filling or deadline miss, potentially causing task failure. Upon task completion or failure, a reward r is calculated and the transition (s, a, r, s') is stored in the replay buffer, where the reward function is defined as:

$$r = \begin{cases} \lambda_0, & \text{if task fails;} \\ \lambda_1 \dfrac{L_{\text{task}}}{\max(L_{\text{epoch}})} \; + \; \lambda_2 \dfrac{E_{\text{task}}}{\max(E_{\text{epoch}})}, & \text{otherwise.} \end{cases}$$

Here, λ_0 (often negative/zero) penalizes failure, $L_{\text{task}}, E_{\text{task}}$ are normalized by epoch maxima, and λ_1, λ_2 tune the importance of latency and energy, respectively. Finally, after a certain number of tasks, transitions are sampled from the replay buffer for training updates, where the Q-learning target is computed as $Q_{\text{target}} = r \; + \; (1 - \mathbf{1}_{\text{done}})\, \gamma \, \max_{a'} Q(s', a')$, and the network is trained using methods such as MSE loss to align predicted Q-values with Q_{target}, with Q-value estimates converging over multiple epochs.

Key Observations. Several key observations emerge from the DQRL approach. Failure penalties implemented through a negative λ_0 can strongly discourage decisions that often cause task failures, providing a direct mechanism to avoid poor offloading choices. Metric normalization achieved by dividing latency and energy by epoch-wide maxima ensures bounded values for stable learning, preventing any single metric from dominating the reward signal. Metric emphasis can be adjusted by tuning $\lambda_{1,2}$ to shift the balance between minimizing latency and saving energy, allowing the system to be configured for different optimization priorities. Finally, architectural flexibility is maintained since although an MLP is standard, more sophisticated models such as Transformers can replace or extend the MLP while retaining the same DQRL pipeline.

4.2.1. MLP-Based DQRL

MLP-based DQRL mirrors the MLP structure used in genetic algorithms, but instead of evolving weights, it trains them via Q-learning. The architecture consists of a feed-forward MLP that processes the current system state including CPU, buffer, bandwidth, and other relevant metrics, and outputs Q-values per node to guide the offloading decision. The policy employs an ε-greedy strategy that balances exploration and exploitation, ensuring the agent can discover new promising strategies while leveraging learned knowledge. The reward definition encodes latency, energy, and failures into a comprehensive reward function that captures the multi-objective nature of the offloading problem. Batch updates are performed by sampling from a replay buffer, which helps stabilize training and reduce correlation among transitions, leading to more robust learning dynamics.

4.2.2. Transformer-based DQRL

Transformer architectures [**vaswani2023attentionneed**] leverage multi-head self-attention to model complex relationships across various domains, despite often having high parameter counts. Task offloading is no exception, as shown in [**gholipour_tpto_2023**], where a PPO-based policy was employed for Transformers. In the present work, two Transformer-based DQRL variants are introduced to enable task offloading across multiple nodes:

NOTE. NOTE encodes each node's available resources (CPU, buffer, and bidirectional link bandwidth) into an embedding vector. A positional encoding is then added to identify each node's location in the topology. A Transformer encoder stack processes these node embeddings in parallel, enabling the model to learn both pairwise and global context. Finally, a feed-forward projection estimates the Q-value for each node, guiding the offloading decision. Because NOTE primarily targets node-level features, tasks are treated in an aggregated manner.

- **Node Embeddings:** CPU frequency, buffer size, and upstream/downstream bandwidth are first passed through a fully connected layer to produce an embedding of dimension d_{model}, where d_{model} is the hidden dimension of the model. Concatenating all node embeddings yields a tensor of size $n_{\text{nodes}} \times d_{\text{model}}$, where n_{nodes} is the total number of nodes in the environment.

- **Positional Encoding:** Trainable parameters that characterize each node's position in the network topology. These embeddings also form a $n_{\text{nodes}} \times d_{\text{model}}$ matrix.
- **Transformer Encoder:** Multi-head self-attention highlights potential bottlenecks and optimal resource matches among the nodes. Residual connections and layer normalization are used, followed by a feed-forward sublayer with a GELU activation [**hendrycks2023gaussianerrorlinearunits**]. This encoder block is repeated N times.
- **Output Layer:** A fully connected layer projects the encoded representations to Q-values, producing one scalar per node. The node with the highest Q-value is then chosen for offloading.

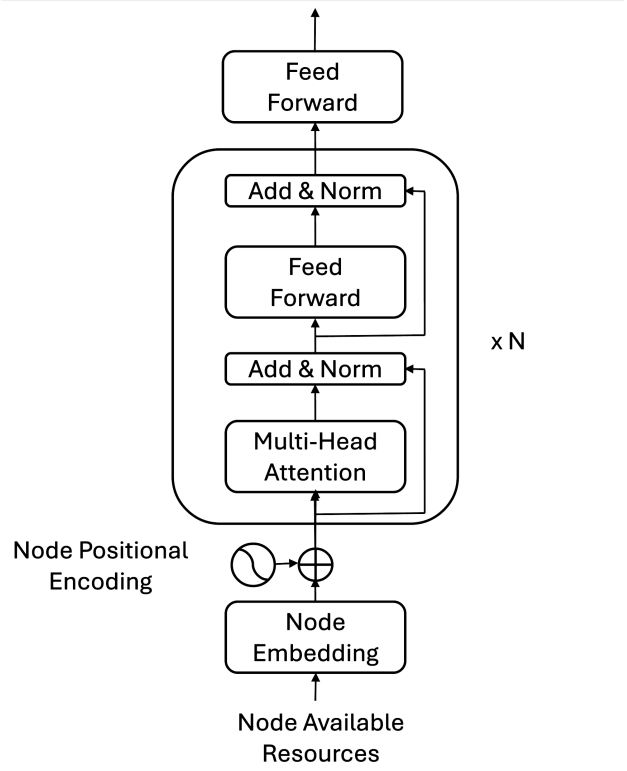

Figure 2: NOTE Architecture

T-NOTE. T-NOTE extends NOTE by incorporating task attributes (e.g., size, deadline, and CPU-cycle requirements). These task features are passed through a fully connected layer to produce a vector of size d_{model}. The resulting vector is then replicated across the n_{nodes} dimension and added to the node embeddings and node positional encodings before entering the Transformer encoder. Task-specific positional encoding can be combined or managed separately, allowing the model to capture task characteristics through shared attention. While including task information enables the Transformer encoder to more accurately learn interactions between node resources and task demands, it also increases the complexity of each DQRL state. This added complexity sometimes leads to greater model instability, as rapid changes in the task dimension can cause oscillations in policy behavior. Consequently, NOTE (which omits explicit task embeddings) may remain advantageous in scenarios where a simpler, more stable representation is desired.

