ELEN4000A/ELEN4011A-ELECTRICAL ENGINEERING DESIGN II-2023-FYR: DESIGN OF A MACHINE LEARNING BASED MOBILE EDGE COMPUTING SYSTEM FOR IOT DEVICES.

Bryce Grahn (2138347)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: This investigation project aims to determine an optimal task offloading scheme for energy harvesting devices in mobile edge computing. The limited processing power, bandwidth, and battery capacity of mobile devices make them unable to meet the demands of computationally intensive and delay-sensitive communication tasks. The emergence of Mobile Edge Computing (MEC) provides a potential solution. This paper formulates an optimization problem with the objective of minimizing the average latency by maximizing throughput under an energy constraint, and proposes a deep learning-based task offloading and time allocation algorithm. The optimal solution was solved using brute force and predictions were made faster using a DNN model in tandem with a custom loss function. The simulation results demonstrate that the task throughput of the proposed implementation can reach 98.87% of the optimal solution and achieves an 85.9 % increase over random guessing. In addition the proposed engineering solution minimized tasks dropped, queue build up, and maximised energy harvested.

Key words: Reinforced learning, Mobile Edge Computing (MEC), Task latency, Energy efficiency, Radio frequency harvesting, Machine Type Communicating Devices (MTCD), 5th generation networks (5G)

Submission date: 28th of August 2023

Supervisor: Dr Fambirai Takawira

Field: Telecommunications

1. INTRODUCTION

With the emergence of 5G, and development in wireless communication technologies, many new applications and services have inspired new growth. This is especially true for Machine Type Communicating Devices (MTCD). Consequently, these devices put forward higher requirements for data transmission and processing capacity. While the Central Processing Unit (CPU) enhances the data processing capabilities of MTCD's [1,2], it may at times be insufficient to meet the demands of numerous computation-intensive tasks [4]. Furthermore, the limited battery capacities, computing capabilities, and storage of mobile MTCD's suggest that running many computation and/or storage-intensive applications on these devices may not be feasible. This effectively limits their use in computational-intensive and delay-sensitive communication.

In order to circumvent these challenges, address transmission latency, energy consumption, and ensure better quality of service (QoS) delivery, mobile edge computing or multi-user edge computing (MEC) is proposed. MEC networks are situated closer to the end user which reduces the delay caused by long-distance data transmission. MTCD's can offload the computationally intensive or latency-critical task to the nearby MEC server so as to reduce latency, save energy and extend battery life. In addition, batteries found in the MTCD's have limited capacities which raises the issue of energy constraints.Various energy harvesting techniques have been proposed to address this. Among them, radio frequency energy harvesting (RF EH) stands out as a promising approach where previous research has confirmed an increased user computational performance when utilizing RF EH in tandem with MEC Networks [3,4].

Motivated by the above observations, this paper focuses on the design of an efficient task offloading scheme for MTCDs in mobile edge computing. We formulate an optimization problem involving dynamic offloading and energy harvesting. The goal is to optimize the use of energy whilst maximizing throughput, effectively minimizing latency. In order to solve this optimization problem, we utilize brute force to develop a training dataset and propose a deep learning based task offloading algorithm to make predictions faster.

The main contributions of this paper are elaborated as follows. We discuss a number of methods in the literature, before defining a system model, design specifications, a machine learning approach to optimize and make predictions faster, before finally simulating and discussing our findings.

1.1. Literature review

This project focuses on the design of a task offloading scheme for energy harvesting internet-of-things devices (MTCD) in mobile edge computing. In what follows, we will analyze and summarize related approaches separately.

1.1.1. Binary vs partial offloading

Binary offloading, also known as full offloading, involves transferring the entire workload or computation from one device to another. In other words each MTCD offloads the whole task to an edge server [5]. Partial offloading involves dividing the task between the edge cloud and the local machine or MTCD. Works by [6, 7, 8] have investigated partial offloading in great detail. The authors have considered heterogeneous energy harvesting systems in conjunction with single and multi-edge servers, Multi-hop multi-task schemes, and Mixed Integer Linear Programming (MILP) for partial offloading resource allocation. A study by reference [9] found that, contrary to their initial expectations, the partial off-loading method only exhibits a slight performance advantage over the binary off-loading method in terms of the sum computation rate. This advantage is only exhibited where the end devices are situated at a moderate distance from the base station.

1.1.2. Order of execution

Prioritization and FIFO (First In First Out) are two different approaches used in queue-based systems. The FIFO principle dictates that the first task that enters the queue should be the first one to be processed or executed. Prioritization involves assigning different levels of importance or urgency to tasks in the queue. Each task is given a priority value, and tasks with higher priority are processed before tasks with lower priority. Numerous works by [10, 11, 12] still rely on FIFO for computation, transmission, and edge server queues. This is because it is by far the easiest, fairest and most popular strategy for content replacement [13].

1.1.3. Transmission protocols

Multiplexing is defined as a method of transmitting and receiving multiple independent signals over a single transmission channel. Two of the most common protocols are Time division multiplexing (TDM) and Frequency division Multiplexing. In TDM the transmit side assigns multiple channels in pre-assigned time slots, where multiple time slots constitute a frame. FDM divides the available frequency spectrum into multiple non-overlapping frequency bands. Each data stream is allocated a separate frequency band, and they are transmitted simultaneously over their respective assigned bands. In the case of multi-user systems it is better to use multiple access technologies such as Time division multiple access (TDMA) or Frequency division multiple access. FDMA. These technologies are a subdivision of their parents, FDM and TDM.

The application scenarios predicted for 5G and 6G networks present challenges that the aforementioned technologies can only address in a limited way [14]. However there is a newer more advanced technology Non-orthogonal multiple-access (NOMA) and is predicted to be the future of 5G/6G. NOMA allocates different power levels to each user's signal within the same time-frequency resource. Users with strong channel conditions are allocated higher power, while users with weaker channel conditions receive lower power. This allows NOMA to serve multiple users simultaneously, even if their data rates and channel qualities vary significantly. It can efficiently handle a massive number of IoT devices and support diverse applications with varying communication needs.The combination of RF EH and NOMA techniques in MEC networks is proposed in some prior works to improve the system performance [11]. It is shown that the deployment of NOMA can efficiently reduce the latency and energy consumption of MEC offloading compared to their conventional orthogonal multiple access approaches [12].

1.1.4. System model

A paper by Li et al. (2022) aims to minimize the total energy consumption subject to the service latency requirement by optimizing the task offloading ratio and resource distribution. This involves the time switching (TS) factor, uplink transmission power of MTCDs, downlink transmission power of the base station, and computation resources of MTCDs and MEC server. They propose a (JTORAEH) algorithm that can achieve better performance in terms of the total energy consumption [15].

Zhang et al. (2022) devotes their paper to developing an efficient task offloading and time allocation scheme based on a dedicated Wireless Power Transfer (WPT) transmitter and Mobile Edge Computing (MEC) for MTCD’s. They define a complex optimization challenge aimed at maximizing computational throughput, and propose a deep learning-based task offloading and time allocation algorithm, named DNN-TOTA. This method employs a Deep Neural Network (DNN) in combination with the Order-Preserving Quantization (OPQ) approach to generate the candidate offloading decision sets. The simulation results demonstrate that the throughput of their proposed DNN-TOTA can reach 98% of the optimal solution [12].

Truong and Ha. (2020) address the issue of accommodating increased users. An energy-constrained user harvests the RF energy from the power station and offloads its tasks to multiple access points via non-orthogonal multiple access (NOMA) and non-access point selection (NAPS). Their proposed protocol NOMA NAPS is shown to outperform traditional orthogonal implementations [11] .

A recent paper by Tang et al. (2022) proposes a model-free deep reinforcement learning-based distributed algorithm, where each device can determine its offloading decision without knowing the task models and offloading decision of other devices. They integrate techniques such as long short-term memory (LSTM), dueling deep Q-network (DQN), and double-DQN to improve the algorithm's ability to estimate long-term costs. Their simulation results reveal that the proposed algorithm effectively harnesses the processing capabilities of each MEC server, and significantly reduces the ratio of dropped tasks and average delay when compared with several existing algorithms [10].

1.1.5. Optimizing the objective function

To achieve latency minimization under energy constraints while guaranteeing QoS, a large number of studies have been carried out where the solutions are grouped into three main groups.

1. Mathematical Solution: Authors have formulated a user computation offloading problem as a Mixed Integer Linear Process (MILP) and a Mixed Integer Nonlinear Programming (MINLP). Both are optimized by a Successive Convex Approximation (SCA). Another popular mathematical solution is the Lyapunov optimization method. 2. Heuristic

Algorithms: Algorithms with a non-trivial competition ratio, along with a variety of greedy algorithms have both been proven to achieve near-optimal performance. 3. MDP and Reinforced learning (RL): Task latency has proven to be minimized by Deep Reinforcement Learning (DRL) in Wang et al. (2018) where a new neural network is proposed for state representation to capture the characteristics of Deep Neural Networks (DNN) [10]. Zhu et al. (2021) proposal models the systems as a multi-agent DRL problem that selects suitable edge servers [15].

Mathematical solutions are NP-hard and heuristic tend to have slow convergence speeds and may become trapped in locally optimal solutions during the solution process. Hence, the reinforcement learning method is leveraged to address these limitations within a complex edge environment.The decision-making capability of reinforcement learning not only accelerates the convergence speed but also enhances the overall solution quality. However, reinforcement learning is both complex and cannot deal with high dimension and continuity problems. We thus combine a heuristic greedy algorithm to generate an optimal training dataset and Deep Learning in combination with a neural network to solve both issues.

1.2. Success criteria

The success of this project is separated into several sections. Firstly, the successful implementation of a stochastic optimization problem aimed at minimizing average latency by maximizing throughput under energy constraints. Secondly, introducing a deep learning-based algorithm for efficient task offloading and time allocation. The proposed solution must display increased performance over non-offloading or random strategies. Success is confirmed through successful modeling and results that display the approach's effectiveness in latency reduction and energy consumption. Finally, all system parameters, objectives, and system models should be carefully justified, results should be critically analysed and the engineering solution provided should also take into consideration matters such as sustainability and the solution’s impact on the environment as well as social and economic impacts.

2. SYSTEM MODEL

Consider a set of edge servers i = (1,2, …, I) and a set of MTCD’s j = (1, 2, ..., J) in an MEC system. For simplicity let J = 1, and I = 3. Each MTCD has a number of tasks to be computed and can offload a subset η, (0< η<1), of said tasks to an MEC server. We focus on one frame or period of duration T. Each frame consists of a number of tasks. We develop this model from previous works found within references [14, 19, 20, 21, 22, 24]

2.1 Assumptions and constraints

- Tasks are offloaded to the closest available MEC server
- Each task consists of one packet
- Each frame consists of multiple tasks
- Excess energy can be carried over to the next frame
- Reducing latency has a far greater impact on a users Quality of Service (QOS) than reducing the energy consumed by the MTCD.
- The MTCDs transmit power varies based on the NOMA protocol
- Each frame has a fixed duration of 10ms
- Assumes a 5G transmission environment with appropriate cpu’s, servers, etc.
- Processing, harvesting, computing, transmitting, and receiving bits happens at random intervals throughout the duration of the frame. For simplicity these are grouped together and modeled as a single block and duration rather than small blocks repeated throughout the frame.
- Assume all design specifications, system state parameters, and system action parameters remain constant for a given frame f with duration T but vary from frame to frame.

2.2 MTCD model

Our focus lies on computational tasks for mobile devices or MTCD’s. Each task is non-divisible such that it can only be processed locally or be offloaded to an edge server for processing. We adapt the approach found within reference [9] so that each frame consists of a number of tasks where a subset is partially offloaded or computed locally (Partial offloading). This approach allows for a more balanced utilization of resources that lead to performance gains without entirely relieving the main processor from its tasks. This is confirmed in references [6, 7, 8] where increased performance was achieved.

Task arrivals to an MTCD can be seen as random events that happen independently of each other. Each new task arrives with a certain probability and assuming the average rate (tasks per time period) is constant and two arrivals cannot occur at the same time we model the arrival of tasks as a poisson process with arrival rate $\lambda_i(t)$. Let β and δ represent the task size and its result size in bytes. There are no task dependencies or task priorities as FIFO follows only a first in first out principle. When a new task arrives, the MTCD must decide whether to process it locally, offload it to an MEC server, or drop it to be assigned later.

If an MTCD device i ∈ I has a newly arrived task k_n the scheduler needs to make the offloading decision for task k_n as follows: First let the variable β_n denote whether the task should be offloaded, computed locally, or dropped. β_n is assigned 1 if offloaded, 0 if computed locally, and -1 if the task is to be dropped. Second, the scheduler assigns task k_n to the queue that maximizes the objective function. This decision is made by the DNN model in section 4. The queueing dynamics can be seen in figure 1:

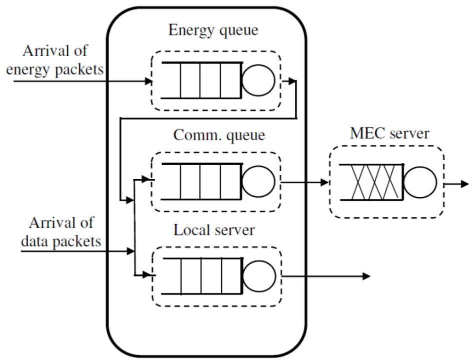


Figure 1: Queueing dynamics

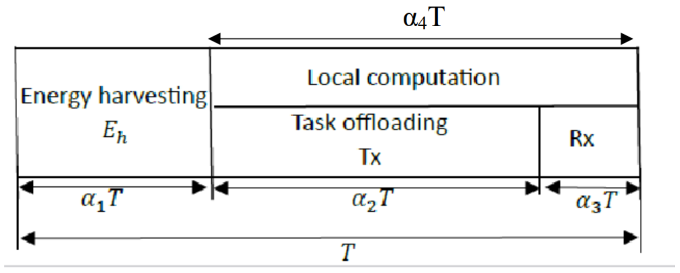


Figure 2: Frame partitioning

Incoming tasks that mathematically won't be processed before the end of the frame are dropped. This ensures the rate entering either queue equals the rate at which they are processed or emptied for the given frame. Furthermore, this strategy prevents the ineffectual allocation of tasks to inappropriate queues as well as task build up. It instead drops the task and provides the tasks with the opportunity to be allocated efficiently in the next frame, thereby enhancing the overall efficiency.

2.2.1. Frame partitioning

By focusing on a single repetitive frame, f ∈ Frames, we streamline the problem's complexity, similar to references [10, 11, 15]. Assuming the MTCD has only one antenna, time division duplexing will be used to split up the frame into times for harvesting energy, transmitting, local computing, and receiving. Let T represent the frame duration in seconds. Processing, harvesting, computing, transmitting, and receiving bits happens at random throughout the frame and is managed by an interrupt handler. For simplicity each category is grouped together and modeled as a single block and duration as seen in figure 2.