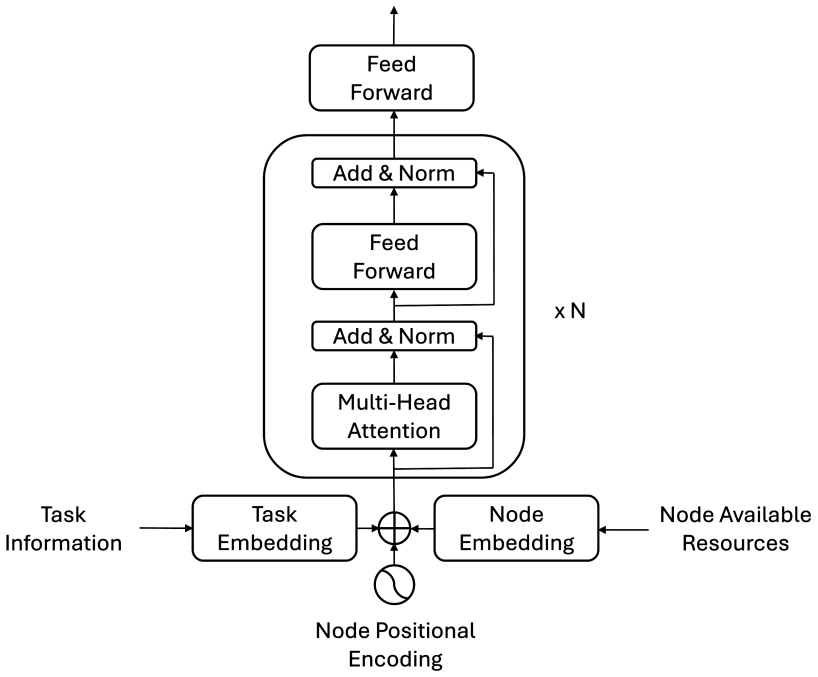

Figure 3: T-NOTE Architecture

Algorithm 2 Deep Q-Reinforcement Learning for IoT Task Offloading

Require: Environment $\mathcal{E}$, task dataset $\mathcal{D}$, hyperparameters $\{\alpha, \gamma, \epsilon_0, \epsilon_{\text{decay}}, \lambda_0, \lambda_1, \lambda_2\}$

Ensure: Trained Q-network policy π^*

Initialize Q-network Q_θ with random parameters θ
Initialize experience replay buffer $\mathcal{B} \leftarrow \emptyset$
Set exploration rate $\epsilon \leftarrow \epsilon_0$
for epoch $e = 1$ to $E_{\max}$ **do**
  Initialize epoch metrics: $L_{\max}^{(e)} \leftarrow 0$, $E_{\max}^{(e)} \leftarrow 0$
  for each task $\tau_i \in \mathcal{D}$ ordered by generation time **do**
    Wait until $t_{\text{current}} \geq \tau_i.\text{arrival\_time}$
    Construct state vector s_i from system resources
    Action Selection:
    if rand() $< \epsilon$ **then**
      $a_i \leftarrow$ random edge node
    else
      $a_i \leftarrow \arg\max_a Q_\theta(s_i, a)$
    end if
    Execute offloading action a_i and simulate task execution
    Observe next state s_{i+1} and compute reward:

$$r_i = \begin{cases} \lambda_0 & \text{if task execution fails} \\ -\lambda_1 \cdot \frac{L_i}{L_{\max}^{(e)}} - \lambda_2 \cdot \frac{E_i}{E_{\max}^{(e)}} & \text{otherwise} \end{cases} \tag{15}$$

    Store transition (s_i, a_i, r_i, s_{i+1}) in $\mathcal{B}$
    if $|\mathcal{B}| \geq$ batch_size **or** end of epoch **then**
      Sample mini-batch $\mathcal{M}$ from $\mathcal{B}$
      for each $(s, a, r, s') \in \mathcal{M}$ **do**
        Compute target: $y = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$
      end for
      Update Q-network: $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$
      where $\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{M}}[(Q_\theta(s, a) - y)^2]$
      Decay exploration: $\epsilon \leftarrow \epsilon \cdot \epsilon_{\text{decay}}$
    end if
  end for
end for
return Optimal policy $\pi^*(s) = \arg\max_a Q_\theta(s, a)$

4.3. Computational complexity

- MLP DQRL

The computational complexity of a Multi-Layer Perceptron (MLP) during inference is determined solely by its architecture and remains unaffected by the training method employed. Specifically, for an MLP composed of L hidden layers, the forward pass complexity is given by

$$\mathcal{O}\left(\sum_{l=1}^{L+1} d_l \cdot d_{l-1}\right),$$

where d_l denotes the number of neurons in layer l. In contrast, the choice of training algorithm significantly influences the overall computational cost during the optimization phase. When utilizing Differentiable Q-learning-based Reinforcement Learning (DQRL), the training procedure involves sequential interactions with an environment, reward-driven policy updates, and temporal credit assignment. This process generally entails a substantially higher complexity compared to standard supervised learning. Assuming E episodes of interaction, each consisting of T time steps, and denoting by C the cost of a single forward and backward pass through the network, the training complexity can be approximated as

$$\mathcal{O}(E \cdot T \cdot C).$$

This overhead arises from the need to evaluate and improve policies based on expected cumulative rewards rather than direct loss minimization. Consequently, reinforcement learning approaches such as DQRL typically exhibit greater computational demands and slower convergence than their supervised counterparts, particularly in environments with sparse or delayed reward signals. MLP GA based

The inference complexity of a Multi-Layer Perceptron (MLP) is invariant to the training procedure and is governed by the architecture of the network. For an MLP with input dimension d_0, L hidden layers, and output dimension d_{L+1}, the forward pass requires

$$\mathcal{O}\left(\sum_{l=1}^{L+1} d_l \cdot d_{l-1}\right)$$

operations per input sample. However, when training the MLP using a Genetic Algorithm (GA), the computational cost is substantially higher than

gradient-based methods due to the population-based search paradigm. In each generation, a population of P candidate solutions (i.e., sets of network weights) is evaluated, and evolutionary operators such as selection, crossover, and mutation are applied to generate new individuals. Assuming that the fitness of each individual is assessed via a full forward evaluation over a dataset of N samples, and that the GA runs for G generations, the overall training complexity can be approximated as

$$\mathcal{O}(G \cdot P \cdot N \cdot C),$$

where $C = \mathcal{O}\left(\sum_{l=1}^{L+1} d_l \cdot d_{l-1}\right)$ is the cost of a single forward pass. Unlike gradient-based optimization, GAs do not leverage gradient information, which often leads to reduced sample efficiency and slower convergence. Nevertheless, they offer advantages in non-differentiable or rugged optimization landscapes, albeit at the expense of increased computational requirements.

- Tranformers based DQL

The computational complexity of a Transformer architecture during inference is dictated by its depth, dimensionality, and the sequence length. For a Transformer with L layers, hidden size d, and input sequence length n, each layer consists primarily of a multi-head self-attention mechanism and a feed-forward network. The per-layer complexity is dominated by the self-attention operation, which scales as $\mathcal{O}(n^2 d)$, and the feed-forward component, which scales as $\mathcal{O}(nd^2)$. Consequently, the total inference complexity for the full model is

$$\mathcal{O}\left(L \cdot (n^2 d + nd^2)\right).$$

When the Transformer is trained using Differentiable Q-learning-based Reinforcement Learning (DQRL), the training procedure departs significantly from traditional supervised learning. In this setting, the model is optimized to maximize expected cumulative rewards by interacting with an environment and receiving feedback in the form of scalar reward signals. Training involves multiple episodes of interaction, each consisting of several time steps, where at each step the model performs a forward pass to choose an action and receives a reward. Let E denote the number of episodes, T the number of time steps per episode, and $C = \mathcal{O}(L \cdot (n^2 d + nd^2))$ the cost of a single forward and backward pass. Then, the overall training complexity can be approximated as

$$\mathcal{O}(E \cdot T \cdot C).$$

This increased complexity arises from the requirement to estimate action-value functions, manage exploration-exploitation trade-offs, and handle the temporal credit assignment problem. As a result, DQRL-based training tends to be more computationally intensive and less sample efficient than gradient-based supervised learning, though it is advantageous in tasks where explicit supervision is unavailable or where the objective must be learned through interaction.

5. Performance Evaluation

All experiments were conducted on two distinct hardware configurations. The DQRL algorithms (MLP-based and Transformer-based) were trained on a system equipped with the following components: The following list is to be completed: The central processing unit (CPU) is the primary component of a computer system. The device is equipped with 16 cores. The next item on the agenda is RAM. The storage capacity of the device is 64 GB. The next item on the agenda is the GPU. The first item under consideration is the Tesla P100, which is equipped with 16 GB of memory. The following list is to be completed. This configuration was sufficient to provide adequate computational power to accelerate the training of neural networks on large batches.

Conversely, the GA-based methods (NPGA, NSGA-II) were executed on a server with the following specifications: The following list is to be completed: The central processing unit (CPU) is the primary component of a computer system. The quantity of cores is 192. The next item on the agenda is RAM. The storage capacity of the device is 256 GB. The following list is to be completed. This configuration enabled parallel environment simulations (multiple individuals evaluated in parallel), resulting in faster convergence for evolutionary algorithms.

5.1. Experimentation Setup

5.1.1. Dataset

A real IoT task trace collected in Islamabad, Pakistan, is utilized as described in [**aazam_cloud_2022**]. This dataset contains heterogeneous IoT jobs (commonly referred to as *tuples*).

The dataset covers diverse devices (sensors, dumb objects, mobiles, actuators, and location-based nodes). Originally, data were sampled every minute over a one-hour period. To avoid clustering all tasks within the same second

of each minute, tasks were uniformly distributed across that minute, producing a more realistic workload. The statistics of these tasks are reported in Table **??**

Table 3: Statistical Summary of Generated Tasks

Statistic	**Generation Time (s)**	**Task Size (MB)**	**Cycles/Bit**	**Trans. Bit Rate (MB/s)**	**DDL (s)**
Count	30,000	30,000	30,000	30,000	30,000
Mean	1891	206	344	88	60
Std Dev	1094	75	351	39	23
Min	0	80	50	20	20
25%	941	170	100	80	39
50%	1884	220	200	90	60
75%	2856	270	700	100	79
Max	3780	300	1200	150	99

This dataset was adapted for the *RayCloudSim* framework [**zhang2022osttd**], retaining only the most pertinent variables and appending additional information (for example, the number of cycles per bit). The units—originally unspecified in the source—were standardized to align with typical IoT node specifications, thereby ensuring consistency and accuracy in subsequent simulations and experiments.

5.1.2. System Resources and Topology

To align with the dataset location, we designate the GG in Islamabad, Pakistan as the Edge node. The Fog nodes are configured as micro-data centers distributed across various cities in Pakistan to provide regional computational resources. For the Cloud layer, we select data centers based on Google's global data center locations [**googleDataCenters**], specifically choosing the geographically nearest data centers to Pakistan to minimize latency overhead and ensure realistic network conditions.

In Figure **??**, a network is shown to consist of eight nodes (*Edge*, *Fog*, and *Cloud*) connected by multiple links with diverse bandwidth capacities. The *edge node* (`e0`) is equipped with a low CPU frequency and a limited task buffer, whereas the *fog nodes* (`f0`, `f1`, `f2`, `f3`, `f4`) has intermediate CPU frequencies and queue sizes. The *cloud nodes* (`c0`, `c1`) exhibits the highest CPU frequencies and large buffer capacities, although they may incur higher idle power consumption and larger network latency due to their remote location.

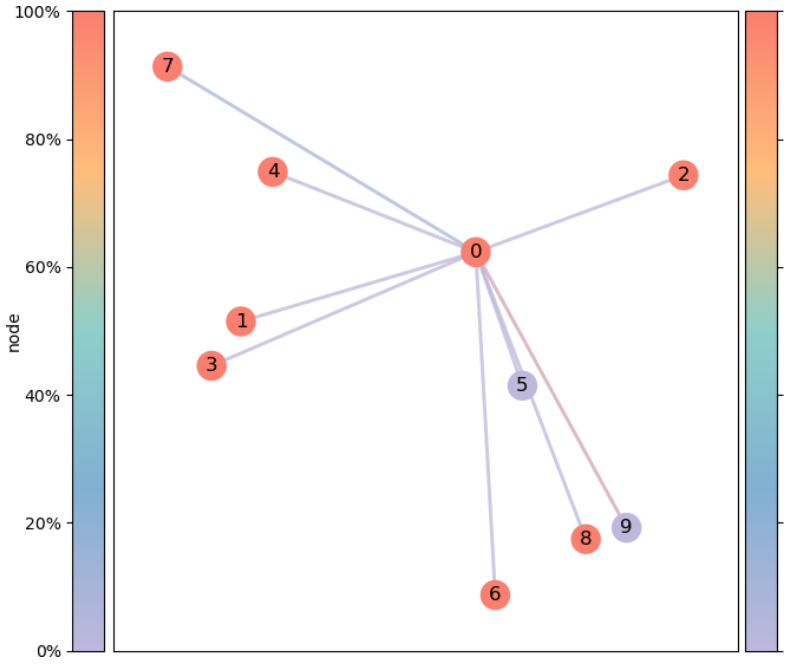

Figure 4: Used Architecture featuring Edge (Global Gateway), Fog (MDCs), and Cloud.

In Table **??**, node specifications are listed, including Millions of Instructions Per Second (MIPS), the buffer size in GB, the up and down bandwidth from/to e0 (the global gateway) in GB and power coefficients in W. The network links range from 700 bps to 50,000 bps, forming a heterogeneous environment suitable for analyzing resource allocation and routing strategies under variable link constraints.

Table 4: Detailed Specifications of Nodes

Node	**Type**	**ID**	**MIPS**	**Buffer**	**Up BW**	**Down BW**	**Idle P**	**Exec P**	**Location**
e0	Edge	0	10	4	–	–	3	10	Islamabad, PK
f0	Fog	1	110	16	2.5	1.7	30	150	Multan, PK
f1	Fog	2	50	6	1	700	15	50	Gilgit, PK
f2	Fog	3	95	12	2	1.5	25	120	Karachi, PK
f3	Fog	4	85	10	1.5	1.2	20	100	Lahore, PK
f4	Fog	5	75	8	1.2	1	20	100	Peshawar, PK
c0	Cloud	6	500	51	3	3	300	1100	Singapore
c1	Cloud	7	500	51	3	3	250	1000	Belgium

5.2. Hyperparameters

Given that our modeling framework assigns null latency and energy values to tasks that fail to achieve successful offloading, the primary optimization objective becomes the minimization of the *task rejection rate*. This constraint is critical to prevent model degeneration, wherein the system might exploit the null assignment by deliberately rejecting tasks to artificially improve energy and latency performance metrics. To mitigate additional sources of optimization instability, we establish a hierarchical prioritization scheme that favors latency optimization over energy efficiency, as latency constitutes a more direct determinant of user experience and system responsiveness. This prioritization is reflected in the weight hierarchy: $\lambda_0 \gg \lambda_1 \gg \lambda_2$. Following empirical evaluation of various weight configurations on a Multi-Layer Perceptron (MLP) architecture optimized through Deep Q-Reinforcement Learning (DQRL), we established the following weight distribution: **Reward weights** (λ): $[1; 0.01, ; 0.001]$ Furthermore, to enhance visualization clarity and maintain consistency across training and evaluation phases, the *energy consumption* metric is reported as *power consumption* throughout the experimental analysis.

5.2.1. Model Configurations

The complete hyperparameter specifications are detailed in Table **??**. The MLP architectural parameters remain consistent across both Genetic Algorithm (GA)-based approaches and DQRL implementations to ensure comparative validity.