The total time T for each frame is split up as follows: The period $T\alpha_i^1(t)$ is used to obtain the necessary energy for computing or offloading of tasks. Period $T\alpha_i^2(t)$ is used for transmitting to the edge server. Period $T\alpha_i^3(t)$ is used for receiving results from the edge server. And finally period $T\alpha_i^4(t)$ is used for local processing.
In order to optimize the utilization of the entire frame, it's essential for either the offloading scheme $\alpha_i^1(t) + \alpha_i^2(t) + \alpha_i^3(t)$ or the local computation scheme $(\alpha_i^1(t) + \alpha_i^4(t))$ equals 1.
The rationale behind the use of 'or' is that, depending on the system state, a higher number of tasks might be allocated to either the local or offloading queue. This approach avoids assigning additional tasks to the local or transmission queue to achieve the necessary sum of 1, especially if doing so would be inefficient.

$$\therefore Max((\alpha_i^1(t) + \alpha_i^2(t) + \alpha_i^3(t)), (\alpha_i^1(t) + \alpha_i^4(t))) \le 1 \tag{1}$$

CD. If $\alpha_i^4(t)$ is the portion of frame T that is allocated for local computation then the energy consumed locally for MTCD i is:

$$E_i^L(t) = K_E(f_i^L) \times \alpha_i^4(t) \times T \tag{2}$$

Let $\vartheta_i^L(t)$ denote the number of tasks remaining from the preceding frame, and $\eta_i(t)$ denote the subset of incoming tasks $\lambda_i(t)$ assigned to the computation queue. If ζ is the number of cycles required to compute one bit then we calculate the time required to process both the newly assigned tasks and the pre-existing tasks within the computation queue, $T_i^L(t)$, as follows:

$$T_i^L = T_{L,new} + T_{L,existing} = (\eta_i(t) \times \lambda_i(t) \times \beta \times 8)_{bits} \times \frac{\zeta}{f_i^L} + (\vartheta_i^L(t) \times \beta \times 8)_{bits} \times \frac{\zeta}{f_i^L}$$

$$\therefore T_i^L = \alpha_i^4(t)T = [(\eta_i(t) \times \lambda_i(t) + \vartheta_i^L(t)) \times \beta \times 8]_{bits\ computed} \times \frac{\zeta}{f_i^L} \tag{3}$$

2.2.3. Transmission queue

Similarly the transmission queue follows the first in first out principle and has a buffer size $B^{offload}$. The arrivals are the tasks to be offloaded to the edge servers. Let $q_x \in Q^{Offloading}$ be one item in the offloading queue. Let $h_i(t)$ denote the channel gain from MTCD i to edge server j. Let $P_i^T(t)$ denote the transmission power allocated to MTCD i in accordance with the NOMA protocol. The transmission rate from MTCD i to edge node j, denoted by R_i^{UL} (in bits per second), is computed as follows:

$$R_i^{UL} = Blog_2(1 + P_i^T(t)|h_i(t)|^2/N_o) \tag{4}$$

where B denotes the bandwidth allocated non-orthogonally to MTCD i, and N_o denotes the received noise power at edge server j. The value of R_i^{UL} is assumed to be a constant for frame f. N_o is both a function of temperature and bandwidth represented by the following equation.

$$N_o = K_N \times Temp \times B \tag{5}$$

The wireless transmission from the MTCD to an edge server suffers from path loss and small-scale fading which affects the channel gain [16, 17]. The channel gain is a complex number whose magnitude is the attenuation and whose angle is the phase shift of the signal. This is modelled as a complex exponential random variable:

$$|h_i(t)|^2 = Path\,loss \times Fading \times Shadow \tag{6}$$

If $\alpha_i^2(t)$ is the portion of frame T that allocated to transmission (offloading of tasks) then the energy consumed whilst offloading for MTCD i is represented by the following equation:

$$E_i^O = P_i^T(t) \times \alpha_i^2(t) \times T \tag{7}$$

Let τ^o be the time it takes to fully process task q_x from allocation to receiving a result:

$$\tau^o = 2t^p + W_{q-transmission} + t^o + W_{q-remote} + t^{ES} + t^{dl} \tag{8}$$

Where the propagation delay $2t^p = 2D/c$, t^o is the wireless transmission delay when offloading, $W_{q-remote}$ and $W_{q-transmission}$ represent the wait time for the edge server queue and offload queue respectively, t^{ES} represents the time required for processing task q_x at edge server j, and lastly t^{dl} represents the wireless transmission delay when receiving. However, since our frame encompasses multiple tasks, we partition the aforementioned equation into two segments: one for transmission (Eq 9) and the other for receiving multiple tasks (Eq 10). The resulting formulation is presented below:

Consider T_i^O as the time span necessary for processing newly assigned tasks, existing tasks within the offload queue, and pre-existing tasks in the MEC queue. Let t^p represent the propagation delay, and f_j^{MEC} be the edge servers CPU cycling frequency. Additionally, let $\vartheta_i^{O}(t)$, $\vartheta_i^{MEC}(t)$ denote the tasks currently in the offload and MEC queue respectively, and $\gamma_i(t)$ represent the subset of incoming tasks $\lambda_i(t)$ assigned to the offload queue.

$$\therefore T_i^{O} = T_{O,new} + T_{O,existing} + T_{O,MEC} = (\gamma_i(t)\lambda_i(t)\beta \times 8)_{bits}/R_i^{UL} + t^p + (\gamma_i(t)\lambda_i(t)(t)\beta \times 8)_{bits} \times \frac{\zeta}{f^{MEC}} + (\vartheta_i^{O}(t)\beta \times 8)_{bits}/R_i^{UL} + t^p + (\vartheta_i^{O}(t)\beta \times 8)_{bits} \times \frac{\zeta}{f^{MEC}} + (\vartheta_i^{MEC}(t) \times \beta \times 8)_{bits} \times \frac{\zeta}{f^{MEC}}$$

In practical scenarios, multiple high-performance MEC servers are present [15, 18]. In addition, computations occur concurrently in the MEC while simultaneous computation and transmission take place at the MTCD. Consequently, we make the assumption that both the propagation delay and the MEC computations are of negligible significance and can be disregarded. This is supported in the following references [11, 15]. The equation thus simplifies too.

$$\therefore T_i^{O} = \alpha_i^{2}(t)T = [(\gamma_i(t)\lambda_i(t) + \vartheta_i^{O}(t))(\beta \times 8)]_{bits\ offloaded}/Blog_2(1 + \frac{P_i^{T}(t)|h_i(t)|^2}{K_N TempB}) \tag{9}$$

The downlink rate R_i^{DL} is similar to equation 4 with $P_i^{T}(t) = P_j^{BS}$. If $\alpha_i^{3}(t)$ is the portion of frame T that allocated to receiving tasks, and δ is the size of the result for task q_x in bytes, we determine the duration required to receive all transmitted tasks in frame f as follows:

$$\therefore \alpha_i^{3}(t) \times T = (\gamma_i(t)\lambda_i(t) + \vartheta_i^{O})(\delta \times 8)/Blog_2(1 + \frac{P_j^{BS}|h_i(t)|^2}{K_N TempB}) \tag{10}$$

See appendix C and D for a detailed derivation.

2.2.4. Energy queue

We introduce an energy queue to prevent the wastage of surplus harvested energy. Instead, this excess energy is used to recharge the battery and supplement future frames in periods of diminished energy harvesting efficiency. This strategy provides our model with the ability to process a constant continuous number of tasks each frame despite the often volatile and unpredictable nature of the system's state parameters. Constraint 3 from section 3.1 ensures that if all tasks are processed before the conclusion of frame f, any remaining time is dedicated to energy harvesting for charging the energy queue $E_i^{Q}(t)$. Given that $\alpha_i^{1}(t)$ designates the fraction of frame T allocated to harvested energy, the energy harvested, denoted by $E_i^{h}(t)$ is as follows (Refer to Appendix B for notations):

$$\therefore E_i^{h}(t) = \alpha_i^{1}(t)\ T I_i^{h}(t) P_j^{BS} (\frac{c}{4\pi d_i(t) f_j^{c}})^{\mu} |h_i(t)|^2 \tag{11}$$

Consequently, the energy available for tasks in frame f and MTCD i is the sum of $E_i^{h}(t)$ and $E_i^{Q}(t)$.

2.3. MEC server model

Each edge server j maintains a single queue shared by all MTCD's. We assume that after an offloaded task is received by an MTCD, the task will be positioned in the queue after the last existing task. Let f_j^{MEC} (in CPU cycles per second) denote the processing capacity of the CPU of edge server j. Let $\vartheta_i^{MEC}(t)$ the number of tasks currently in the MEC queue. If ζ is the number of cycles required to compute one bit then we calculate the time required to process the tasks within the MEC queue, T_j^{MEC}, as follows:

$$T_j^{MEC} = (\vartheta_i^{MEC}(t) \times \beta \times 8)_{bits} \times \frac{\zeta}{f_j^{MEC}} \tag{12}$$

Similarly the MEC queue follows the first in first out principle and has a buffer size B^{MEC}. The arrivals are the tasks to be processed by edge server j. Let $q_x \in Q^{MEC}$ be one item in the MEC queue. Tasks are discarded if they cannot be accommodated within the MEC queue ($x \leq B^{MEC}$).

In order to efficiently and effectively handle a substantial user base, the selection of NOMA has been grounded in reasons outlined in the literature review. NOMA allocates different power levels to each user's signal within the same time-frequency resource. Conceptually, NOMA can be modelled as FDM with varying transmission power. Let B and $P_i^T(t)$ be the bandwidth and power allocated to MTCD i by the NOMA scheme.

3. SYSTEM OBJECTIVE AND SPECIFICATIONS

The system model aims to minimize the average task latency by maximising the system's sum computation rate or throughput under an energy constraint. This is supported in the following articles [9, 12]. We maximise the number of bits processed for a given frame by increasing $\alpha_i^2(t)$ or $\alpha_i^4(t)$. However, this won't result in the lowest latency if the tasks haven't been allocated to the correct queue. Hence it is necessary to determine the frame partitioning as well as the portion of incoming tasks computed locally, offloaded, or dropped to maximize our objective. The computation rate or throughput is represented by the following equation:

$$\{Bits_{computed} + Bits_{offloaded}\}$$

However we require some term that ensures the incoming tasks have been allocated to the correct queues. We do this by minimizing tasks dropped as bits dropped, $Bits_{dropped}$. Tasks dropped are incoming tasks that are not allocated to a queue because they mathematically are unable to be computed in the current frame. To prevent inefficient allocation these tasks are deliberately withheld and rescheduled for processing in the subsequent frame. Therefore our main objective becomes :

$$\max\ \{Bits_{computed} + Bit_{offloaded} - Bits_{dropped}\}$$

3.1. Objective function

Let S = $\{\vartheta_i^L(t), \vartheta_i^O(t), \vartheta_i^{MEC}(t), E_i^Q(t), h_i(t), \lambda_i(t), P_i^T(t), d_i(t)\}$ represent the observable state at the beginning of frame f. Based on the state the model will determine a near optimal action A = $\{\alpha_i^1(t), \alpha_i^2(t), \alpha_i^3(t), \alpha_i^4(t), \theta_i(t), \eta_i(t), \gamma_i(t)\}$. An optimal action is derived from maximising the objective function represented by equation 13 below (Refer to appendix C for derivation):

$$\max_{(\alpha_i^2, \alpha_i^4(t), f_i^L, P_i^T(t), h_i(t), \vartheta_i^L(t), \vartheta_i^O(t), \lambda_i(t))} \{2\alpha_i^4(t)Tf_i^L/\zeta + 2\alpha_i^2(t)TBlog_2(1 + \frac{P_i^T(t)|h_i(t)|^2}{K_N TempB}) - (8 \times \beta)(\vartheta_i^L(t) + \vartheta_i^O(t) + \lambda_i(t))\} \quad \forall\, i \in I \tag{13}$$

With constraints (Refer to appendix D for derivation and justification):

$Q_1(S,A) \geq 0$

$$Q_1(S,A) = \alpha_i^1(t)TI_i^h(t)P_j^{BS}\left(\frac{c}{4\pi d_i(t)f_j^c}\right)^{\mu_i(t)}|h_i(t)|^2 + E_i^Q(t) - K_E(f_i^L)^3\alpha_i^4(t)T - P_i^T(t)\alpha_i^2(t)T \quad \forall\, i \in I; j \in I$$

$Q_2(S,A) \leq tolerance$

$$Q_2(S,A) = \alpha_i^3(t) - \frac{\alpha_i^2(t)\delta log_2(1+P_i^T(t)|h_i(t)|^2/(K_N TempB))}{\beta log_2(1+P_j^{BS}|h_i(t)|^2/(K_N TempB))} \quad \forall\, i \in I$$

$Q_3(S,A) = 1$

$$Q_3(S,A) = Max((\alpha_i^1(t) + \alpha_i^2(t) + \alpha_i^3(t)), (\alpha_i^1(t) + \alpha_i^4(t))) \quad \forall\, i \in I$$

$0 \leq Q_4(S,A) \leq 1$

$$Q_4(S,A) = \alpha$$

$0 \leq Q_5(S,A) \leq 1$

$$Q_5(S,A) = (\alpha_i^4(t)T\frac{f_i^L}{\zeta\beta \times 8} - \vartheta_i^L(t))/\lambda_i(t) \quad \forall\, i \in I$$

$0 \leq Q_6(S,A) \leq 1$

$$Q_6(S,A) = (\frac{\alpha_i^2(t)T}{\beta \times 8}Blog_2(1 + P_i^T(t)|h_i(t)|^2/(K_N TempB)) - \vartheta_i^O(t))/\lambda_i(t) \quad \forall\, i \in I$$

3.2. Determining the system parameters

In this section we will discuss the decision process behind the system parameters. Refer to appendix B for notations.

The channel gain encompasses the effects of path loss, shadowing, and fading. It is expressed as a complex number, where its magnitude signifies the signal attenuation, and its angle represents the phase shift of the signal at a specific time instant [5]. Many systems employ complex techniques to achieve a favorable channel gain, $h_i(t) \simeq 1$ with the aim of minimizing signal loss [3]. Similarly we aim to reach a favourable channel gain. As such our approach involves modeling $h_i(t)$ using an exponential average of 0.7. The variable $\mu_i(t)$ represents the channel path loss exponent. When substituted in equation $(\frac{c}{4\pi df})^{\mu}$, it describes how the signal strength diminishes as it travels through space. In free-space environments with minimal obstacles, the value of u is close to 2. Generally, $\mu_i(t)$ falls within the range of 0 to 6. An average of 3 is adopted in our analysis to best encapsulate diverse scenarios. This approach is consistently applied to other variables. The distance between the end user and the base station, represented as $d_i(t)$, typically spans from 10 to 150 meters within 5G contexts [19]. For our assessment, we consider an average distance of 80 meters. The maximum packet size permitted by 5G standards is 1500 bytes [20]. Most telecommunications technologies strive to closely approach this 1500-byte threshold, as it yields optimal performance. We, thus, select 1500 bytes as our designated task size. In 5G networks, the uplink and downlink speeds range between 150 Mbps and 250 Mbps [21]. To simulate these speeds within a 5G environment, we define the bandwidth, B, as 5×10^6 Hz. Energy efficiency ratios, $I_i^h(t)$, ranging from 0.5 to 0.9 have been documented in existing literature [11, 15]. We opt for 0.7 to comprehensively represent various scenarios. The standard power value for most IoT devices, $P_i^T(t)$ is set at 0.5W.

We carefully select parameters f_j^c, $\lambda_i(t)$, f_i^L, P_j^{BS}, $B^{offload}$, B^{Comp}, to have both sufficient energy and time to compute all the tasks with arrival rate $\lambda_i(t)$ before the end of frame f. For radio broadcast, the AM and FM frequency bands span from 540 kHz to 1700 kHz and 88 MHz to 108 MHz respectively. Base stations typically emit power ranging from 60 to 120W [5, 9]. Our objective is maximising throughout which consequently requires a larger portion of frame period T dedicated to processing tasks. As such we design our system such that approximately 70% of the frame is used for local computation and offloading. The remaining 30 % dedicated to energy harvesting must contain enough energy to supply that 70 %. From cumbersome substituting of possible variations we conclude that having a dedicated channel frequency, f_j^c, for power transmission yields many orders of magnitude more energy for the MTCD. By setting f_j^c to 1 MHz, we avert the risk of energy harvesting impeding task processing, as proven by the calculations below. For our experimental setup, we opt for P_j^{BS} = 100W. Additionally, in terms of task management, let's denote $\lambda_i(t) + \omega$ as the sum of arriving tasks and the average pre-existing tasks in the queue, resulting in $\lambda_i(t) + \omega = 90 + 5 = 95$ tasks. The chosen f_i^L is 0.65 GHz. We size the buffer to accommodate the edge case of complete local computation or full offloading, in addition to accounting for the accumulation of an entire extra queue of tasks. This results in a buffer size of 180 tasks. Our model is intentionally crafted to prevent task accumulation, a characteristic reinforced by the observations depicted in figures 5 and 6 of section 5. As such a buffer size of 180 tasks is reasonable.

We prove the system's ability to effectively match and exceed the task arrival rate $\lambda_i(t)$ through the utilization of the following equations:

The energy harvested for a duration equal to 30% of frame duration T using Eq 11 is:

$$TI_i^h(t)P_j^{BS}(\frac{c}{4\pi d_i(t)f_j^c})^{\mu}|h_i(t)|^2 = 0.3 \times 0.01 \times 0.7 \times 100 \times (\frac{3\times10^8}{4\pi\times80\times10^6})^3 = 0.0055806\text{J}$$

The energy consumed whilst processing tasks for a duration equal to 70% of T using Eq 2,7 is

$$0.7 \times (K_E(f_i^L)T + P_i^T(t)T) = 0.7(10^{-27} \times (0.65 \times 10^9)^3 \times 0.01 + 0.5 \times 0.01$$
$$= 0.005422 \text{ J}$$

Thus we meet our objective of a 70-30 % split. 0.005422 J < 0.0055806 J.

We select parameters such that the arrivals in the computation or transmission queues are of the same order as departures (in packets/s). Partial offloading is implemented so we want to ensure that the computation rate is sufficient enough to compute tasks of the order of half the incoming rate (The other half will go to the transmission queue). Remember that we defined the other parameters so that 0.7 of the frame could be used for offloading or computing locally. Let μ_{local} and $\mu_{offload}$ be the computation rate (tasks per frame) for local computation and offloading respectfully.

$$\mu_{local} = 0.7 \times T\frac{f_i^L}{\zeta\beta\times 8} = 0.7 \times 0.01 \times \frac{0.65\times10^9}{10\times1500\times8} = 38 \text{ tasks per frame.}$$

$$\mu_{offload} = 0.7 \times \frac{T}{\beta\times8}Blog_2(1 + P_i^T(t)|h_i(t)|^2/(K_N TempB))$$

$$= 0.7 \times \frac{0.01}{1500\times8} \times 5 \times 10^6 \times log_2(1 + 0.5 \times 0.7/(1.38 \times 10^{-23} \times 298.15 \times 5 \times 10^6))$$

$$= 128\ tasks\ per\ frame$$

Combining both yields an average of 166 tasks computed per frame. Selecting a task arrival rate of 90 is thus of suitable order that can be processed before the end of frame f.