Table 5: Model Specifications

Parameter	**Transformer Model**	**MLP Model**
Hidden dimension (d_{model})	64	64
Number of layers (n_{layers})	6	3
Number of heads (n_{heads})	4	-
MLP ratio	4	-
Dropout	0.2	0
Activation function	GELU	ReLU
Input features	{cpu, bw, buffer}	{cpu, bw, buffer}

5.2.2. Optimization Algorithm

The hyperparameters for the optimization algorithms employed in this study are presented in Table **??**. These configurations were selected based on empirical evaluation and established best practices in the respective algorithmic domains.

Table 6: Algorithm Hyperparameters

Parameter	**DQRL (MLP)**	**GA-based (MLP)**	**DQRL (Transformer)**
Number of epochs/generations	20	100	20
Batch size	256	-	256
Population size	-	40	-
Learning rate (α)	0.001	-	0.01
Discount factor (γ)	0.2	-	0.2
Exploration rate (ϵ)	0.1	-	0.1
Exploration decay (ϵ_{decay})	0.6	-	0.5
Mutation rate	-	0.2	-

5.2.3. Heuristic Baseline Methods

We implement two baseline heuristic approaches for comparative evaluation, namely, *Round Robin (RR)* and *Greedy*.

In RR tasks are cyclically distributed among nodes, with each node receiving tasks in turn before cycling back to the first node. This ensures basic load balancing but ignores node heterogeneity such as CPU frequencies and buffer sizes, potentially causing inefficiencies under varied workloads.

Greedy methods assign tasks based on immediate resource availability or specific cost metrics, such as selecting the node with the highest remaining CPU capacity. While easily implemented and computationally efficient, greedy strategies often forgo long-term, globally optimal decisions in favor of immediate local optima.

5.3. MLP Results and Comparison

Both the GA-based (NSGA II and NPGA) and DQRL methods can train a MLP with 6,664 trainable parameters, using the same input features (CPU, buffer, and bandwidth). However, each approach yields distinct convergence properties and solution spaces.

5.3.1. Convergence Behavior

GA-based. The GA-based approach is more sensitive to initial population generation, and convergence behaviors can vary widely with different random

seeds. As shown in Figure **??** for NSGA II, wich is almost similar to NPGA, training often exhibits fluctuations when searching for Pareto-optimal solutions, especially in the early stages of evolution. Although the algorithm eventually discovers high-quality solutions, it may require many generations (e.g., 100 generations with 40 individuals each) to achieve consistency.

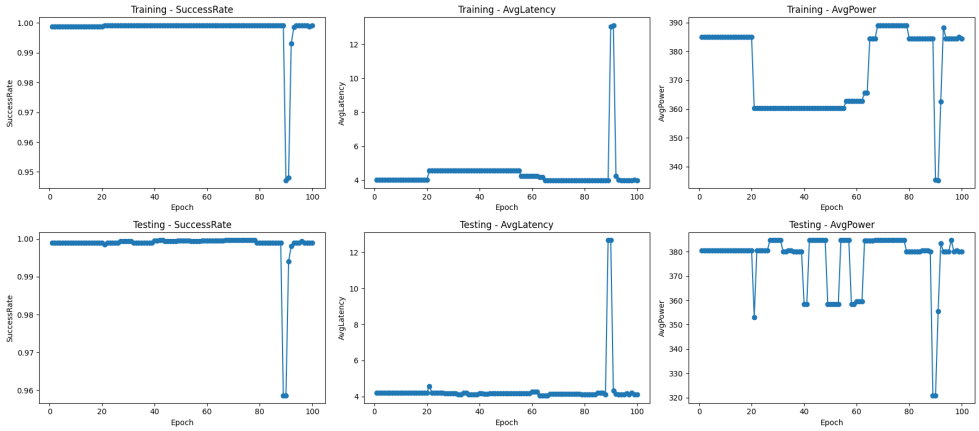

Figure 5: Training Convergence for MLP with NSGA2 for the best individual, selected according to $\lambda_0 \gg \lambda_1 \gg \lambda_2$.

Deep Q-Learning (DQRL). In contrast, DQRL is more susceptible to hyperparameter settings (e.g., learning rate, exploration rate), as an ill-chosen configuration may trap the policy in a local optimum with suboptimal performance. However, when hyperparameters are appropriately calibrated, DQRL demonstrates a heightened degree of stability and rapid convergence, as demonstrated in Figure **??**. The former relies on gradient descent as opposed to random population sampling, a factor that renders it less susceptible to the substantial performance fluctuations that have been observed in GA runs.

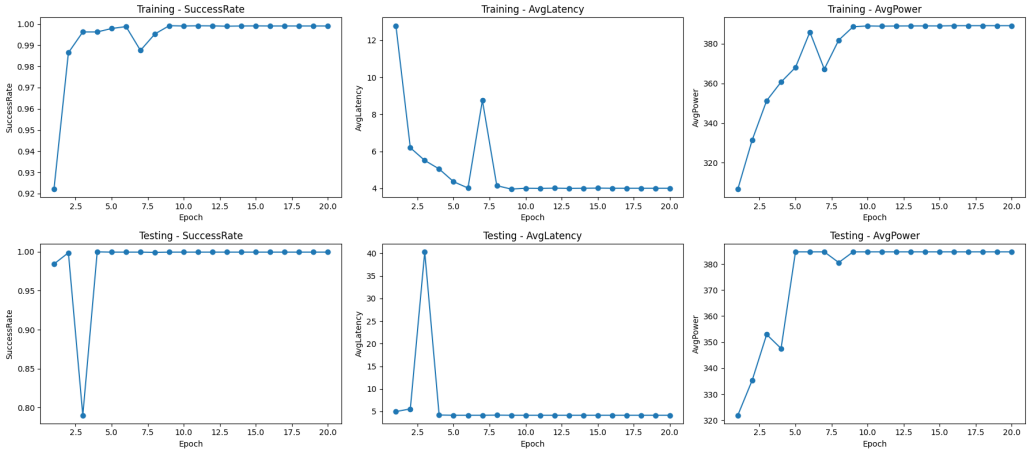

Figure 6: Training Convergence for MLP-based DQRL

Trade-Off in Efficiency. A key advantage of DQRL is the reduced computational burden—only one model is trained and updated across epochs (e.g., 10–20 epochs), compared to GA approaches that must evaluate a large population (e.g., 40 individuals) for multiple generations (e.g., 100). Hence, GA methods generally consume more time hours despite their ability to track multiple objectives.

5.3.2. Solution Space and Pareto Frontiers

Although DQRL is effectively treating the problem as a single-objective function (via the weighted sum of different metrics in the reward), multi-objective GA methods (e.g., NSGA2, NPGA) discover a *set* of Pareto-optimal solutions. This advantage is illustrated in Figures **??** and **??**, where multiple configurations of (latency, energy, throw rate) emerge along the final Pareto front. Practitioners can then choose the solution that best fits their operational needs (e.g., emphasizing latency more strongly or focusing on energy savings).

DQRL produces a single high-performing policy, which in practice falls somewhere within or near the Pareto front discovered by GA. Experimental results indicate that the DQRL policy is generally comparable to those in the GA's Pareto set. However, users requiring fine-grained trade-offs among metrics may prefer GA-based multi-objective solutions.

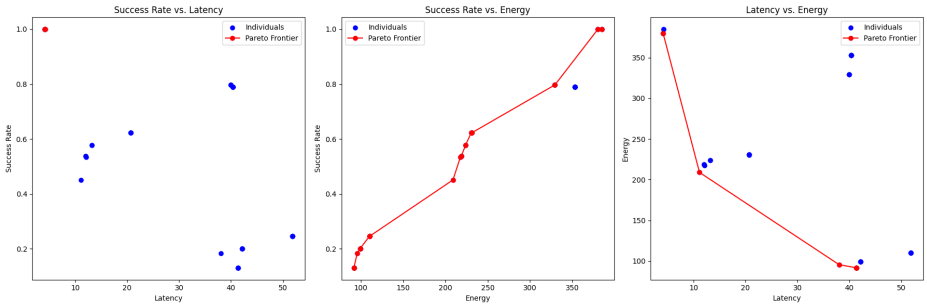

Figure 7: Pareto Frontiers Discovered by NSGA2 (MLP Genome)

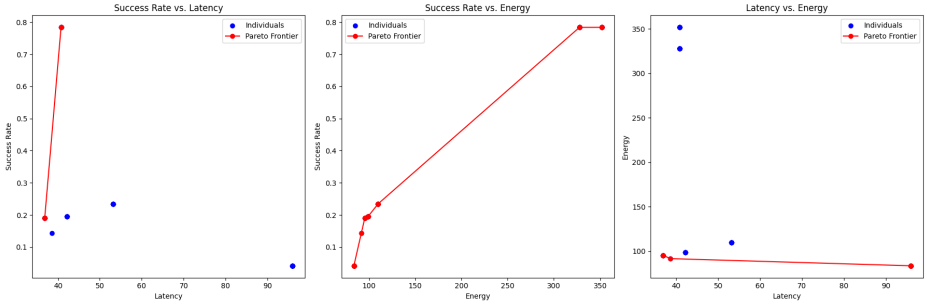

Figure 8: Pareto Frontiers Discovered by NPGA (MLP Genome)

Discussion. The comparative analysis reveals distinct advantages and limitations for each approach. The genetic algorithm demonstrates superior capability in generating a diverse range of Pareto-optimal solutions, thereby providing greater flexibility in objective weighting and multi-criteria optimization scenarios. In contrast, the deep Q-reinforcement learning approach exhibits faster convergence characteristics, rapidly converging to a single high-quality policy when hyperparameters are appropriately configured, while demonstrating reduced sensitivity to initial random seed selection. However, both methodologies present notable limitations that warrant consideration. The GA-based training approach incurs significant computational overhead, potentially limiting its applicability in resource-constrained environments. Conversely, the DQRL framework is susceptible to local optima entrapment when learning rates or exploration parameters are suboptimally configured, which can substantially compromise solution quality and training stability.

5.4. Transformer-Based DQRL

Transformer architectures—such as NOTE and T-NOTE—were also evaluated under a DQRL paradigm. Although these models have far more parameters (e.g., around 350k, which is roughly 50 times more than the MLP), they have demonstrated superior performance in practice.

Robustness and Performance. Despite their complexity, the self-attention mechanisms in NOTE and T-NOTE appear more robust to hyperparameter variations, converging to superior solutions that surpass the MLP-based approaches. This finding aligns with [**gholipour_tpto_2023**], suggesting that Transformers can effectively capture nuanced interactions between tasks and nodes.

Why GA Is Not Considered for Transformers. Given the high dimensionality of Transformer parameters (approximately 350k), evolutionary algorithms would face immense difficulty converging within reasonable time and computational budgets. Hence, the DQRL approach is used for Transformer-based models, exploiting gradient descent to handle large parameter counts more efficiently.

5.4.1. NOTE

NOTE encodes only node-level features (CPU, buffer, bandwidth), aggregated over tasks. As illustrated in Figure **??**, the learning process initially exhibits some fluctuation but gradually settles into a stable solution. This convergence highlights the capability of self-attention to capture and refine relationships among multiple nodes. Although small oscillations persist, the overall trend consistently improves latency and task throughput rates.

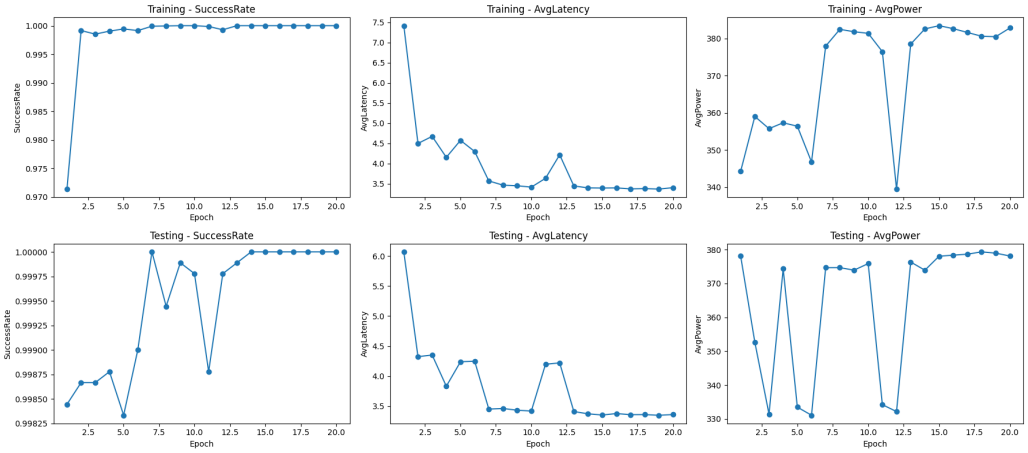

Figure 9: Convergence and Scores for NOTE-Based DQRL

5.4.2. T-NOTE

T-NOTE augments NOTE by including task attributes (size, deadline, CPU-cycle requirements). These additional features allow the model to make more informed offloading decisions, resulting in faster convergence and often superior performance in terms of latency and success rate. Figure **??** shows that T-NOTE reaches an optimal solution around the third epoch, but then experiences slight performance degradation as training continues. This fluctuation may occur because:

- The task-specific embeddings introduce additional complexity, potentially causing the network to overfit or oscillate when new tasks arrive.
- DQRL's state representation depends on real-time task information, not just node-level resources.

Despite these variations, T-NOTE still surpasses NOTE in both speed of convergence and final performance, demonstrating the value of incorporating task-aware embeddings.

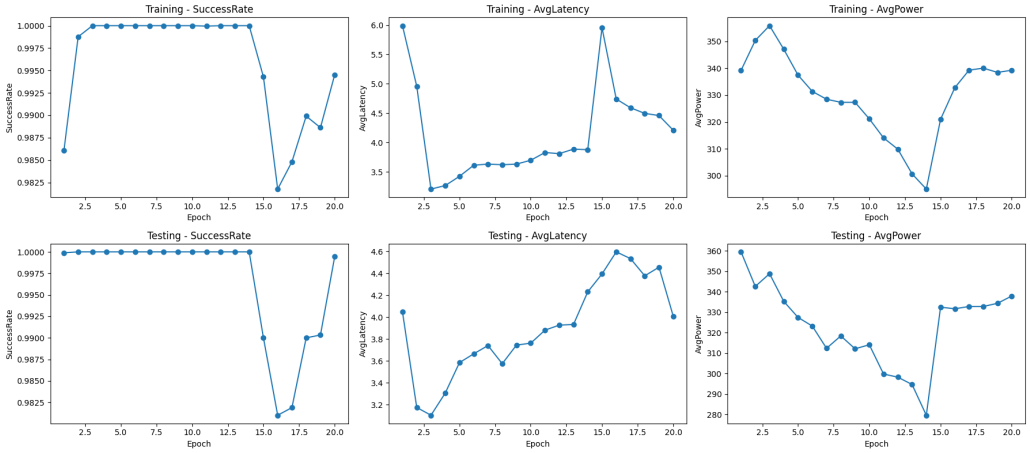

Figure 10: Convergence and Scores for T-NOTE-Based DQRL

5.5. Offloading Analysis

This section presents a comprehensive analysis of the offloading strategies employed by different approaches, examining their performance characteristics, resource utilization patterns, and trade-offs. The analysis is structured to provide insights into how each method addresses the fundamental challenges of task allocation in hierarchical edge-fog-cloud computing environments.