4. SUPERVISED LEARNING DNN SOLUTION

4.1 Solving the objective function

Papers such as [20, 21, 24] use Lagrange, gradient descent, and Bisection method, etc to solve the optimization problem. However our objective function is linear

$$\max_{(\alpha_i^2, \alpha_i^4(t), f_i^L, P_i^T(t), h_i(t), \vartheta_i^L(t), \vartheta_i^O(t), \lambda_i(t))} \{2\alpha_i^4(t)Tf_i^L/\zeta + 2\alpha_i^2(t)TBlog_2(1 + \frac{P_i^T(t)|h_i(t)|^2}{K_N TempB}) - (8 \times \beta)(\vartheta_i^L(t) + \vartheta_i^O(t) + \lambda_i(t))\} \quad \forall i \in I$$

and maximising this equation will always result in $\alpha_2 = \alpha_4 = 1$ as maximising the duration maximises the bits processed. We can't have $\alpha_2 = \alpha_4 = 1$ as there is no time for harvesting energy and thus we cannot process anything. The specified constraints $Q_1(S, A)$ to $Q_6(S, A)$ solve this issue.

Our problem thus becomes maximising a linear objective function constrained by 6 inequalities. Solving this mathematically is both complex and at 56 % of the time does not converge to the optimal solution. Instead, we adopt a brute-force approach to attain results within a predefined level of precision. Initial testing revealed that employing brute-force techniques yielded an accuracy of 99% of the optimal solution. In addition, this was accomplished in a third of the time while ensuring 100% reliability. The high level algorithm is (For a comprehensive implementation guide, please refer to Appendix E for the corresponding MATLAB code):

Algorithm 1: Solving for an optimal action based on a given system state using brute force.

```
1. Define a number of random guesses G
2. For guess g ϵ G:
3. Randomly guess $\alpha_i^1(t), \alpha_i^2(t), \alpha_i^3(t), \alpha_i^4(t)$ within a predefined range
    4. If guess meets constraint $Q_1(S, A), Q_2(S, A), Q_3(S, A), Q_4(S, A), Q_5(S, A), Q_6(S, A)$ and $O(\alpha_i^2(t), \alpha_i^4(t))$ is a new maximum
        5. Let $\alpha_i^1(t), \alpha_i^2(t), \alpha_i^3(t), \alpha_i^4(t)$ be the new optimal solution with objective $O(\alpha_i^2(t), \alpha_i^4(t))$
6. Return $(\alpha_i^1(t), \alpha_i^2(t), \alpha_i^3(t), \alpha_i^4(t), O(\alpha_i^2(t), \alpha_i^4(t)))$
```

We employ MATLAB to execute both Algorithm 1 and Algorithm 2. Matlab is indubitably faster than Python especially at sequential execution [22]. This was further substantiated by the considerable improvement observed when employing MATLAB which resulted in a drastic improvement from days to hours as compared to running the algorithm in Python.

4.2. Machine learning model

4.2.1. Training dataset

In section 4.1 we developed a greedy algorithm that solves for the optimal action for a given environment state to some predefined degree of certainty. This action maximises the objective function. Because this greedy solution is both computationally and time intensive we require a model to make predictions faster. Thus, we train a deep neural network (DNN) model on a generated dataset of optimal actions. This approach transitions the problem into a supervised learning framework, combined with Deep neural network. The algorithm is as follows (For a comprehensive implementation guide, please refer to Appendix E for the corresponding MATLAB code):

Algorithm 2: Solving the objective function for a given system state and generate a training dataset

```
7.  Define system design parameters ($T$, $Temp_i(t)$, $I_i^h(t)$, $\mu_i(t)$, $f_j^c$, $B$, $P_j^{BS}$, $f_i^L$, $f_j^{MEC}$, $N_i^o(t)$, $\zeta$, $\beta$, $\delta$, $K_N$, $C$, $K_E$)
8.  Generate system state parameters and their distributions ($\vartheta_i^L(t)$, $\vartheta_i^O(t)$, $\vartheta_i^{MEC}(t)$, $h_i(t)$, $E_i^Q(t)$, $P_i^T(t)$, $d_i(t)$), [$rayleigh, rayleigh, rayleigh, rayleigh, Exponential\ random\ variable, poisson, randint, rayleigh$]
9.  For system state $s \in S$:
    10. Define and simplify the objective function $O(\alpha_i^2(t), \alpha_i^4(t))$
    11. Define and simplify the constraint functions $Q_1(S,A)$, $Q_2(S,A)$, $Q_3(S,A)$, $Q_4(S,A)$, $Q_5(S,A)$, $Q_6(S,A)$,
    12. Solve the optimization problem using brute force with $n$ random guesses. Return ($\alpha_i^1(t), \alpha_i^2(t), \alpha_i^3(t), \alpha_i^4(t), O(\alpha_i^2(t), \alpha_i^4(t))$)
    13. Solve for $\theta_i(t)$, $\gamma_i(t)$, $\eta_i(t)$
    14. Determine $E_i^R(t)$, $N_0$ tasks computed, $N_0$ tasks dropped.
15. Export the training dataset:
```

4.2.2. DNN model

The objective of the neural network is to find a mapping from each environment state to a set of actions. Figure 3 shows an illustration of the neural network of MTCD i. Specifically, the environment state information is passed to the neural network through an input layer. Then, the mapping from all the environment states are learned through five fully-connected (FC) layers. Meanwhile, batch normalization is used in neural network layers to stabilize and accelerate training. Lastly a sigmoid activation layer is used to compress output values within the range of 0 to 1. (For a comprehensive implementation guide, please refer to Appendix H for the corresponding python code).

The input layer is responsible for taking the environment state as an input and passing them to the following layers [24]. For MTCD device i, the environment state information: $\{\vartheta_i^L(t), \vartheta_i^O(t), \vartheta_i^{MEC}(t), E_i^Q(t), h_i(t), \lambda_i(t), P_i^T(t), d_i(t)\}$ will be passed to the FC layer. The five FC layers are responsible for learning the mapping from the state and the learned load level dynamics to the actions. Each FC layer contains a set of neurons with rectified linear units (ReLU), which are connected with the neuron [11]. Batch normalization is used in neural network layers to stabilize and accelerate training. It normalizes the output of each layer within a mini-batch, reducing internal covariate shift and enabling more efficient and faster convergence during training [23]. This helps in faster learning, better generalization, and improved overall performance of the neural network in the previous and following layers. The sigmoid activation is employed in the output layer of Deep Neural Network (DNN).

By compressing output values within the range of 0 to 1, the sigmoid function facilitates the interpretation of outputs as probabilities. Consequently, guaranteeing constraints $Q_4(S, A)$, $Q_5(S, A)$, $Q_6(S, A)$ are met.

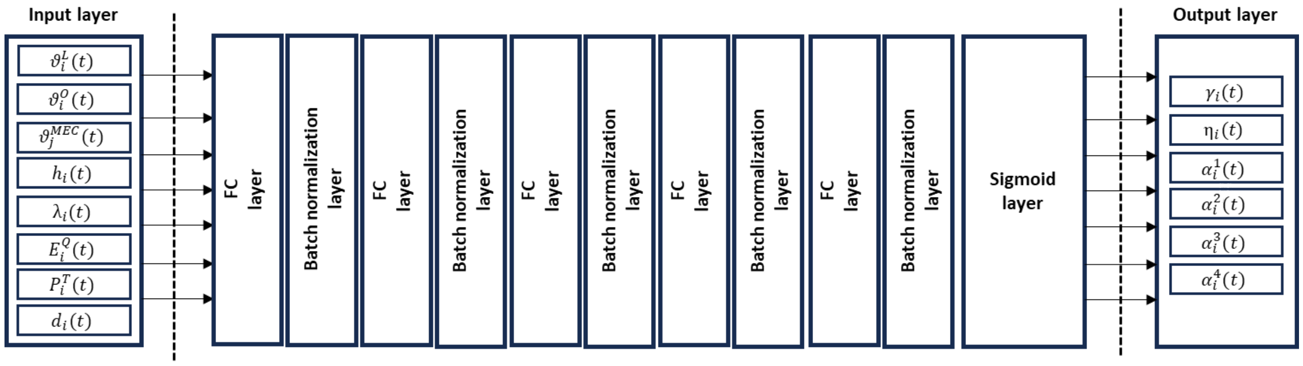


Figure 3: DNN model

A random grid search was employed to optimize the hyperparameters. The resulting architecture consisted of 5 ReLU layers, each with hidden nodes set at 512, 256, 128, 64, and 32, minimizing the loss function. To mitigate overfitting and minimize training-testing loss disparity, we integrate batch normalization and reduce the number of epochs, 65 to be exact. This improved the testing loss from 0.302 to 0.132 (illustrated in Figure 4). However, a notable 32% of predictions were characterized as invalid—falling short of adhering to constraints detailed in equations $Q_{1-6}(S, A)$. This challenge was addressed through the implementation of a customized loss function for DNN training (Algorithm 3). Although the impact on overall loss was marginal, a mere 0.003 improvement, the discernible achievement manifested in the increased number of valid predictions, ascending from 68% to an impressive 99.6%. The custom loss function is as follows (For a comprehensive implementation guide, refer to Appendix G for the corresponding python code):

```
Algorithm 3: Custom loss function ($y_{pred}, y_{true}$)

1. Calculate MSE loss between y_true and y_pred.
2. Compute penalties for constraint violations by summing specific elements of y_pred and applying ReLU to
   positive differences from constraint values
       $0 \le \alpha_i^1(t) + \alpha_i^2(t) + \alpha_i^3(t) \le 1;$
       $0 \le \alpha_i^1(t) + \alpha_i^4(t)) \le 1;$
       $0 \le (\eta_i(t) + \gamma_i(t)) \le 1;$
3. Calculate the total loss as the sum of MSE loss and constraint penalties.
4. Return the total loss as the result.
```

When selecting a suitable code editor for training the DNN model and simulating an environment we opt for Python in conjunction with Google Colab due to its ease of use and rich ecosystem for machine learning tasks [36]. TensorFlow provides efficient computation for building and training complex neural networks, while Keras simplifies the model construction process.

4.2.3. Optimal and random solution

Our success criteria requires a critical evaluation such that our solution performs better than random guessing and successfully converges to an optimal solution. We introduce a greedy method as a benchmark which generates the optimal offloading decision set by enumerating all $(4 \times 6 \times 40 \times 100 \times 200 \times 20 = 384\,000\,000)$ offloading decision sets. We denote the maximum throughput rate of the greedy method as C_{max}. We define the throughput rate ratio as $C_{ratio} = C_{DNN}/C_{max}$ where C_{DNN} is the throughput obtained from our DNN model. Figure 9 illustrates the evolution of C_{ratio} of which the results are discussed in section 5.3. In addition to the benchmark, we establish a definition for a random guess – a decision set that adheres to constraints $Q_{1-6}(S, A)$ but fails to optimize the objective function. To comprehensively assess random guessing, we generate a dataset of valid random guesses and train a new DNN model with the same hyper parameters. Detailed findings are presented in Section 5.

4.3. Environment simulation

We consider a scenario with 2 mobile devices and 1 edge server. The parameter settings are given in Table 1 to 3 of appendix B. In these simulations, we focus on a scenario with a non-stationary environment, i.e.,the system transitions from environment state and action to the next environment state over continuous frames. Each new environment state relies on the action taken in the previous frame. This approach allows us to accurately evaluate the model's performance over time. Especially its ability to recover back to the optimal solution as depicted in figure 11. The outcomes of our simulations are presented in Figures 7 to 11. These subfigures feature the frame on the x-axis and the target on the y-axis. Explanation of the code can become quite cumbersome, for a comprehensive implementation guide, refer to Appendix G for the corresponding python code.

5. RESULTS AND DISCUSSION

5.1. Results for training, optimal, and random

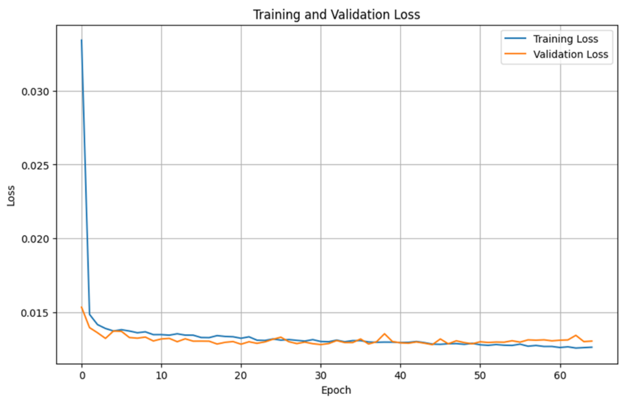
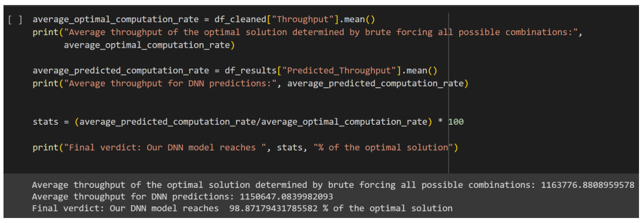


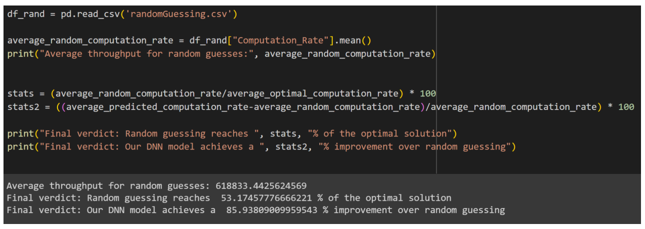
Figure 4: Training and validation loss



```
[ ] average_optimal_computation_rate = df_cleaned["Throughput"].mean()
    print("Average throughput of the optimal solution determined by brute forcing all possible combinations:",
          average_optimal_computation_rate)

    average_predicted_computation_rate = df_results["Predicted_Throughput"].mean()
    print("Average throughput for DNN predictions:", average_predicted_computation_rate)


    stats = (average_predicted_computation_rate/average_optimal_computation_rate) * 100

    print("Final verdict: Our DNN model reaches ", stats, "% of the optimal solution")
```

```
Average throughput of the optimal solution determined by brute forcing all possible combinations: 1163776.8808959578
Average throughput for DNN predictions: 1150647.0839982093
Final verdict: Our DNN model reaches  98.87179431785582 % of the optimal solution
```

Figure 5: Comparing predicted throughput to optimal throughput



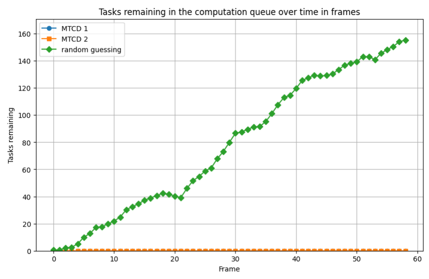
```
df_rand = pd.read_csv('randomGuessing.csv')

average_random_computation_rate = df_rand["Computation_Rate"].mean()
print("Average throughput for random guesses:", average_random_computation_rate)


stats = (average_random_computation_rate/average_optimal_computation_rate) * 100
stats2 = ((average_predicted_computation_rate-average_random_computation_rate)/average_random_computation_rate) * 100

print("Final verdict: Random guessing reaches ", stats, "% of the optimal solution")
print("Final verdict: Our DNN model achieves a ", stats2, "% improvement over random guessing")
```

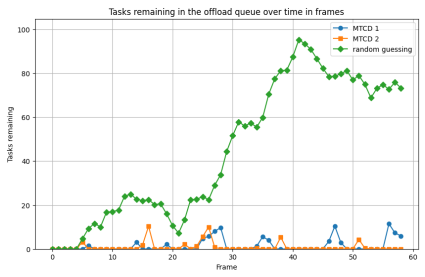
```
Average throughput for random guesses: 618833.4425624569
Final verdict: Random guessing reaches  53.17457776666221 % of the optimal solution
Final verdict: Our DNN model achieves a  85.93809009959543 % improvement over random guessing
```

Figure 6: Comparison with random guesses

In this section, we assess the performance of our proposed MEC system based on training and validation loss, as well as the accuracy of our DNN predictions against both the optimal and random guess datasets as defined in section 4.2.3. The training and validation loss was 0.0127 and 0.0134 respectively (see figure 4). Our model achieves an impressive 98.87% accuracy, as demonstrated in Figure 5, compared to the optimal solution. Additionally, it demonstrates a 85.9% improvement over random guessing, as illustrated in Figure 6. Notably, our final model improved over the previous model's 82% accuracy (where neither a customized loss function or hyperparameter tuning was utilized). A paper by Zhang et al. (2022) in reference [12], achieved a comparable 98% accuracy and a training loss of 0.04 using a similar approach. However, their method relied on DQ learning with a reward function, without integrating a custom loss function. In conclusion, our approach not only meets but surpasses the solutions presented in existing literature.

5.2. Results for the environment simulation





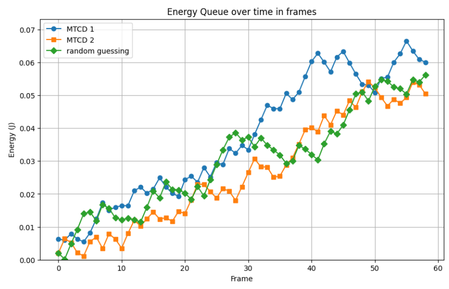


Figure 7: Tasks remaining in the computation queue over time

Figure 8: Tasks remaining in the offload queue over time

Figure 9: Energy queue over time

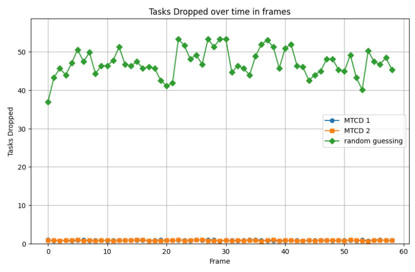


Figure 10: Tasks dropped over time

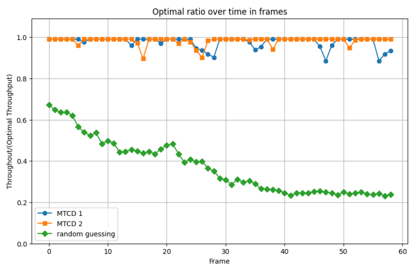


Figure 11: Throughput ratio over time

In this section, we present the performance analysis of the proposed MEC system, focusing on tasks dropped, remaining tasks, energy in the queue, and throughput ratio. Figures 7 and 8 illustrate the remaining tasks in the computation and offload queues at each frame's end. By introducing an objective function parameter that reduces task drops through non-allocation, we intentionally defer certain tasks for processing in subsequent frames to avoid inefficient allocation. This is evident from the absence of task accumulation in Figure 7 and the swift return to no accumulation in Figure 8. In contrast, omitting this parameter leads to task accumulation in the queue, as shown in the random guessing scenario where the objective function is ignored. Figure 9 conveys the deliberate accumulation of energy in the energy queue, a design feature governed by constraints $Q_1(S, A)$ and $Q_3(S, A)$). These constraints ensure that any spare time after task computation within a frame is utilized for energy harvesting. The benefit being that the model clearly has the capacity for an increased task arrival rate for an even greater throughput. The evolution of dropped tasks over time is depicted in Figure 10. We clearly see how incorporating a parameter in the objective function that aims to minimise tasks dropped compared to one that does not (random guessing) performs considerably better. Figure 11 illustrates the throughput ratio defined in section 4.3.2. Impressively, our model achieves an average of 96.1% of the optimal solution and rapidly converges to the optimal state (within 1.04 frames) when environmental conditions deviate from the norm. In summary, our model demonstrates outstanding performance, energy management, adaptability, and robustness. This achievement maximizes throughput while meeting energy constraints and minimizing latency.

5.3. Future improvements

The engineered solution manages to achieve 96.1 to 98.87% of the optimal solution and performs considerably better than random guessing. However, our approach utilizes supervised learning techniques, as such the model performs poorly for environment states that exist outside of the training dataset. We therefore could utilize Deep Q learning which is a model-free deep reinforcement learning technique that enables each MTCD to make its offloading decision without knowing the information (e.g., task models, offloading decisions) of other MTCD devices in addition to adapting to new environment states for greater flexibility. The model also relies heavily on the simplifications made and it would be much better to have less assumptions and a more complex feature space such as multiple edge servers, queues, etc, to better represent a real world scenario. Additionally, instead of using brute force we use the bisection method when generating our training dataset which in theory reduces the overall time taken to solve for the optimal action.

6. ENVIRONMENTAL, SOCIAL, AND ECONOMIC IMPACTS

To better understand the environmental, social, and economic impacts we begin by outlining the standard operation of our engineering solution. This entails successfully and sustainably meeting the demands of computationally intensive and delay-sensitive communication tasks for a substantial user base. Secondly the solution adheres to regulatory privacy norms ensuring various security protocols are met. It is both energy efficient and economically sustainable, and has a good reputation amongst the user base. Finally its design minimizes discrimination, algorithmic bias, disposition, and environmental costs (reducing environmental footprints and conserving resources). In what follows we consider the impacts and risks should the aforementioned criteria fail.

6.1. Environmental

Our solution's inability to meet the computation and latency demands of a growing user base will result in the deployment of additional MEC servers. These servers consume more energy, negatively impacting the environment. . Should reliability prove to be insufficient and given the dependency of the user base on the MEC Server, standby/back up MEC Servers would be required. Inefficient task offloading decisions could lead to unnecessary data transmission between devices and MEC servers, consuming additional energy for data transfer. All hardware has a limited lifespan. Swift tech adoption can escalate electronic waste (e-waste) if old devices are discarded, adding to pollution and depleting resources.

6.2. Social

One major concern associated with MEC revolves around security. When data is processed beyond the usual corporate firewall, it becomes more exposed to potential breaches. Furthermore, edge devices are commonly set up in less-regulated settings, putting them at risk of physical interference or harm. With the widespread deployment of multiple MTCD devices and edge servers, the chances of security breaches, like DDoS attacks, also go up due to the expanded points vulnerable to attacks. It can be challenging to secure each MTCD device, as they are often distributed across a wide geographic area. Centralised processing, increases the effectiveness of such an attack, impacting on a wider audience of users, attracting sophisticated criminal elements. The impact of such attacks would be exponential when compared to decentralised processing impacting communities as opposed to individuals. As more advanced technologies like MEC become prominent, individuals or communities without access to 5G networks or edge computing infrastructure might face a digital divide, leading to unequal access to enhanced services and opportunities increasing social inequalities in developing economies. The MTCD's battery charging depends on RF signals. In rural areas with fewer RF signals, charging is slower or often interrupted. This renders the device ineffective, with life threatening consequences if inaccessible. Additionally, since this technology is relatively new and unfamiliar, a flawed initial implementation could create a negative perception of task offloading using MEC. This could directly or indirectly hinder the progress of this promising technology. Algorithmic bias is another concern. Since the model relies on algorithms to predict the best actions according to a particular environmental state, individuals in locations with environment states not included in the model's training might encounter even more unfavorable results. Hence, having a strong encryption and authentication strategy is crucial, along with a versatile and dependable model that adjusts to all state parameters.

6.3. Economic

Setting up MEC infrastructure and network upgrades demand significant initial investments, potentially deterring smaller businesses from adopting this tech. This situation allows major corporations to dominate the market, akin to AWS, Microsoft Azure, and Google Cloud Platform's dominance in cloud services. Consequently, they can charge high prices for an otherwise affordable technology. Poorly managed offloading might lead to higher data transfer costs, especially if tasks are offloaded to the incorrect server. The presence of RF energy in the atmosphere is highly inconsistent. To address this challenge, one potential solution is to build dedicated RF power transmission towers to enhance coverage in these areas. However, it's worth noting that these RF towers could consume more energy compared to a traditional battery-powered device.

7. SUSTAINABILITY

Any engineering solution designed and developed needs to be sustainable and secure whilst promoting diversity and fairness. Maintainability and skilled technical support is critical to the longevity and sustainability of the MTCD. The implementation of planned preventative maintenance, with documented checks and critical spares holding for quick turnaround during breakdowns to reduce interruptions is essential. Upskilling and sourcing from local communities also enhances social development. Another good way to ensure sustainability is to ensure that the MEC network design follows all safety and environmental regulations provided by the regulator. Ensuring a relatively more ”greener” device by using smaller rechargeable cells with a higher charge cycle count is one of the ways this solution aims to ensure that it promotes a better environment. The use of recycled or recyclable material in the making of MEC networks would also be something that would ensure a much more environmentally friendly and sustainable solution. The incorporation of green energy (solar and wind) to power the MTCD combined a partnership with an entity for recycling of batteries conserving global resources. The inclusion of green energy will attract "green investments" increasing the success of funding and speed of implementation into developing sectors enhancing economic development and social equity. Reducing carbon emissions, and energy usage combined with sourcing products from fair trade organisations and physical waste disposal with a low carbon footprint is essential. On the positive side, the paper emphasizes the reduction of average latency, maximization of throughput, and energy efficiency, all of which are crucial factors for sustainability. The use of energy harvesting techniques, especially radio frequency energy harvesting, aligns well with sustainable practices by reducing the dependence on conventional power sources and potentially extending device lifetimes. The reduction of tasks dropped, queue buildup, and maximized energy harvested are also positive outcomes for a sustainable engineering solution, indicating effective resource utilization.

8. CONCLUSION

This paper investigated an optimal task offloading scheme for energy harvesting devices in mobile edge computing. The purpose is to address the challenges of limited processing power, bandwidth, and battery capacity of MTCD's making them unable to meet the demands of computationally intensive and delay-sensitive communication tasks. This paper formulated an optimization problem with the objective of minimizing the average latency by maximizing throughput under an energy constraint, and proposes a deep learning-based task offloading and time allocation algorithm. The simulation results demonstrate that the computation rate of the proposed implementation can reach 98.87% of the optimal solution and achieve an 85.9% increase over "random guessing". The engineering solution effectively fulfilled the success criteria and outperformed certain existing literature. While guaranteeing system performance, we minimized tasks dropped, queue build up, and maximised energy harvested. Simulation results show that compared with our greedy benchmark, our proposed supervised DNN model can achieve a comparable task throughput. Finally, the numerical results demonstrate the rapid flexibility of our model as it takes an average of 1.04 frames to converge back to the optimal solution.

ACKNOWLEDGEMENTS

The author would like to thank the project supervisor, Dr Fambirai Takawira for their help throughout the length of the project.

REFERENCES

[1] Guo, H., Liu, J., 2018. Collaborative computation offloading for multiaccess edge computing over fiber–wireless networks. IEEE Trans. Veh. Technol. 67 (5), 4514–4526.http://dx.doi.org/10.1109/TVT.2018.279042.

[2] Liu, F., Shu, P., Jin, H., Ding, L., Yu, J., Niu, D., Li, B., 2013. Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications. IEEE Wirel. Commun. 20 (3), 14–22. http://dx.doi.org/10.1109/MWC.2013.6549279

[3] X. Lu, P. Wang, D. Niyato, D. I. Kim, and Z. Han, "Wireless networks with RF energy harvesting: A contemporary survey," IEEE Communications Surveys and Tutorials, 17(2), pp. 757–789, 2014.

[4] F. Wang and X. Zhang, "Dynamic computation offloading and resource allocation over mobile edge computing networks with energy harvesting capability," in IEEE International Conference on Communications (ICC), Kansas City, MO, USA, May 2018.

[5] Feng, C., Han, P., Zhang, X., Yang, B., Liu, Y. and Guo, L. (2022). Computation offloading in mobile edge computing networks: A survey. *Journal of Network and Computer Applications*, 202, p.103366. doi:https://doi.org/10.1016/j.jnca.2022.103366.

[6] Zhang, T., Chen, W., 2021. Computation offloading in heterogeneous mobile edge computing with energy harvesting. IEEE Trans. Green Commun. Netw. 5 (1), 552–565. http://dx.doi.org/10.1109/TGCN.2021.3050414

[7] Li, L., Quek, T.Q.S., Ren, J., Yang, H.H., Chen, Z., Zhang, Y., 2021a. An incentive-aware job offloading control framework for multi-access edge computing. IEEE Trans. Mob. Comput. 20 (1), 63–75. http://dx.doi.org/10.1109/TMC.2019.2941934.

[8] Sahni, Y., Cao, J., Yang, L., Ji, Y., 2021. Multi-hop multi-task partial computation offloading in collaborative edge computing. IEEE Trans. Parallel Distrib. Syst. 32 (5), 1133–1145. http://dx.doi.org/10.1109/TPDS.2020.3042224.

[9] Poposka, M. (2022). Binary vs partial offloading in wireless powered mobile edge computing systems with fairness guarantees. *ITU Journal on Future and Evolving Technologies*, [online] pp.498–507. Available at: https://www.researchgate.net/publication/364142585_Binary_vs_partial_offloading_in_wireless_powered_mobile_edge_computing_systems_with_fairness_guarantees [Accessed 7 Aug. 2023].

[10] Tang, M. and Wong, V.W.S. (2020). Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems. *IEEE Transactions on Mobile Computing*, pp.1–1. doi:https://doi.org/10.1109/tmc.2020.3036871.

[11] Truong, V.-T. (2020). Secured Scheme for RF Energy Harvesting Mobile Edge Computing Networks based on NOMA and Access Point Selection. [online] Available at: https://0-ieeexplore-ieee-org.innopac.wits.ac.za/document/9335833 [Accessed 27 Jul. 2023].

[12] Zhang, R. and Li, H. (2022). Deep Learning-based Task Offloading and Time Allocation for Edge Computing WBANs. [online] Available at: https://0-ieeexplore-ieee-org.innopac.wits.ac.za/document/10001493 [Accessed 27 Jul. 2023].

[13] Safavat, S., Sapavath Naveen, N.N. and Rawat, D.B. (2019) 'Recent advances in mobile edge computing and content caching', *Digital Communications and Networks* [Preprint]. Available at: https://doi.org/10.1016/j.dcan.2019.08.004.

[14] Michailow, N., Matthé, M. and Gaspar, I. (2014). Generalized Frequency Division Multiplexing for 5th Generation Cellular Networks (Invited Paper). [online] pp.1–18. Available at: https://www.researchgate.net/publication/264576550_Generalized_Frequency_Division_Multiplexing_for_5th_Generation_Cellular_Networks_Invited_Paper [Accessed 12 Aug. 2023].

[15] Li, S., Zhang, N., Jiang, R., Zhou, Z., Zheng, F. and Yang, G. (2022). Joint task offloading and resource allocation in mobile edge computing with energy harvesting. *Journal of Cloud Computing*, 11(1). doi:https://doi.org/10.1186/s13677-022-00290-w.

[16] L. Li, T. Q. S. Quek, J. Ren, H. H. Yang, Z. Chen, and Y. Zhang, "An incentive-aware job offloading control framework for multiaccess edge computing," IEEE Trans. Mobile Comput., early access, Sep. 17, 2019, doi: 10.1109/TMC.2019.2941934.

[17] Y. Mao, J. Zhang, S. Song, and K. B. Letaief, "Stochastic joint radio and computational resource management for multi-user mobileedge computing systems," IEEE Trans. Wireless Commun., vol. 16, no. 9, pp. 5994–6009, Sep. 2017.

[18] Gul-E-Laraib *et al.* (2023) 'Content Caching in Mobile Edge Computing Based on User Location and Preferences Using Cosine Similarity and Collaborative Filtering', *Electronics*, 12(2), p. 284. Available at: https://doi.org/10.3390/electronics12020284.

[19] Liu, M., Liu, Y., 2018. Price-based distributed offloading for mobile-edge computing with computation capacity constraints. IEEE Wirel. Commun. Lett. 7 (3), 420–423. http://dx.doi.org/10.1109/LWC.2017.2780128

[20] Anon, (2021). *MTU in LTE & 5G Transmission Networks – Part 1 | Nick vs Networking*. [online] Available at: https://nickvsnetworking.com/mtu-in-lte-5g-transmission-networks-part-1/ [Accessed 27 Aug. 2023].

[21] innopac.wits.ac.za. (n.d.). *Millennium Web Catalog*. [online] Available at: https://0-ieeexplore-ieee-org.innopac.wits.ac.za/document/10167924 [Accessed 27 Aug. 2023].

[22] Zenodo.org. (2016). Available at: https://zenodo.org/record/7161998/files/IORD100129CSE011.pdf [Accessed 27 Aug. 2023].

[23] Zheng, J., Sun, H., Wang, X., Liu, J. and Zhu, C. (2019). *A Batch-Normalized Deep Neural Networks and its Application in Bearing Fault Diagnosis*. [online] IEEE Xplore. doi:https://doi.org/10.1109/IHMSC.2019.00036.

[24] Pietro, M.D. (2022). *Deep Learning with Python: Neural Networks (complete tutorial)*. [online] Medium. Available at: https://towardsdatascience.com/deep-learning-with-python-neural-networks-complete-tutorial-6b53c0b06af0 [Accessed 25 Jan. 2022].

Appendix A: Non-technical report written in the style and format of a science magazine article

Optimizing Mobile Edge Computing: A Leap Towards Seamless Mobile Experiences

In a world increasingly reliant on smartphones and mobile devices, the race to meet the demands of both computational capacity and speed has reached a new zenith. The emergence of 5G/6G technology and the advancement of Machine Type Communicating Devices (MTCDs) have ushered in a new era of possibilities, but also underscored the limitations of our current telecommunications infrastructure. Enter Mobile Edge Computing (MEC), a solution aimed at revolutionizing how our devices handle complex tasks and data.

Imagine your smartphone as a miniature powerhouse, capable of performing intricate calculations, rendering graphics, and handling data transfers at the blink of an eye. However, this vision is often stymied by practical limitations. Mobile devices are bound by finite processing power, limited battery capacities, and constrained bandwidth. Tasks that demand heavy computational loads, instantaneous responses, constrained in size and weight, like augmented reality applications or real-time video streaming, often push these devices to their limits.

This is where MEC comes to the rescue. Imagine if, instead of your device struggling to execute resource-intensive tasks, it could offload these tasks to nearby servers that are purpose-built for such challenges. Alternatively, you could eliminate the battery altogether and extract energy wirelessly from ambient radio frequencies, such as those emitted by your home Wi-Fi. This would not only make your mobile phone substantially lighter but also enhance its sleekness.