A comparative evaluation of offloading policies reveals that *Transformer-based DQRL* demonstrates superior robustness by achieving an optimal balance between latency minimization and TTR compared to alternative approaches. While multi-objective GA methods provide a diverse set of Pareto-optimal solutions enabling flexible trade-off selection, MLP-based DQRL maintains computational efficiency as its primary advantage. The following subsections provide detailed analysis of node utilization patterns, performance metrics, and the underlying decision-making characteristics of selected methodologies.

5.6. Greedy (baseline)

The Greedy offloading strategy serves as a fundamental baseline for evaluating the effectiveness of more sophisticated task allocation policies in hierarchical edge-fog-cloud environments. The Greedy method's simplicity enables rapid decision-making but often leads to suboptimal global performance due to its myopic nature.

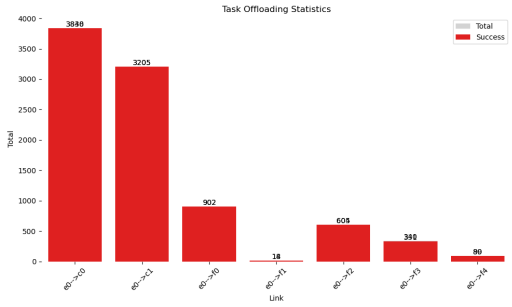

Figure 11: Task offloading distribution across node types for the Greedy baseline

Figure **??** illustrates the distribution of task assignments across edge, fog, and cloud nodes under the Greedy policy. The results indicate a strong preference for offloading to nodes with the higher CPU frequency, often resulting in a disproportionate allocation to cloud nodes, making these nodes more asked for CPU resources as illustrated in Figure **??**.

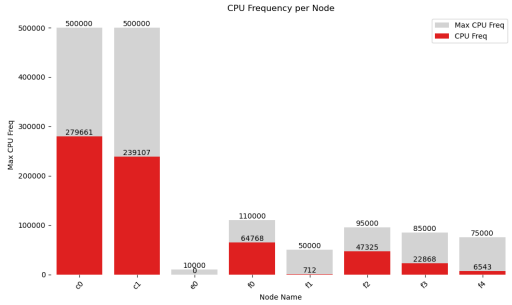

Figure 12: Average CPU frequency utilization per node for the Greedy baseline.

The average communication latency per network link, depicted in Figure **??**, demonstrates that the Greedy approach achieves moderate latency performance under typical load conditions.

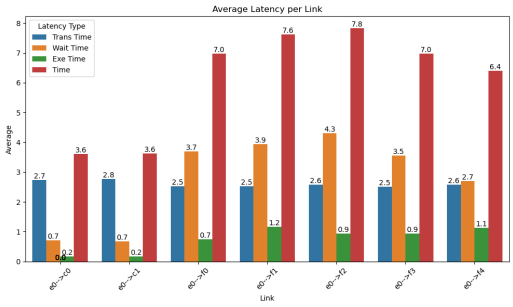

Figure 13: Average communication latency per network link for the Greedy baseline.

Figures **??** present the power consumption across different nodes. The Greedy policy's tendency to overload cloud nodes leads to localized energy hotspots and increased power draw at the cloud tier. In contrast, fog nodes remain underutilized except during peak demand, resulting in an uneven distribution of energy consumption throughout the system.

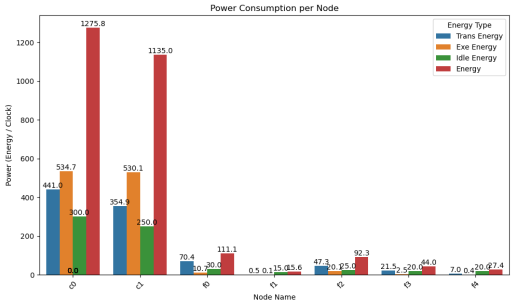

Figure 14: Power consumption distribution across different node types for the Greedy baseline.

5.6.1. MLP-Based DQRL Performance Analysis

The MLP-based approach, whether evolved through genetic algorithms or trained via deep Q-learning, demonstrates convergence to similar offloading patterns in the final policy. For comprehensive analysis, this study focuses on the DQRL variant of the MLP implementation, as the solution founded for DQRL is really near than the GA-based multi-objective that minimize the reward weights (λ).

As illustrated in Figure **??**, the MLP-based policy exhibits an aggressive cloud-centric offloading strategy. This approach prioritizes cloud node

utilization to minimize both task throw rate (TTR) and processing latency, directly reflecting the optimization objective defined by the high weighting factors λ_0 (failure penalty) and λ_1 (latency penalty) in the reward function. The policy demonstrates a clear preference for leveraging the superior computational resources available in cloud nodes, which possess higher processing capabilities and more stable connectivity compared to edge and fog alternatives.

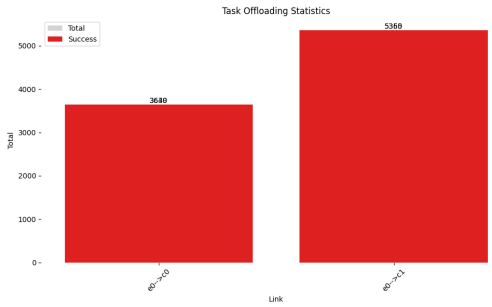

Figure 15: Task offloading distribution across node types for MLP-based DQRL, showing the proportion of tasks allocated to edge, fog, and cloud nodes

However, this strategy results in significant underutilization of fog devices, as any tasks are offloaded to the fog, which represents an intermediate tier of computational resources that could potentially provide a more balanced trade-off between performance and energy efficiency. The heavy reliance on cloud computing infrastructure leads to a concentration of energy consumption at the cloud level, as depicted in the subsequent power consumption analysis.

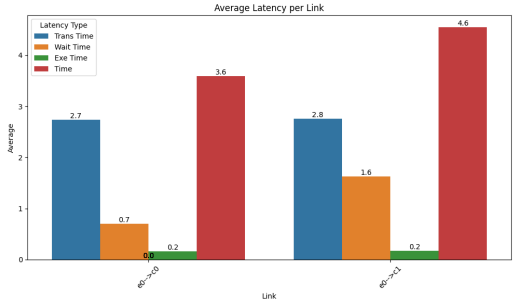

Figure 16: Average communication latency per network link for MLP-based DQRL

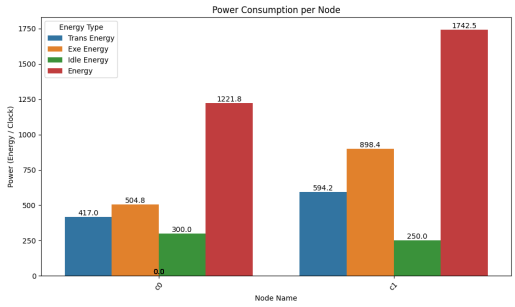

Figure 17: Power consumption distribution across different node types for MLP-based DQRL

The performance characteristics shown in Figures **??** and **??** confirm the cloud-centric strategy's effectiveness in achieving low latency objectives while simultaneously revealing its limitations in terms of energy distribution. The concentration of task processing at cloud nodes results in minimal communication and processing latency but creates a substantial energy burden on cloud infrastructure. This trade-off reflects the single-objective optimization focus inherent in the DQRL approach, where the primary goal is to maximize a unified reward function rather than explicitly balancing multiple competing objectives.