Edge servers, located at the edge of the network, form the backbone of MEC, acting as efficient assistants to your MTCD. They process tasks swiftly, reducing latency and optimizing your device's battery life. What's even cooler is that both the edge server and the MTCD can handle a bunch of tasks all at once, as long as the timing is done right.

At the heart of this innovative approach lies a complex optimization problem: how can we ensure that tasks are offloaded in a way that minimizes latency, maximizes throughput (number of tasks computed per second), and conserves energy? This is the question that researcher Bryce Grahn is grappling with, and has come up with a fascinating solution.

The key lies in a deep learning-based algorithm that optimizes task offloading. By leveraging the power of artificial intelligence, and a neural network similar to how our brains work, this algorithm predicts the best way to distribute tasks between mobile devices and MEC servers. The result? A streamlined approach that significantly reduces the time it takes to complete tasks, while also optimizing the utilization of energy resources.One of the most remarkable aspects of this solution is its ability to strike a balance between local processing and offloading, energy harvesting, and receiving.

Rather than completely relieving the device's main processor, the algorithm intelligently allocates tasks in a way that optimizes performance. Through exhaustive simulations and meticulous analysis, the study revealed impressive results. The proposed solution achieved task throughput levels that were tantalizingly close to the optimal solution, reaching an impressive 98.87%. This translates to reduced latency, smoother performance. The engineering solution also tackled the problem of dropped tasks and queue buildup, further enhancing efficiency and user experience.

In the world of Mobile Edge Computing (MEC), promising advancements are met with their fair share of challenges. Security concerns emerge as MEC ventures beyond traditional firewalls, demanding security measures to safeguard data processing. The digital landscape isn't immune to inequality either as uneven access to MEC services due to geographical disparities could amplify service quality discrepancies and limit opportunities for some. Algorithmic bias poses a new frontier of concerns, with the proposed solutions potentially yielding less accurate results in uncharted environments. Meanwhile, the ambitious construction of MEC infrastructure requires a considerable financial investment, possibly favoring industry giants and leaving smaller players in the shadows.

As MEC paves the way for efficient energy harvesting, the introduction of additional infrastructure like RF power transmission towers carries both promises and challenges, potentially tipping the scales in environmental sustainability efforts.

In the face of these challenges, the world of Mobile Edge Computing (MEC) is abuzz with innovative solutions aimed at turning obstacles into opportunities. To counter security risks, experts emphasize the implementation of rigorous encryption protocols and advanced authentication mechanisms. Algorithmic bias finds its match in ongoing research that seeks to expand training datasets to encompass a diverse array of environments, ensuring accurate predictions across various scenarios

From an environmental perspective, the energy efficiency brought about by this solution is undoubtedly a boon. However, the deployment of additional infrastructure, such as RF power transmission towers, could raise questions about the overall sustainability of the technology. Striking a balance between progress and responsible resource utilization is imperative. Conversely, the team's use of energy harvesting techniques, particularly radio frequency energy harvesting, adds a sustainable twist to the innovation, aligning with the growing emphasis on eco-friendly solutions.

Despite these impressive strides, the path ahead is still dotted with opportunities for further refinement. The model's dependency on supervised learning techniques limits its effectiveness in handling unforeseen scenarios. However, the researchers are already exploring the potential of Deep Q learning, an adaptive technique that could enable devices to make offloading decisions in real time without having to know what's going on at other MTCD's. Furthermore, as technology continues to evolve, refining assumptions and incorporating more complex features could make the model even more capable in real-world scenarios.

In essence, the investigation project on optimizing task offloading within MEC has illuminated a path toward a more efficient and seamless mobile experience. It underscores the potential of harnessing AI to address the challenges of our increasingly demanding digital lives. As this technology continues to evolve, it brings with it promises of reduced latency, enhanced energy efficiency, and a glimpse into the future of mobile computing.

While the journey is far from over, the strides made in this realm point to a horizon where the limitations of today may well become the strengths of tomorrow. Through innovation, collaboration, and a dedication to sustainability, we are shaping a future where our mobile devices truly become the extensions of our capabilities that we've always envisioned.

Appendix B: Key Notations

Table 1: Summary of key notations (Design specifications)

Notation	Description	Value
T	Frame duration	10 ms
$Temp_i(t)$	Surrounding temperature (K)	298.15 Kelvin
$I_i^h(t)$	Energy conversion efficiency	0.7
$\mu_i(t)$	Attenuation factor	3.0
f_j^c	Channel frequency	1 MHz
B	MTCD bandwidth	5 MHZ
P_j^{BS}	Base station transmit power	100 W
f_i^L	MTCD CPU cycling frequency	650 MHz
f_j^{MEC}	MEC server CPU cycling frequency	16x2GHZ = 32 GHZ
$N_i^o(t)$	Noise power	2.057235×10^{-14} J.MHz
ζ	Number of cycles required to compute one bit	10 cycles/bits
β	Task size	1500 bytes
δ	Task result size	200 bytes
K_N	Boltzmann constant	$1.38 * 10^{-23}$ J/K
C	Speed of light	3.0×10^9 m/s
K_E	Constant	$10^{-27}\ s^3/cycles$
$E_i^h(t)$	Energy harvested	Dependent on frame partitioning
$E_i^L(t)$	Energy consumed during local computation	Dependent on frame partitioning
$E_i^O(t)$	Energy consumed when offloading	Dependent on frame partitioning
$E_i^R(t)$	Remaining surplus energy at the end of the frame.	Dependent on frame partitioning

Table 2: Summary of key notations (System state parameters)

Notation	Description	Value
$\vartheta_i^L(t)$	Number of tasks in the local queue	Average of 4 (Rayleigh System state parameter)
$\vartheta_i^O(t)$	Number of tasks in the offloading queue	Average of 6 (Rayleigh System state parameter)
$\vartheta_i^{MEC}(t)$	Number of tasks in the MEC server queue	Average of 5000 (Rayleigh System state parameter)
$h_i(t)$	Channel gain	$\sqrt{0.70}$. (Exponential random variable System state parameter)
$\lambda_i(t)$	Task arrival rate	90 (Poisson System state parameter)
$E_i^Q(t)$	Total energy in the energy queue	0.0005 J (Poisson System state parameter)
$P_i^T(t)$	MTCD transmit power	0.4-0.6 W (Random system state parameter)
$d_i(t)$	Distance to base station	Average of 80 m (Rayleigh System state

		parameter)

Table 3: Summary of key notations (System action parameters)

Notation	Description	Value
$\theta_i(t)$	Portion of tasks dropped	Dependent on frame partitioning
$\gamma_i(t)$	Portion of tasks offloaded	Dependent on frame partitioning
$\eta_i(t)$	Portion of tasks computed locally	Dependent on frame partitioning
$\alpha_i^1(t)$	Portion of frame used for harvesting energy	Predicated by the DNN model
$\alpha_i^2(t)$	Portion of frame used for offloading	Predicated by the DNN model
$\alpha_i^3(t)$	Portion of frame used for receiving	Predicated by the DNN model
$\alpha_i^4(t)$	Portion of frame used for local computation	Predicated by the DNN model

Appendix C: Deriving the objective function

$$(S, A) = (\{\vartheta_i^L(t), \vartheta_i^O(t), \vartheta_i^{MEC}(t), E_i^Q(t), h_i(t), \lambda_i(t), P_i^T(t), d_i(t)\}, \{\alpha_i^1(t), \alpha_i^2(t), \alpha_i^3(t), \alpha_i^4(t), \theta_i(t), \eta_i(t), \gamma_i(t)\})$$

$$\text{Max } \{Bits_{computed} + Bit_{offloaded} - Bits_{dropped}\}:$$

1. $Bits_{computed}$:

Let T_L represent the duration required to process both the newly assigned tasks and the pre-existing tasks within the computation queue.

$$T_i^L = T_{L,new} + T_{L,existing} = (\eta_i(t)\lambda_i(t)\beta \times 8)_{bits} \times \frac{\zeta}{f_i^L} + (\vartheta_i^L(t)\beta \times 8)_{bits} \times \frac{\zeta}{f_i^L}$$

$$\therefore T_i^L = \alpha_i^4(t)T = [(\eta_i(t)\,\lambda_i(t) + \vartheta_i^L(t))\beta \times 8]_{bits\ computed} \times \frac{\zeta}{f_i^L}$$

$$\therefore Bits_{computed} = [(\eta_i(t)\,\lambda_i(t) + \vartheta_i^L(t))\beta \times 8]_{bits\ computed} \times = \alpha_i^4(t)T \times \frac{f_i^L}{\zeta}$$

$$\therefore \eta_i(t) = [(\alpha_i^4(t)T \times \frac{f_i^L}{\zeta\beta\times 8}) - \vartheta_i^L(t)]/\,\lambda_i(t)$$

2. $Bit_{offloaded}$:

Let T_i^O represent the duration required to process, the newly assigned tasks, the pre-existing tasks within the offload queue, and the pre-existing tasks in the MEC queue.

Let $R_i^{UL} = Blog_2(1 + P_i^T(t)|h_i(t)|^2/N_0)$ and t^prepresent the uplink transmission rate and the propagation delay respectively.

$$\therefore T_i^O = T_{O,new} + T_{O,existing} + T_{O,MEC} = (\gamma_i(t)\lambda_i(t)\beta \times 8)_{bits}/R_i^{UL} + t^p + (\gamma_i(t)\lambda_i(t)(t)\beta \times 8)_{bits}\frac{\zeta}{f^{MEC}}$$
$$+ (\vartheta_i^O(t)\beta \times 8)_{bits}/R_i^{UL} + t^p + (\vartheta_i^O(t)\beta \times 8)_{bits} \times \frac{\zeta}{f^{MEC}}$$
$$+ (\vartheta_i^{MEC}(t) \times \beta \times 8)_{bits} \times \frac{\zeta}{f^{MEC}}$$

In practical scenarios, multiple high-performance MEC servers are present. In addition, computations occur concurrently in the MEC while simultaneous computation and transmission take place at the MTCD. Consequently, we make the assumption that both the propagation delay and the MEC computations are of negligible significance and can be disregarded.

$$\therefore T_i^O = \alpha_i^2(t)T = [(\gamma_i(t)\lambda_i(t) + \vartheta_i^O(t))(\beta \times 8)]_{bits\,offloaded}/Blog_2(1 + \frac{P_i^T(t)|h_i(t)|^2}{K_N TempB})$$

$$\therefore Bit_{offloaded} = [(\gamma_i(t)\lambda_i(t) + \vartheta_i^O(t))(\beta \times 8)]_{bits\,offloaded} = \alpha_i^2(t)TBlog_2(1 + \frac{P_i^T|h_i(t)|^2}{K_N TempB})$$

$$\therefore \gamma_i(t) = [\frac{\alpha_i^2(t)T}{\beta\times 8}Blog_2(1 + P_i^T|h_i(t)|^2/(K_N TempB)) - \vartheta_i^O(t)]/\lambda_i(t)$$

3. $Bit_{dropped}$:

   Dropped tasks represent incoming tasks that cannot be accommodated within either queue due to insufficient time or energy for computation or transmission within the current frame. These tasks will have the opportunity for allocation in the subsequent frame, thus preventing improper allocation to either queue.

$$\therefore Bits_{dropped} = \theta_i(t)\lambda_i(t)\beta \times 8 = (1 - \eta_i(t) - \gamma_i(t))\lambda_i(t)\beta \times 8 =$$
$$(1 - [(\alpha_i^4(t)T \times \frac{f_i^L}{\zeta\beta\times 8}) - \vartheta_i^L(t)]/\lambda_i(t) - [\frac{\alpha_i^2(t)T}{\beta\times 8}Blog_2(1 + P_i^T|h_i(t)|^2/(K_N TempB))$$
$$- \vartheta_i^O(t)]/\lambda_i(t))\lambda_i(t)\beta \times 8$$

4. Simplifying the objective function:

$$\{Bits_{computed} + Bit_{offloaded} - Bits_{dropped}\} = \alpha_i^4(t)T \times \frac{f_i^L}{\zeta} + \alpha_i^2(t)TBlog_2(1 + \frac{P_i^T|h_i(t)|^2}{K_N TempB}) - [$$
$$(1 - [(\alpha_i^4(t)T \times \frac{f_i^L}{\zeta\beta\times 8}) - \vartheta_i^L(t)]/\lambda_i(t) - [\frac{\alpha_i^2(t)T}{\beta\times 8}Blog_2(1 + P_i^T|h_i(t)|^2/(K_N TempB))$$
$$- \vartheta_i^O(t)]/\lambda_i(t))\lambda_i(t)\beta \times 8]$$

$$\max_{(\alpha_i^2,\alpha_i^4(t), f_i^L, P_i^T(t), h_i(t), \vartheta_i^L(t), \vartheta_i^O(t), \lambda_i(t))} \{2\alpha_i^4(t)Tf_i^L/\zeta + 2\alpha_i^2(t)TBlog_2(1 + \frac{P_i^T(t)|h_i(t)|^2}{K_N TempB}) - (8 \times \beta)(\vartheta_i^L(t) + \vartheta_i^O(t) + \lambda_i(t))\} \quad (13)$$

Appendix D: Deriving the objective function constraints

1. $Q_1(S, A)$: Energy constraint:

   Adequate energy must be available to either offload or compute all assigned tasks within the frame.

$$\therefore E_i^h(t) + E_i^Q(t) - (E_i^L(t) + E_i^O(t)) \geq 0$$
$$\therefore \alpha_i^1(t)\; TI_i^h(t)P_j^{BS}(\frac{c}{4\pi d_i(t)f_j^c})^{\mu}|h_i(t)|^2 + E_i^Q(t) - K_E(f_i^L) \times \alpha_i^4(t) \times T - P_i^T(t)\times \alpha_i^2(t) \times T \geq 0$$

2. $Q_2(S, A)$: Receiving constraint:

   Every task transmitted within the current frame is ensured ample time for reception. i.e:
   Receiving time >= Number of tasks transmitted/Downlink rate

$$\therefore \alpha_i^3(t)T \geq (\gamma_i(t)\lambda_i(t) + \vartheta_i^O)(\delta \times 8)/Blog_2(1 + \frac{P_j^{BS}|h_i(t)|^2}{K_N TempB})$$

$$\therefore \alpha_i^3(t)T \geq (\alpha_i^2(t)TBlog_2(1 + \frac{P_i^T|h_i(t)|^2}{K_N TempB}))/(\beta \times 8))(\delta \times 8)/Blog_2(1 + \frac{P_j^{BS}|h_i(t)|^2}{K_N TempB})$$

$$\therefore \alpha_i^3(t) - \frac{\alpha_i^2(t)\delta log_2(1+P_i^T|h_i(t)|^2/(K_N TempB))}{\beta log_2(1+P_j^{BS}|h_i(t)|^2/(K_N TempB))} \geq 0$$

3. $Q_3(S, A)$: Alpha duration constraint:

   In order to optimize the utilization of the entire frame, it's essential for either the offloading scheme $\alpha_1 + \alpha_2 + \alpha_3$ or the local computation scheme $(\alpha_1 + \alpha_4)$ to sum up to 1. The rationale behind the use of 'or' is that, depending on the system state, a higher number of tasks might be allocated to either the local or offloading queue. This approach avoids assigning additional tasks to the local or transmission queue to achieve the necessary sum of 1, especially if doing so would be inefficient.

$$\therefore Max((\alpha_i^1(t) + \alpha_i^2(t) + \alpha_i^3(t)), (\alpha_i^1(t) + \alpha_i^4(t))) \leq 1$$

4. $Q_4(S, A)$: Tasks computed constraint:

   The fraction representing the number of tasks assigned to the local computation queue lies between zero and one, inclusive.

$$0 \leq \eta_i(t) \leq 1$$

$$\therefore 0 \leq (\alpha_i^4(t)T \times \frac{f_i^L}{\zeta\beta\times 8} - \vartheta_i^L(t))/\lambda_i(t) \leq 1$$

5. $Q_5(S, A)$: Tasks offloaded constraint:

   The fraction representing the number of tasks assigned to the offload queue lies between zero and one, inclusive.

$$0 \leq \gamma_i(t) \leq 1$$

$$\therefore 0 \leq [\frac{\alpha_i^2(t)T}{\beta\times 8}Blog_2(1 + P_i^T|h_i(t)|^2/(K_N TempB)) - \vartheta_i^O(t)]/\lambda_i(t) \leq 1$$

6. $Q_6(S, A)$: Tasks dropped constraint:

   The fraction representing the number of tasks dropped lies between zero and one, inclusive.

$$0 \leq \theta_i(t) \leq 1$$

$$0 \leq 1 - \eta_i(t) - \gamma_i(t) \leq 1$$

$$\therefore 0 \leq 1 - (\alpha_i^4(t)T \times \frac{f_i^L}{\zeta\beta\times 8} - \vartheta_i^L(t))/\lambda_i(t) - [\frac{\alpha_i^2(t)T}{\beta\times 8}Blog_2(1 + P_i^T|h_i(t)|^2/(K_N TempB)) - \vartheta_i^O(t)]/\lambda_i(t) \leq 1$$