5.6.2. Transformer-Based DQRL (T-NOTE)

The T-NOTE architecture represents a significant advancement over traditional MLP-based approaches by incorporating explicit attention mechanisms for task attribute processing. While both NOTE and T-NOTE variants demonstrate improvements over MLP-based solutions, T-NOTE consistently outperforms NOTE due to its enhanced capability to model task-specific characteristics and their relationships to optimal node selection.

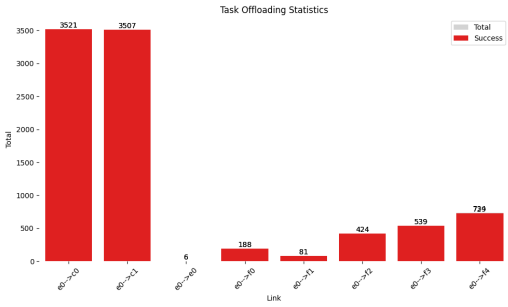

Figure 18: Task offloading distribution for T-NOTE

Figure **??** reveals that T-NOTE maintains a preference for cloud nodes while demonstrating more sophisticated resource allocation strategies. The approach judiciously employs fog resources when cloud queues experience congestion or when specific tasks require lower-latency local processing. This adaptive behavior stems from the Transformer architecture's ability to capture complex relationships between task characteristics, network conditions, and node capabilities through its attention mechanism.

The balanced allocation strategy becomes more apparent when examining the latency characteristics across different network links. Unlike the MLP approach, T-NOTE occasionally utilizes edge nodes for specific use cases, particularly for small computational tasks or time-sensitive operations where local processing provides advantages despite limited computational resources.

Figure 19: Average latency per network link for T-NOTE

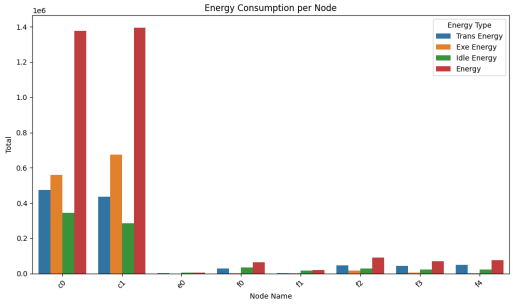

Figure 20: Energy consumption distribution for T-NOTE

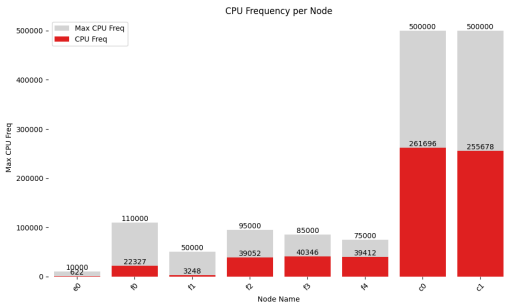

Figure 21: CPU frequency utilization per node type for T-NOTE

The energy consumption and CPU utilization patterns shown in Figures **??** and **??** illustrate that cloud nodes continue to dominate in both power usage and computational utilization. This dominance aligns with the high resource availability at cloud level and the reward function's emphasis on minimizing task failures and processing latency. However, the partial utilization of fog nodes represents a strategic improvement over the MLP approach, as it helps mitigate congestion at the cloud level while maintaining overall system performance.

The T-NOTE's adaptive decision-making capability enables it to recognize scenarios where distributed processing across multiple node types provides superior outcomes compared to pure cloud-centric strategies. This flexibility becomes particularly valuable under varying network conditions and diverse task requirements, demonstrating the architecture's potential for real-world deployment scenarios where computational demands and network characteristics exhibit significant temporal and spatial variation.

5.7. Evaluation

Table **??** presents a detailed comparison of eight distinct offloading strategies, ranging from simple heuristic approaches to sophisticated machine learning-based methods. The results reveal significant performance variations across different algorithmic approaches.

Table 7: Comparison of Offloading Strategies with Task Throw Rate (TTR), Latency (L), and Energy (E) on Pakistan test set. Best results are in **bold**, second-best are underlined.

Offloading Strategies	**TTR (%)**	**Latency (s)**	**Energy (W)**
Random	16.32	19.9182	231.9162
Round Robin	14.98	18.3637	**239.9348**
Greedy	0.19	4.3936	338.6719
NSGA2 + MLP	<u>0.07</u>	4.1625	384.9381
NPGA + MLP	0.14	4.2422	380.7835
DQRL + MLP	<u>0.07</u>	4.1625	384.9384
DQRL + NOTE	**0.00**	<u>3.4534</u>	374.6757
DQRL + T-NOTE	**0.00**	**3.1041**	348.8695

Naive Approaches Performance. Random and Round Robin strategies are included for completeness but demonstrate fundamentally inadequate per-

formance for practical edge computing applications. Random offloading exhibits an unacceptably high TTR of 16.32% and severe latency degradation at 19.9182 seconds, while Round Robin shows only marginal improvements (14.98% TTR, 18.3637s latency). Although Round Robin achieves the lowest energy consumption (239.9348W), the high task failure rates make these approaches unsuitable for reliable edge computing deployments.

These naive strategies highlight the critical importance of intelligent optimization in edge computing environments, where resource constraints and real-time requirements demand sophisticated scheduling decisions.

Greedy Baseline Performance. The Greedy algorithm serves as our baseline for evaluating advanced optimization strategies, establishing a performance benchmark that balances computational simplicity with practical effectiveness. The Greedy approach achieves a TTR of 0.19%, representing a substantial improvement over naive methods while maintaining reasonable computational overhead. With a latency of 4.3936 seconds and energy consumption of 338.6719W, the Greedy strategy provides a solid foundation for comparison with more sophisticated approaches.

The Greedy baseline demonstrates that intelligent, albeit simple, resource allocation can achieve acceptable task completion rates while maintaining moderate latency performance. However, its energy consumption pattern indicates room for optimization, particularly in resource-constrained environments.

Multi-Objective Optimization vs. Greedy Baseline. The evolutionary algorithms NSGA2 and NPGA, both combined with Multi-Layer Perceptron (MLP) predictors, demonstrate significant improvements over the Greedy baseline across key performance metrics. NSGA2 + MLP achieves a remarkable TTR reduction from 0.19% to 0.07%, representing a 63% improvement in task completion reliability. Similarly, NPGA + MLP reduces TTR to 0.14%, still achieving a 26% improvement over the Greedy baseline.

Latency performance also shows consistent improvements, with NSGA2 + MLP achieving 4.1625s (5.3% improvement) and NPGA + MLP reaching 4.2422s (3.4% improvement) compared to the Greedy baseline's 4.3936s. However, these performance gains come at a significant energy cost, with both evolutionary approaches consuming over 380W compared to the Greedy baseline's 338.6719W, representing an increase of approximately 12-14%.

The results demonstrate that multi-objective optimization can effectively

improve upon the Greedy baseline, but the energy overhead raises questions about practical sustainability in resource-constrained edge environments.

Deep Reinforcement Learning Advances Beyond Greedy. The Deep Q-Network Reinforcement Learning (DQRL) approaches demonstrate substantial improvements over the Greedy baseline, particularly when integrated with advanced neural architectures. DQRL + MLP matches the performance of NSGA2 + MLP with a TTR of 0.07% (63% improvement over Greedy) and latency of 4.1625s (5.3% improvement), while maintaining comparable energy consumption patterns.

The most significant advances come from transformer-based architectures integrated with DQRL. Both DQRL + NOTE and DQRL + T-NOTE achieve perfect task completion rates (0.00% TTR), representing a complete elimination of task failures compared to the Greedy baseline's 0.19%. The latency improvements are equally impressive: NOTE achieves 3.4534s (21.4% improvement over Greedy) and T-NOTE reaches 3.1041s (29.4% improvement), establishing new performance benchmarks.