Appendix E: Contains the Matlab code for producing the training dataset, wherein the optimal action is determined for a given system state by maximizing the objective function.

```
clear;
clear all;
clc;
```

```
%System specifications
T = 0.01; % Frame duration in seconds
J = 298.15; % Room temperature in Kelvin
L = 0.7; % Energy efficiency factor
b = 3; % Attenuation factor
M = 0.001*10^9; %Carrier frequency
Z = 5*10^6; % Bandwidth
Q = 100; % MTCD transmit power
R = 0.65*10^9; % MTCD CPU cycling frequency
d = 10; % Number of clock cycles required to compute 1 bit
e = 1500; % Task size in bytes
q = 200; % Task result size in bytes
K = 1.38*10^(-23); % Boltzmann constant
I = 3*10^8; %Speed of light
U = 10^(-27); % Constant
g = 80; % Average distance from Base station

% A = alpha1, B = alpha2, C = Alpha 3, D = alpha 4

syms A B C D

% Number of samples to simulate
num_samples = 100;

% Generate a matrix of zeros with the specified dimensions
num_rows = num_samples;
num_cols = 18;
data = zeros(num_rows, num_cols);

%Tasks
lambda = 90; % Average rate of task arrivals
a = poissrnd(lambda, num_samples, 1); % Generate task arrivals using Poisson distribution

%Existing tasks in the local computation queue (Rayleigh distribution)
%V = round(randi([0, 6], 1, num_samples));
V = round(raylrnd(4,num_samples,1));

%Existing tasks in the transmission queue (Rayleigh distribution)
% W = round(randi([0, 6], 1, num_samples));
W = round(raylrnd(6,num_samples,1));

%NOMA Transmission power
lower_bound = 40;
upper_bound = 60;
% Generate random integers between lower_bound and upper_bound
P = randi([lower_bound, upper_bound], 1, num_samples)/100;

% Generate energy in energy queue using Rayleigh distribution
Energy = 7; % Average energy
dec = randi([1, 1000], 1, num_samples)/1000000; % Generate random numbers between 1 and 1000
E = raylrnd(Energy,num_samples,1);
E = E/10000 + dec';
E = round(E,6);

% Generate channel gain using an exponential random variable
averageHSquared = 0.7;
rateParameter = 1 / averageHSquared;
```

```
% Generate exponential random samples and clamp to maximum value of 1
H = sqrt(min(exprnd(rateParameter, 1, num_samples), 1));
H = round(H,3)

for i = 1:num_samples

    % Objective function handle
    O = (-(V(i)+W(i)+a(i))*e*8 + 2*D*T*R/d + 2*B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z)));

    % Constraint function handles
    N1 = ((A*T*L*Q*H(i)*H(i)*(I/(4*g*pi*M))^b + E(i)) >= (U*D*T*(R)^3 + P(i)*B*T));
    N2 = ((0 <= (C - (B*q*log2(1+P(i)*H(i)*H(i)/(K*J*Z)))/(e*log2(1+Q*H(i)*H(i)/(K*J*Z))))) & ((C - ...
        (B*q*log2(1+P(i)*H(i)*H(i)/(K*J*Z)))/(e*log2(1+Q*H(i)*H(i)/(K*J*Z)))) <= 0.005));
    N3 = ((0 <= ((D*T*R/(8*d*e))-V(i))/a(i)) & (((D*T*R/(8*d*e))-V(i))/a(i) <= 1));
    N4 = ((0 <= (B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i)) &
((B*T*Z*log2(1+P(i)*(H(i)*H(i))/...
        (K*J*Z))/(e*8)-W(i))/a(i) <= 1));
    N5 = ( (0 <= (((D*T*R/(8*d*e))-V(i))/a(i) +
(B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i))) & ...
        ((((D*T*R/(8*d*e))-V(i))/a(i) + (B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i)) <= 1));
    N6 = ((0 <= (1-max((A+B+C),(A+D)))) & ((1-(max((A+B+C),(A+D)))) <= 0.01));

    % Determine the alpha values that maximise the objective function
    result = optRandSolution({matlabFunction(O), matlabFunction(N1), matlabFunction(N2),
matlabFunction(N3), ...
        matlabFunction(N4), matlabFunction(N5), matlabFunction(N6)}); %(alpha1, alpha2, alpha3, alpha4)

    %   Local, offloaded, dropped
    S = ((result(4)*T*R/(8*d*e))-V(i))/a(i); %Portion of tasks computed locally
    G = (result(2)*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i); % Portion of tasks offloaded
    F = 1-S-G; % Portion of tasks dropped
    numTasks = (S + G) * a(i) + V(i) + W(i);
    tasksDropped = F * a(i);
    %Remaining energy at the end of the frame
    E_rem = result(1)*T*L*Q*H(i)*H(i)*(I/(4*g*pi*M))^b + E(i) -U*result(4)*T*(R)^3 -0.2*result(2)*T;

    %Insert into data matrix
    data(i, :) = [V(i), W(i), 0, a(i), H(i), E(i), P(i), result(1), result(2), result(3), result(4), ...
        result(5), S, G, F, E_rem, numTasks, tasksDropped];
end

% Define column names
columnNames = {'Local_Queue', 'Transmission_Queue', 'MEC_Queue', 'Task_Arrival_Rate',
'Channel_Attenuation', ...
    'Energy_Queue', 'MTCD_Transmission_Power', 'Alpha_1', 'Alpha_2', 'Alpha_3', 'Alpha_4', ...
    'Computation_Rate', 'Local_Computation_Ratio', 'Offloading_Ratio', 'Dropped_Ratio',
'Remaining_Energy', ...
    'Tasks_Computed', 'Tasks_Dropped'};
% Specify the file name
filename = 'Training.csv';
% Convert data matrix to a table
dataTable = array2table(data, 'VariableNames', columnNames);
% Write the table to the CSV file
writetable(dataTable, filename);

disp('Matrix exported to CSV file successfully.');
```

```
function result = optRandSolution(system_functions)

    % Define number of random guesses
    num_guesses = 2000000;

    % Objective function handle
    O = system_functions{1};

    % Constraint function handles
    C1 = system_functions{2};
    C2 = system_functions{3};
    C3 = system_functions{4};
    C4 = system_functions{5};
    C5 = system_functions{6};
    C6 = system_functions{7};

    % Define constraint bounds
    lower_bound_A = 0.1;
    upper_bound_A = 0.7;
    lower_bound_B = 0.1;
    upper_bound_B = 0.7;
    lower_bound_C = 0;
    upper_bound_C = 0.1;
    lower_bound_D = 0.1;
    upper_bound_D = 0.9;

    % Initialize best f and corresponding values of A, B, C, D
    best_result = 0;
    best_A = NaN;
    best_B = NaN;
    best_C = NaN;
    best_D = NaN;

    % Perform random search
    for guess = 1:num_guesses
        A = lower_bound_A + (upper_bound_A - lower_bound_A) * rand();
        B = lower_bound_B + (upper_bound_B - lower_bound_B) * rand();
        C = lower_bound_C + (upper_bound_C - lower_bound_C) * rand();
        D = lower_bound_D + (upper_bound_D - lower_bound_D) * rand();

        if C1(A, B, D) && C2(B, C) && C3(D) && C4(B) && C5(B, D) && C6(A, B, C, D)
            current_result = O(B, D);
            if current_result > best_result
                best_result = current_result;
                best_A = A;
                best_B = B;
                best_C = C;
                best_D = D;
            end
        end
    end

    result = [best_A, best_B, best_C, best_D, best_result];
end
```

Appendix F: Contains the Matlab code for producing a dataset where a random action is solved for a given system state

```
clear;
clear all;
clc;

%System specifications
T = 0.01; % Frame duration in seconds
J = 298.15; % Room temperature in Kelvin
L = 0.7; % Energy efficiency factor
b = 3; % Attenuation factor
M = 0.001*10^9; %Carrier frequency
Z = 5*10^6; % Bandwidth
Q = 100; % MTCD transmit power
R = 0.65*10^9; % MTCD CPU cycling frequency
d = 10; % Number of clock cycles required to compute 1 bit
e = 1500; % Task size in bytes
q = 200; % Task result size in bytes
K = 1.38*10^(-23); % Boltzmann constant
I = 3*10^8; %Speed of light
U = 10^(-27); % Constant
g = 80; % Average distance from Base station

% A = alpha1, B = alpha2, C = Alpha 3, D = alpha 4

syms A B C D

dataTable = readtable('results.csv');

mat = table2array(dataTable(:, :));

[numRows, numCols] = size(mat);

V = mat(:,1);
W = mat(:,2);
a = mat(:,3);
H = mat(:,4);
E = mat(:,5);
P = mat(:,6);


for i = 1:numRows

    % Objective function handle
    O = (-(V(i)+W(i)+a(i))*e*8 + 2*D*T*R/d + 2*B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z)));

    % Constraint function handles
    N1 = ((A*T*L*Q*H(i)*H(i)*(I/(4*g*pi*M))^b + E(i)) >= (U*D*T*(R)^3 + P(i)*B*T));
    N2 = ((0 <= (C - (B*q*log2(1+P(i)*H(i)*H(i)/(K*J*Z)))/(e*log2(1+Q*H(i)*H(i)/(K*J*Z))))) & ((C - ...
        (B*q*log2(1+P(i)*H(i)*H(i)/(K*J*Z)))/(e*log2(1+Q*H(i)*H(i)/(K*J*Z)))) <= 0.005));
    N3 = ((0 <= ((D*T*R/(8*d*e))-V(i))/a(i)) & (((D*T*R/(8*d*e))-V(i))/a(i) <= 1));
    N4 = ((0 <= (B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i)) &
((B*T*Z*log2(1+P(i)*(H(i)*H(i))...
        /(K*J*Z))/(e*8)-W(i))/a(i) <= 1));
    N5 = ( (0 <= (((D*T*R/(8*d*e))-V(i))/a(i) +
(B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i))) & ...
```

```
        ((((D*T*R/(8*d*e))-V(i))/a(i) + (B*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i)) <= 1));
    N6 = ((0 <= (1-max((A+B+C),(A+D)))) & ((1-(max((A+B+C),(A+D)))) <= 0.01));

    % Determine the alpha values that maximise the objective function
    result = randSolution({matlabFunction(O), matlabFunction(N1), matlabFunction(N2), matlabFunction(N3),
...
        matlabFunction(N4), matlabFunction(N5), matlabFunction(N6)}); %(alpha1, alpha2, alpha3, alpha4)

    %   Local, offloaded, dropped
    S = ((result(4)*T*R/(8*d*e))-V(i))/a(i); %Portion of tasks computed locally
    G = (result(2)*T*Z*log2(1+P(i)*(H(i)*H(i))/(K*J*Z))/(e*8)-W(i))/a(i); % Portion of tasks offloaded
    F = 1-S-G; % Portion of tasks dropped
    numTasks = (S + G) * a(i) + V(i) + W(i);
    tasksDropped = F * a(i);
    %Remaining energy at the end of the frame
    E_rem = result(1)*T*L*Q*H(i)*H(i)*(I/(4*g*pi*M))^b + E(i) -U*result(4)*T*(R)^3 -0.2*result(2)*T;

    %Insert into data matrix
    data(i, :) = [V(i), W(i), 0, a(i), H(i), E(i), P(i), result(1), result(2), result(3), result(4), ...
        result(5), S, G, F, E_rem, numTasks, tasksDropped];
end

% Define column names
columnNames = {'Local_Queue', 'Transmission_Queue', 'MEC_Queue', 'Task_Arrival_Rate',
'Channel_Attenuation', ...
    'Energy_Queue', 'MTCD_Transmission_Power', 'Alpha_1', 'Alpha_2', 'Alpha_3', 'Alpha_4', ...
    'Computation_Rate', 'Local_Computation_Ratio', 'Offloading_Ratio', 'Dropped_Ratio',
'Remaining_Energy', ...
    'Tasks_Computed', 'Tasks_Dropped'};
% Specify the file name
filename = 'randomGuessing.csv';
% Convert data matrix to a table
dataTable = array2table(data, 'VariableNames', columnNames);
% Write the table to the CSV file
writetable(dataTable, filename);

disp('Matrix exported to CSV file successfully.');
```

```
function result = randSolution(system_functions)

    % Define number of random guesses
    num_guesses = 2000000;

    % Objective function handle
    O = system_functions{1};

    % Constraint function handles
    C1 = system_functions{2};
    C2 = system_functions{3};
    C3 = system_functions{4};
    C4 = system_functions{5};
    C5 = system_functions{6};
    C6 = system_functions{7};

    % Define constraint bounds
    lower_bound_A = 0.1;
    upper_bound_A = 0.7;
    lower_bound_B = 0.1;
    upper_bound_B = 0.7;
    lower_bound_C = 0;
    upper_bound_C = 0.1;
    lower_bound_D = 0.1;
    upper_bound_D = 0.9;

    % Initialize best f and corresponding values of A, B, C, D
    best_result = 0;
    best_A = NaN;
    best_B = NaN;
    best_C = NaN;
    best_D = NaN;

    % Perform random search
    for guess = 1:num_guesses
        A = lower_bound_A + (upper_bound_A - lower_bound_A) * rand();
        B = lower_bound_B + (upper_bound_B - lower_bound_B) * rand();
        C = lower_bound_C + (upper_bound_C - lower_bound_C) * rand();
        D = lower_bound_D + (upper_bound_D - lower_bound_D) * rand();

        if C1(A, B, D) && C2(B, C) && C3(D) && C4(B) && C5(B, D) && C6(A, B, C, D)
            current_result = O(B, D);
            if current_result > best_result
                best_result = current_result;
                best_A = A;
                best_B = B;
                best_C = C;
                best_D = D;
                result = [best_A, best_B, best_C, best_D, best_result];
                return;
            end
        end
    end

    result = [best_A, best_B, best_C, best_D, best_result];
end
```

Appendix G: Presents the code for a DNN model and environment simulation using Google Colab, Python, TensorFlow, and Keras.

```
import necessary libraries:

[ ]
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Model
```

```
import matplotlib.pyplot as plt
from IPython.display import display
import joblib
from tensorflow.keras.models import load_model
import joblib
```

Load and preprocess the dataset:

```
[ ]
# Load the dataset
df = pd.read_csv('training.csv')

# Drop rows with NULL or NaN values
df.dropna(inplace=True)

#Reset index
df = df.reset_index(drop=True)
display(df)

# Separate features and target variables
features = df[['Local_Queue', 'Transmission_Queue', 'Task_Arrival_Rate',
'Channel_Attenuation', 'Energy_Queue', 'MTCD_Transmission_Power']]
targets = df[['Alpha_1', 'Alpha_2', 'Alpha_3', 'Alpha_4','Local_Computation_Ratio',
'Offloading_Ratio']]

# Normalize features
scaler = StandardScaler()
scaled_features = scaler.fit_transform(features)
```

Split the dataset into training and testing sets:

```
[ ]
X_train, X_test, y_train, y_test = train_test_split(scaled_features, targets,
test_size=0.2, random_state=42)
```

Build and train the multi-target deep neural network:

```
[ ]
import keras
from keras import layers
import keras.backend as K

# Custom loss function with constraints
def custom_loss_with_constraints(y_true, y_pred):
    mse = tf.keras.losses.MeanSquaredError()
    mse_loss = mse(y_true, y_pred)

    # Apply constraints penalty
    alpha_sum_1_to_3 = y_pred[:, 0] + y_pred[:, 1] + y_pred[:, 2]
    alpha_sum_1_to_4 = y_pred[:, 0] + y_pred[:, 3]
    alpha_sum_n_and_y = y_pred[:, 4] + y_pred[:, 5]

    constraint_penalty = K.mean(K.maximum(0.0, alpha_sum_1_to_3 - 1.0)) + \
                         K.mean(K.maximum(0.0, alpha_sum_1_to_4 - 1.0)) + \
                         K.mean(K.maximum(0.0, alpha_sum_n_and_y - 1.0))
    total_loss = mse_loss + constraint_penalty

    return total_loss

# Build the neural network model
modelTuned = keras.Sequential([
    layers.Dense(512, activation='relu', input_shape=(X_train.shape[1],)),
    layers.BatchNormalization(),
    layers.Dense(256, activation='relu'),
    layers.BatchNormalization(),
    layers.Dense(128, activation='relu'),
    layers.BatchNormalization(),
    layers.Dense(64, activation='relu'),
    layers.BatchNormalization(),
    layers.Dense(32, activation='relu'),
    layers.BatchNormalization(),
    layers.Dense(16, activation='sigmoid'),
    layers.Dense(targets.shape[1])  # Number of target variables
])

# Compile the model
modelTuned.compile(optimizer='adam', loss=custom_loss_with_constraints)
# modelTuned.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = modelTuned.fit(X_train, y_train, epochs=65, batch_size=32, validation_split=0.2)
Epoch 1/65
585/585 [==============================] - 6s 6ms/step - loss: 0.0334 - val_loss: 0.0154
Epoch 2/65
585/585 [==============================] - 4s 7ms/step - loss: 0.0149 - val_loss: 0.0140
Epoch 3/65
585/585 [==============================] - 4s 7ms/step - loss: 0.0142 - val_loss: 0.0136
Epoch 4/65
585/585 [==============================] - 4s 6ms/step - loss: 0.0139 - val_loss: 0.0132
585/585 [==============================] - 4s 6ms/step - loss: 0.0126 - val_loss: 0.0135
Epoch 64/65
585/585 [==============================] - 4s 6ms/step - loss: 0.0126 - val_loss: 0.0130
Epoch 65/65
585/585 [==============================] - 4s 6ms/step - loss: 0.0127 - val_loss: 0.0131
```