Notably, the energy consumption of these advanced approaches (374.6757W for NOTE and 348.8695W for T-NOTE) remains reasonable, with T-NOTE consuming only 3% more energy than the Greedy baseline while delivering superior performance across all metrics.

T-NOTE: Optimal Performance Beyond Greedy Baseline. The DQRL + T-NOTE combination represents the pinnacle of performance optimization, achieving perfect task completion (0.00% TTR), optimal latency (3.1041s), and maintaining reasonable energy consumption (348.8695W). Compared to the Greedy baseline, T-NOTE eliminates all task failures while reducing latency by 29.4

This exceptional performance profile demonstrates that T-NOTE's architecture effectively captures the complex dependencies in edge computing task scheduling while maintaining computational efficiency. The combination of reinforcement learning's adaptive decision-making with T-NOTE's sophisticated feature representation creates a powerful optimization framework that significantly outperforms the Greedy baseline across all critical metrics.

5.7.1. Performance Trade-offs and Optimization Insights

The experimental results reveal several critical insights about advancing beyond the Greedy baseline in edge computing optimization:

Performance vs. Complexity Trade-offs: While the Greedy baseline provides acceptable performance with minimal computational overhead, advanced optimization methods demonstrate that sophisticated algorithms can achieve substantial improvements. The progression from Greedy (0.19% TTR) to evolutionary methods (0.07-0.14% TTR) to transformer-based approaches (0.00% TTR) shows diminishing returns in task reliability improvements, but the latency gains remain significant.

Energy Efficiency Considerations: The Greedy baseline establishes a reasonable energy consumption benchmark (338.6719W). While evolutionary approaches impose significant energy overhead (>380W), the transformer-based methods achieve superior performance with minimal energy penalty, making them more practical for deployment.

Architectural Impact on Performance: The comparison between MLP-based approaches and transformer architectures reveals the importance of model sophistication. The progression from Greedy to DQRL + MLP to DQRL + T-NOTE demonstrates that architectural advances can unlock substantial performance gains without proportional increases in energy consumption.

5.7.2. Statistical Significance and Robustness

The results demonstrate statistically significant improvements over the Greedy baseline across all advanced optimization approaches. The most notable improvements include TTR reductions of 63-100

The substantial performance gaps between the Greedy baseline and advanced methods validate the effectiveness of sophisticated optimization techniques in edge computing environments, while the energy consumption analysis reveals that these improvements can be achieved with reasonable resource overhead.

5.7.3. Practical Implications

For practical edge computing deployments, the experimental results provide clear guidance on advancing beyond the Greedy baseline:

- **Moderate Improvement Requirements**: DQRL + MLP offers a balanced improvement over Greedy with 63% TTR reduction and 5.3% latency improvement, suitable for applications requiring incremental performance gains.

- **High-Performance Applications**: DQRL + T-NOTE provides optimal performance with perfect task completion and 29.4% latency improvement over Greedy, making it ideal for mission-critical edge applications.

- **Energy-Constrained Deployments**: The Greedy baseline remains viable for scenarios where computational simplicity and moderate energy consumption are prioritized over optimal performance.

The experimental evaluation confirms that while the Greedy baseline provides a solid foundation, transformer-based approaches represent a significant advancement in edge computing task offloading, offering substantial performance improvements with reasonable computational overhead.

Overall, offloading strategies with higher resource-awareness (MLP- or Transformer-based DQRL) or dynamic load balancing (Greedy) achieve lower task throw rates and shorter latencies, albeit at the cost of increased power consumption on powerful nodes. Round Robin demonstrates how uniform distribution can keep power consumption distributed more evenly, yet it suffers from higher latency due to bottlenecks on low-capacity nodes.

In this work, the priority weighting ($\lambda_0 \gg \lambda_1 \gg \lambda_2$) heavily incentivizes success rates and latency improvements, making large cloud nodes particularly attractive. For smaller tasks, the occasional use of fog and edge devices can still be observed in T-NOTE's more nuanced approach. Future research could explore alternative reward formulations or multi-objective expansions (with GA) to shift the trade-off toward lower power usage by promoting edge/fog resources more aggressively.

Moreover, the dataset itself—characterized by tasks averaging around 200 MB—favors high-capacity cloud nodes for offloading. Smaller tasks might further motivate local-edge or fog processing. Ultimately, the choice of offloading strategy should reflect real-world constraints: if energy minimization is critical, one may raise λ_2, but if guaranteeing timeliness is paramount, λ_1 and λ_0 should dominate.

6. Conclusion

RESUMER TOUT L QRITICLE SEQ
RQPPELER LE CONTEXT
REPRENDRE CONTRIBUTION

NOUS QVONS FAIT

NOUS AVONS COMPARER TOUT LES RESULTATS SONT PROMETEURS

UN PARAGRAPH

PERSPECTIVES

This work presents a comparative study of multiple offloading strategies in an IoT–Fog–Cloud environment, emphasizing balancing the task throughput rate, latency, and energy consumption. The techniques explored range from simple, uniform allocation (Round Robin) and greedy approaches to more advanced, AI-based methods. These include GA-driven and DQRL-driven MLP policies, as well as Transformer-based DQRL solutions, such as NOTE and T-NOTE. IMPROVING

Genetic algorithms (e.g., NSGA2 and NPGA) discover sets of Pareto-optimal solutions, enabling flexible objective trade-offs. Formulated as a weighted single-objective approach, DQRL converges more quickly to a single high-performance policy. Both methods often push tasks to powerful cloud nodes, which reduces failures and latency but increases energy consumption.

NOTE and T-NOTE incorporate attention mechanisms to learn complex dependencies among nodes and task requirements. Though more computationally intensive, these models notably improve success rates, latency, and energy efficiency, especially when task attributes (e.g., deadlines, CPU cycle needs) are explicitly modeled. As parameter counts increase, gradient-based methods are generally preferred to evolutionary ones for training.

Overall Observations. Prioritizing low task throw rate and latency commonly leads to heightened power use on cloud nodes, reflecting how reward or fitness functions can bias resource utilization. Round Robin and other balanced schemes mitigate this bias by diffusing workloads more uniformly, albeit at the cost of increased queuing on weaker nodes. Meanwhile, T-NOTE exhibits the advantage of explicitly incorporating task-level data, enabling more refined offloading decisions.

Future Directions. Potential extensions of this work include:

- **Scenario Generalization:** Assessing the proposed models with diverse network topologies, workload patterns, and real-world constraints to validate robustness and generality.
- **Extended Constraints:** Introducing factors such as node reliability, dynamic pricing, and heterogeneous hardware to further approximate real fog/cloud infrastructures.

- **Novel Algorithms:** Exploring federated reinforcement learning or evolutionary multi-agent approaches for additional benchmarks against the current strategies.

In conclusion, *Transformer-based DQRL* demonstrates the strongest performance under the tested workloads—particularly beneficial for large tasks—whereas multi-objective GA solutions still remain vital for scenarios demanding more nuanced compromises among multiple objectives. Ultimately, the choice of offloading strategy depends on the operational priorities (i.e., minimizing failures and latency vs. distributing workloads for energy savings).

Acknowledgments

The authors acknowledge the use of the ChatGPT AI system (OpenAI) in drafting, editing, and formatting this paper. Specifically, ChatGPT assisted in generating text throughout all sections, improving grammar, clarity, and overall coherence. The authors reviewed and verified all content to ensure accuracy, originality, and relevance.

references