Test loss

```
[ ]
lossTuned = modelTuned.evaluate(X_test, y_test)
```

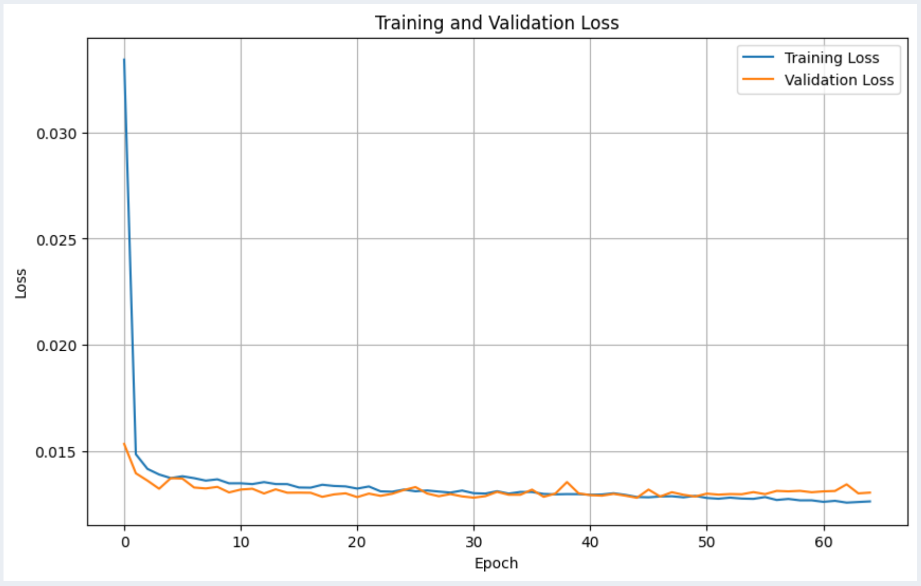
```
print(f"Test loss: {lossTuned}")
183/183 [==============================] - 0s 2ms/step - loss: 0.0131
Test loss: 0.013125120662152767
```

Train loss

```
[ ]
lossTuned = modelTuned.evaluate(X_train, y_train)
print(f"Train loss: {lossTuned}")
731/731 [==============================] - 1s 2ms/step - loss: 0.0123
Train loss: 0.012287535704672337
```

Plot training and validation loss

```
[ ]
# Plot training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid(True)
plt.show()
```



Use the model to predict actions for known system states

Load best model and scaler

```
[ ]
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import custom_object_scope
```

```
# Define your custom loss function
def custom_loss_with_constraints(y_true, y_pred):
    mse = tf.keras.losses.MeanSquaredError()
    mse_loss = mse(y_true, y_pred)

    # Apply constraints penalty
    alpha_sum_1_to_3 = y_pred[:, 0] + y_pred[:, 1] + y_pred[:, 2]
    alpha_sum_1_to_4 = y_pred[:, 0] + y_pred[:, 3]
    alpha_sum_n_and_y = y_pred[:, 4] + y_pred[:, 5]

    constraint_penalty = K.mean(K.maximum(0.0, alpha_sum_1_to_3 - 1.0)) + \
                         K.mean(K.maximum(0.0, alpha_sum_1_to_4 - 1.0)) + \
                         K.mean(K.maximum(0.0, alpha_sum_n_and_y - 1.0))
    total_loss = mse_loss + constraint_penalty

    return total_loss


# Load the scaler
scaler_filename = "scaler.pkl"
loaded_scaler = joblib.load(scaler_filename)

# Load the trained DNN model
model_filename = "trained_model.h5"

# Load the model within a custom_object_scope to register the custom loss function
with custom_object_scope({'custom_loss_with_constraints': custom_loss_with_constraints}):
    loaded_model = load_model(model_filename)

modelTuned = loaded_model
scaler = loaded_scaler
```

Predict actions

```
[ ]
# Predict Alpha values using the trained model
predicted_alphas = modelTuned.predict(scaled_features)

# Define the column names
column_names = ['Predicted_Alpha_1', 'Predicted_Alpha_2', 'Predicted_Alpha_3',
'Predicted_Alpha_4', 'Predicted_Local_Computation_Ratio', 'Predicted_Offloading_Ratio']

# Convert predicted_alphas to a DataFrame
df_predicted_alphas = pd.DataFrame(data=predicted_alphas, columns=column_names)

display(df_predicted_alphas)
```

Create a new table with the system state **and** predicted actions

```
[ ]
df_p = pd.concat([features, df_predicted_alphas], axis=1)
display(df_p)
```

Calculate tasks dropped, throughput, tasks computed, remaining tasks and the remaining energy

```
[ ]
# Create a new DataFrame to store the results
df_results = pd.DataFrame()

# Constants and column names
T = 0.01
J = 298.15
L = 0.7
b = 3
M = 0.001*10**9
Z = 5 * 10**6
Q = 100
R = 0.65 * 10**9
d = 10.0
e = 1500.0
q = 200
K = 1.38*10**(-23)
I = 3*10**8
U = 10**(-27)
g = 80
P = 'MTCD_Transmission_Power'
H = 'Channel_Attenuation'
B = 'Predicted_Alpha_2'
D = 'Predicted_Alpha_4'
a = 'Task_Arrival_Rate'
W = 'Transmission_Queue'
V = 'Local_Queue'
E = 'Energy_Queue'

df_results = df_p

# Loop through each row in df_p
for index, row in df_p.iterrows():
    Predicted_Throughput = (-(row['Local_Queue'] + row['Transmission_Queue'] +
row['Task_Arrival_Rate']) * e * 8 + 2 * row['Predicted_Alpha_4'] * T * R / d + 2 *
row['Predicted_Alpha_2'] * T * Z * np.log2(1 + row['MTCD_Transmission_Power'] *
(row['Channel_Attenuation'] * row['Channel_Attenuation']) / (K * J * Z)))
    df_results.loc[index, 'Predicted_Throughput'] = Predicted_Throughput

for index, row in df_p.iterrows():
    Maximum_Local_Computation_Ratio = ((row[D] * T * R / (8 * d * e)) - row[V]) / row[a]
    Predicted_Local_Computation_Ratio = row['Predicted_Local_Computation_Ratio']
    if Maximum_Local_Computation_Ratio < Predicted_Local_Computation_Ratio:
      tasks_remaining_in_local_queue = (Predicted_Local_Computation_Ratio -
Maximum_Local_Computation_Ratio) * row[a]
      df_results.loc[index, 'Tasks_Remaining_In_Local_Queue'] =
tasks_remaining_in_local_queue
    else:
      df_results.loc[index, 'Tasks_Remaining_In_Local_Queue'] = 0

    Maximum_Offloading_Ratio = (row[B] * T * Z * np.log2(1 + row[P] * (row[H]**2) / (K * J
* Z))/(8 * e) - row[W])/row[a]
    Predicted_Offloading_Ratio = row['Predicted_Offloading_Ratio']

    if Maximum_Offloading_Ratio < Predicted_Offloading_Ratio:
      tasks_remaining_in_offloading_queue = (Predicted_Offloading_Ratio -
Maximum_Offloading_Ratio) * row[a]
      df_results.loc[index, 'Tasks_Remaining_In_Offloading_Queue'] =
tasks_remaining_in_offloading_queue
    else:
      df_results.loc[index, 'Tasks_Remaining_In_Offloading_Queue'] = 0

for index, row in df_p.iterrows():
    Predicted_Dropped_Ratio = 1 - row['Predicted_Local_Computation_Ratio'] -
row['Predicted_Offloading_Ratio']
    df_results.loc[index, 'Predicted_Dropped_Ratio'] = Predicted_Dropped_Ratio

for index, row in df_p.iterrows():
    Predicted_Tasks_Computed = (row['Predicted_Local_Computation_Ratio'] +
row['Predicted_Offloading_Ratio']) * row[a] + row[V] + row[W] -
(row['Tasks_Remaining_In_Offloading_Queue'] + row['Tasks_Remaining_In_Local_Queue'])
    df_results.loc[index, 'Predicted_Tasks_Computed'] = Predicted_Tasks_Computed

    Predicted_Tasks_Dropped = row['Predicted_Dropped_Ratio'] * row[a]
    df_results.loc[index, 'Predicted_Tasks_Dropped'] = Predicted_Tasks_Dropped

    Predicted_Remaining_Energy = (row['Predicted_Alpha_1'] * T * L * Q * (row[H]**2) * (I
/ (4 * g * np.pi * M))**b) + row[E] - U * row[D] * T * (R**3) - row[P] * row[B] * T
    df_results.loc[index, 'Predicted_Remaining_Energy'] = Predicted_Remaining_Energy

# Display the resulting DataFrame
display(df_results)

# Write the DataFrame to a CSV file
df_results.to_csv("results.csv", index=False)
```

Clean up valid results

Determine how the model would perform if these edge cases were implemented

```
[ ]
# Drop rows with NULL or NaN values
```

```
df_results.dropna(inplace=True)

# Restrict tasks computed to the theoretical maximum when valid (When there are negative
tasks dropped and the remaining energy is greater then zero)
for index, row in df_results.iterrows():
  if (row['Predicted_Tasks_Dropped'] < 0) & (row['Predicted_Remaining_Energy'] > 0):
    df_results.loc[index,'Predicted_Tasks_Dropped'] = 0.0
    df_results.loc[index,'Predicted_Dropped_Ratio'] = 0.0
    df_results.loc[index, 'Predicted_Tasks_Computed'] = row[V] + row[W] + row[a] -
(row['Tasks_Remaining_In_Offloading_Queue'] + row['Tasks_Remaining_In_Local_Queue'])
    df_results.loc[index, 'Predicted_Throughput'] = row['Predicted_Throughput'] -
abs(row["Predicted_Tasks_Dropped"]) * e * 8 #Negative tasks effect the computation rate

# Remove surplus tasks and increase the energy (When there are Negative tasks dropped but
also negative renmaining energy)
for index, row in df_results.iterrows():
  if (row['Predicted_Tasks_Dropped'] < 0) & (row['Predicted_Remaining_Energy'] < 0):
     true_remaining_energy = row['MTCD_Transmission_Power'] *
row['Predicted_Tasks_Dropped'] * e * 8 / (Z * np.log2(1 + row['MTCD_Transmission_Power'] *
(row['Channel_Attenuation'] * row['Channel_Attenuation']) / (K * J * Z))) +
row['Predicted_Remaining_Energy']
     df_results.loc[index,'Predicted_Tasks_Dropped'] = 0.0
     df_results.loc[index, 'Predicted_Remaining_Energy'] = true_remaining_energy
     df_results.loc[index, 'Predicted_Throughput'] = row['Predicted_Throughput'] -
abs(row["Predicted_Tasks_Dropped"]) * e * 8 #Negative tasks effect the computation rate

# Drop additional tasks if there wasn't enough energy to compute them (Positive tasks
dropped but negative remaining energy)
for index, row in df_results.iterrows():
  if (row['Predicted_Tasks_Dropped'] > 0) & (row['Predicted_Remaining_Energy'] < 0):
    if row['Predicted_Alpha_2'] > row['Predicted_Alpha_4']:
     new_dropped = row['Predicted_Remaining_Energy'] * (Z * np.log2(1 +
row['MTCD_Transmission_Power'] * (row['Channel_Attenuation'] * row['Channel_Attenuation'])
/ (K * J * Z)))/(row['MTCD_Transmission_Power'] * e * 8) + row['Predicted_Tasks_Dropped']
     df_results.loc[index,'Predicted_Tasks_Dropped'] = new_dropped
     df_results.loc[index, 'Predicted_Remaining_Energy'] = 0.0

# Delete rows that don't meet the alpha constraint
df_results = df_results[
    (df_results['Predicted_Alpha_1'] + df_results['Predicted_Alpha_2'] +
df_results['Predicted_Alpha_3'] <= 1) &
    (df_results['Predicted_Alpha_1'] + df_results['Predicted_Alpha_4'] <= 1)
]

[ ]
display(df_results)
```

Clean up optimal predictions dataset so we can compare results

[ ]

```
# Get the index values that have not been dropped in df_results
remaining_indexes = df_results.index

# Extract the corresponding entries in df
df_cleaned = df.loc[remaining_indexes]

display(df_cleaned)

# Restrict tasks computed to the maximum when valid
for index, row in df_cleaned.iterrows():
  if (row['Tasks_Dropped'] < 0.001):
    df_cleaned.loc[index,'Tasks_Dropped'] = 0.0
    df_cleaned.loc[index,'Dropped_Ratio'] = 0.0
    df_cleaned.loc[index, 'Tasks_Computed'] = row[V] + row[W] + row[a]
    df_cleaned.loc[index, 'Throughput'] = (row[V] + row[W] + row[a]) * e * 8

display(df_cleaned)
```

Compare predictions to the optimal solution

```
[ ]
average_optimal_computation_rate = df_cleaned["Throughput"].mean()
print("Average throughput of the optimal solution determined by brute forcing all possible
combinations:", average_optimal_computation_rate)
average_predicted_computation_rate = df_results["Predicted_Throughput"].mean()
print("Average throughput for DNN predictions:", average_predicted_computation_rate)
stats = (average_predicted_computation_rate/average_optimal_computation_rate) * 100

print("Final verdict: Our DNN model reaches ", stats, "% of the optimal solution")
```

```
Average throughput of the optimal solution determined by brute forcing all possible
combinations: 1163776.8808959578
Average throughput for DNN predictions: 1150647.0878185323
Final verdict: Our DNN model reaches  98.87179464612518 % of the optimal solution
```

Compare results to random guessing

```
[ ]
# Load the dataset
df_rand = pd.read_csv('randomGuessing.csv')

average_random_computation_rate = df_rand["Computation_Rate"].mean()
print("Average throughput for random guesses:", average_random_computation_rate)
stats = (average_random_computation_rate/average_optimal_computation_rate) * 100
stats2 =
((average_predicted_computation_rate-average_random_computation_rate)/average_random_compu
tation_rate) * 100

print("Final verdict: Random guessing reaches ", stats, "% of the optimal solution")
print("Final verdict: Our DNN model achieves a ", stats2, "% improvement over random
guessing")

Average throughput for random guesses: 618833.4425624569
Final verdict: Random guessing reaches  53.17457776666221 % of the optimal solution
Final verdict: Our DNN model achieves a  85.9380907169381 % improvement over random
guessing
```

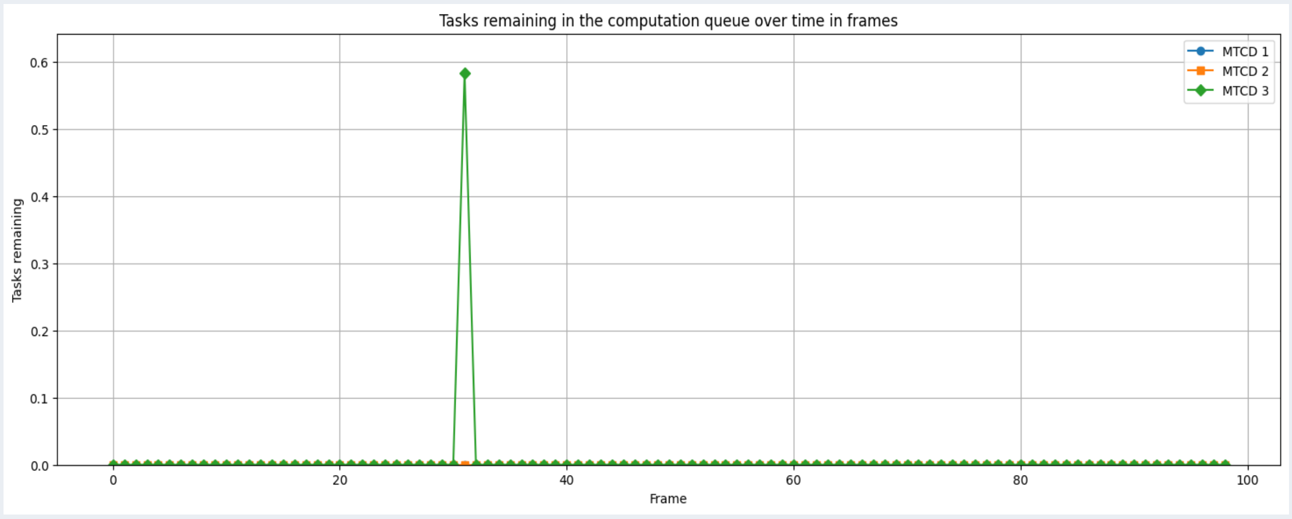
Export the learned model to an environment simulation with multiple MTCD's and frames

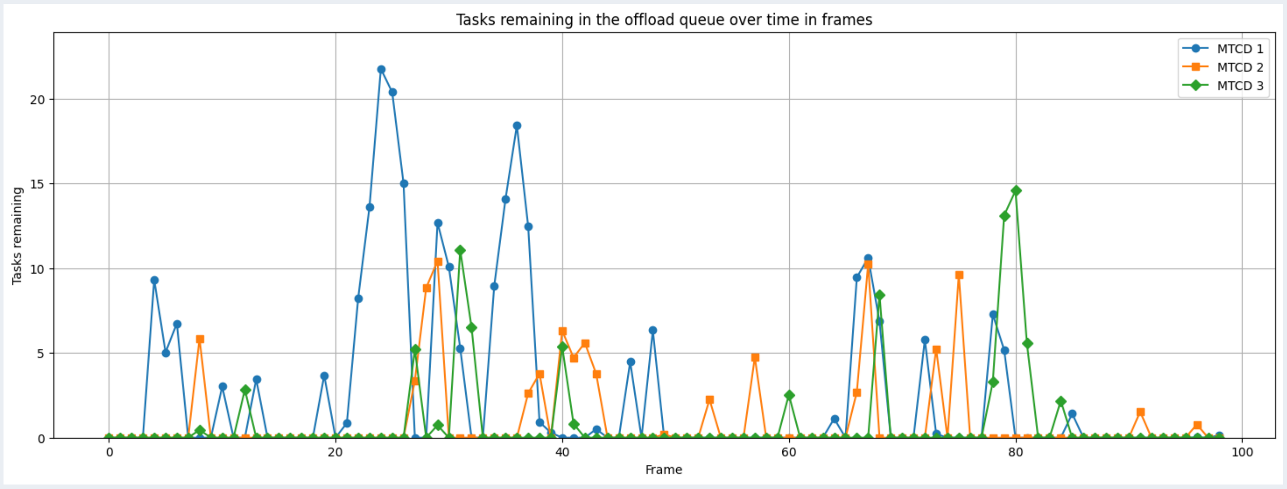
```
[ ]
import numpy as np
import pandas as pd
from scipy.stats import rayleigh, expon
from sympy import symbols
from scipy.optimize import minimize

# Constants and column names
T = 0.01
J = 298.15
L = 0.7
b = 3
M = 0.001*10**9
Z = 5 * 10**6
Q = 100
R = 0.65 * 10**9
d = 10.0
e = 1500.0
q = 200
K = 1.38*10**(-23)
I = 3*10**8
U = 10**(-27)
g = 80
# P = 'MTCD_Transmission_Power'
# a = 'Task_Arrival_Rate'
# W = 'Transmission_Queue'
# V = 'Local_Queue'
# E = 'Energy_Queue'

num_frames = 100
num_mtcd = 3

#Define result arrays
Local_remaining = np.zeros((num_mtcd, num_frames-1))
Offloading_remaining = np.zeros((num_mtcd, num_frames-1))
Energy_queue = np.zeros((num_mtcd, num_frames-1))
Tasks_dropped = np.zeros((num_mtcd, num_frames-1))
```

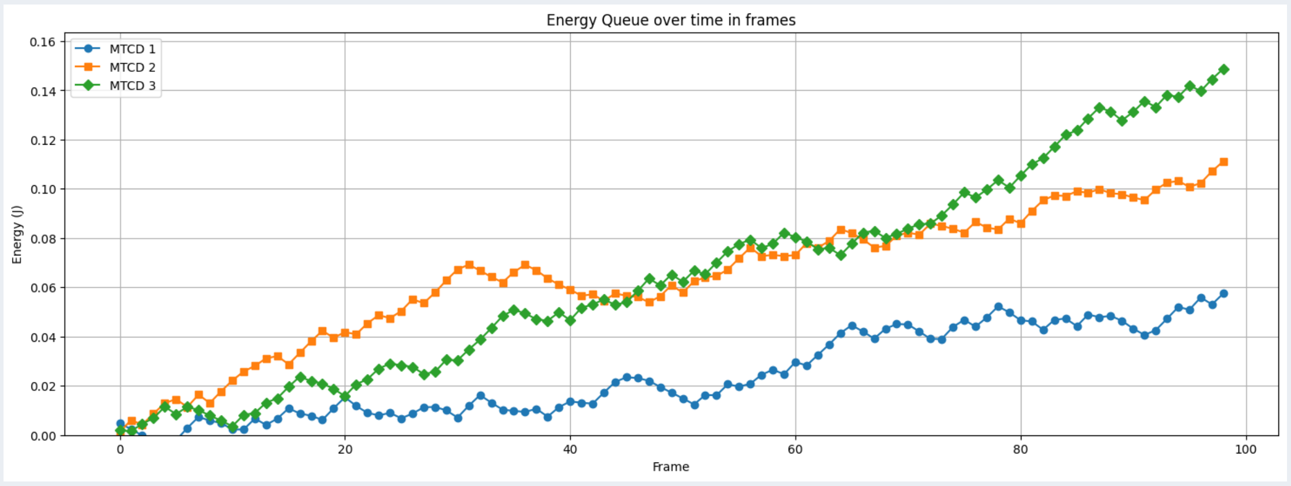
```
Tasks_computed = np.zeros((num_mtcd, num_frames-1))
Objective = np.zeros((num_mtcd, num_frames-1))
Optimal = np.zeros((num_mtcd, num_frames-1))

#MTCD Loop

for j in range(0,num_mtcd):

  E = np.random.rayleigh(scale=0.001, size=1) # Generate energy in energy queue using
Rayleigh distribution
  V = np.random.randint(0, 7, size=1) # Existing tasks in the local computation queue
(Rayleigh distribution)
  W = np.random.randint(0, 7, size=1) # Existing tasks in the transmission queue (Rayleigh
distribution)
  # Tasks
  lambda_val = 90
  a = np.random.poisson(lambda_val, num_frames)
  # NOMA Transmission power
  lower_bound = 40
  upper_bound = 60
  P = np.random.randint(lower_bound, upper_bound + 1, size=num_frames) / 100
  # Generate channel gain using an exponential random variable
  averageHSquared = 0.7
  rateParameter = 1 / averageHSquared
  H = np.sqrt(np.minimum(expon.rvs(scale=1/rateParameter, size=num_frames), 1))
  # Tasks dropped
  Predicted_Tasks_Dropped = 0;

# Frame Loop

  for i in range(0,num_frames-1):
    #Features
    # Constructing features as a 2D array
    features = np.column_stack((V, W, a[i], H[i], E, P[i]))
    scaled_features = scaler.fit_transform(features)

    predicted_alphas = modelTuned.predict(scaled_features)
    A1 = predicted_alphas[0][0]
    A2 = predicted_alphas[0][1]
    A3 = predicted_alphas[0][2]
    A4 = predicted_alphas[0][3]
    pre_comp_ratio = predicted_alphas[0][4]
    pre_off_ratio = predicted_alphas[0][5]

    a[i] = a[i] + Predicted_Tasks_Dropped

    #Perform necessary calculations
    predicted_throughput = (-(V + W + a[i]) * e * 8 + 2 * A4 * T * R / d + \
                          2 * A2 * T * Z * np.log2(1 + P[i] * (H[i] * H[i]) / (K * J * Z)))
    Maximum_Local_Computation_Ratio = ((A4 * T * R / (8 * d * e)) - V) / a[i]
    if Maximum_Local_Computation_Ratio < pre_comp_ratio:
        tasks_remaining_in_local_queue = (pre_comp_ratio - 
Maximum_Local_Computation_Ratio) * a[i]
    else:
        tasks_remaining_in_local_queue = 0

    Maximum_Offloading_Ratio = (A2 * T * Z * np.log2(1 + P[i] * (H[i]**2) / (K * J * 
Z))/(8 * e) - W)/a[i]
    if Maximum_Offloading_Ratio < pre_off_ratio:
        tasks_remaining_in_offloading_queue = (pre_off_ratio - Maximum_Offloading_Ratio) * 
a[i]
    else:
        tasks_remaining_in_offloading_queue = 0

    Predicted_Dropped_Ratio = 1 - pre_comp_ratio - pre_off_ratio
    Predicted_Tasks_Computed = (pre_comp_ratio + pre_off_ratio) * a[i] + V + W - 
(tasks_remaining_in_local_queue + tasks_remaining_in_offloading_queue)
    Predicted_Tasks_Dropped = Predicted_Dropped_Ratio * a[i]
    Predicted_Remaining_Energy = (A1 * T * L * Q * (H[i]**2) * (I / (4 * g * np.pi * 
M))**b) + E - U * A4 * T * (R**3) - P[i] * A2 * T

    # Restrict tasks computed to the theoretical maximum when valid (When there are 
negative tasks dropped and the remaining energy is greater then zero)
    if (Predicted_Tasks_Dropped < 0) & (Predicted_Remaining_Energy > 0):
      Predicted_Tasks_Dropped = 0.0
      Predicted_Dropped_Ratio = 0.0
      Predicted_Tasks_Computed = V + W + a[i] - (tasks_remaining_in_local_queue + 
tasks_remaining_in_offloading_queue)
      predicted_throughput = predicted_throughput - abs(Predicted_Tasks_Dropped) * e * 8

    # Remove surplus tasks and increase the energy (When there are Negative tasks dropped 
but also negative renmaining energy)
    if (Predicted_Tasks_Dropped < 0) & (Predicted_Remaining_Energy < 0):
      true_remaining_energy = P[i] * Predicted_Tasks_Dropped * e * 8 / (Z * np.log2(1 + 
P[i] * (H[i] * H[i]) / (K * J * Z))) + Predicted_Remaining_Energy
      Predicted_Tasks_Dropped = 0.0
      Predicted_Remaining_Energy = true_remaining_energy
      predicted_throughput = predicted_throughput - abs(Predicted_Tasks_Dropped) * e * 8

    # Drop additional tasks if there wasn't enough energy to compute them (Positive tasks 
dropped but negative remaining energy)
    if (Predicted_Tasks_Dropped > 0) & (Predicted_Remaining_Energy < 0):
      if A2 > A4:
        new_dropped = Predicted_Remaining_Energy * (Z * np.log2(1 + P[i] * (H[i]*H[i]) / 
(K * J * Z)))/(P[i] * e * 8) + Predicted_Tasks_Dropped
        Predicted_Tasks_Dropped = new_dropped
        Predicted_Remaining_Energy = 0.0

    #optimal_ratio = predicted_throughput/((a[i] + V + W)*(8 * e))
    optimal_ratio = Predicted_Tasks_Computed/(a[i] + V + W)

    #Results to be passed onto the next frame
    V = tasks_remaining_in_local_queue
    W = tasks_remaining_in_offloading_queue
    E = Predicted_Remaining_Energy

    #Stores results
    Local_remaining[j][i] = tasks_remaining_in_local_queue
    Offloading_remaining[j][i] = tasks_remaining_in_offloading_queue
    Energy_queue[j][i] = Predicted_Remaining_Energy
    Tasks_dropped[j][i] = Predicted_Tasks_Dropped
    Tasks_computed[j][i] = Predicted_Tasks_Computed
    Objective[j][i] = predicted_throughput
    Optimal[j][i] = optimal_ratio
```

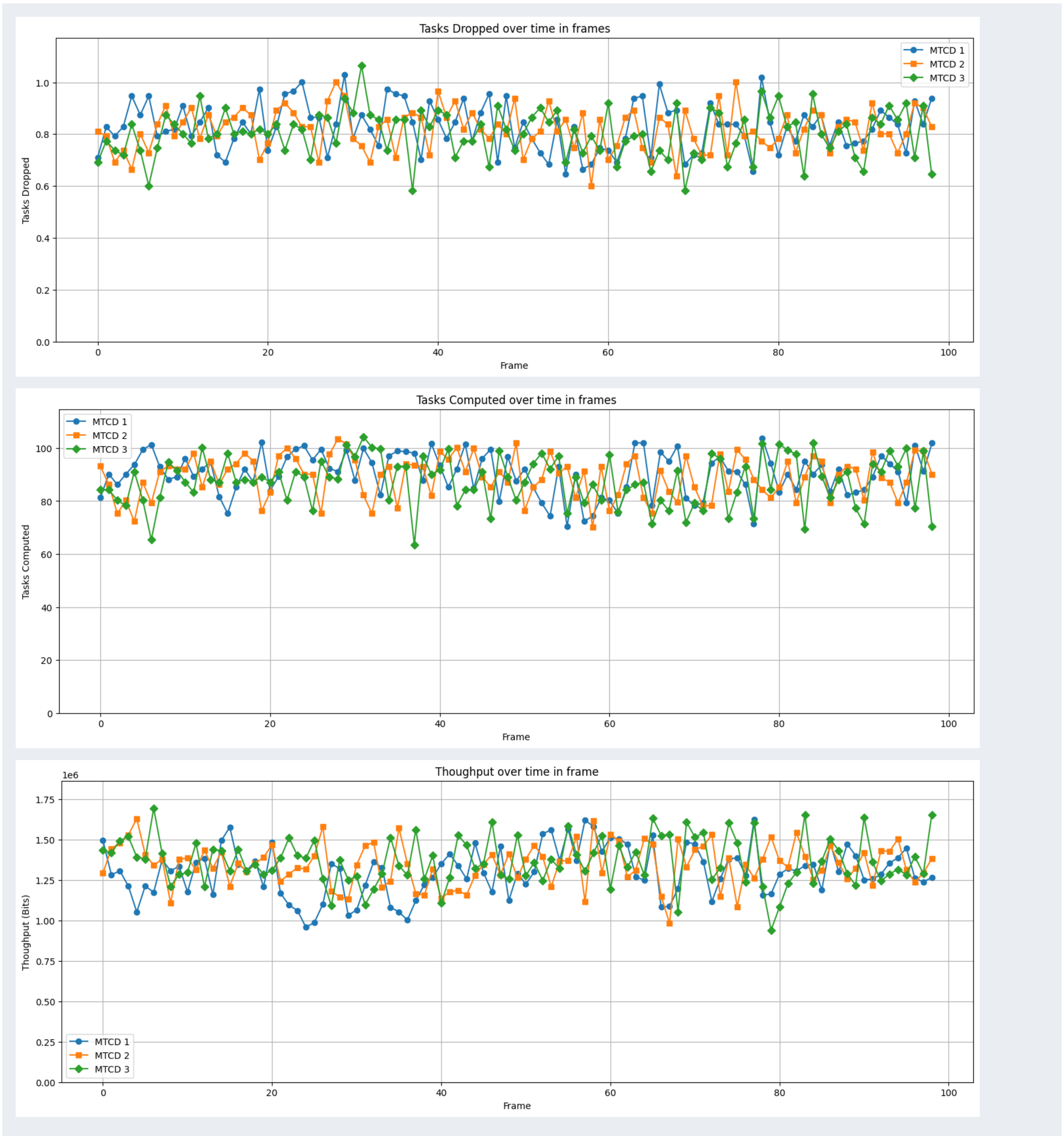
Plot results

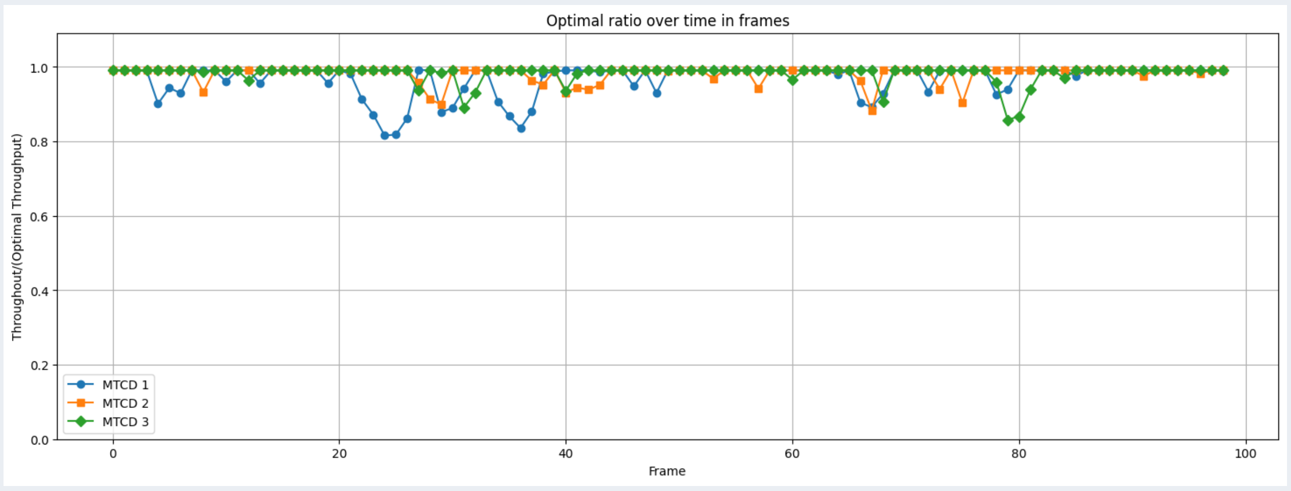
```
[ ]
# Arrays of 2D arrays for easy iteration
array_list = [Local_remaining, Offloading_remaining, Energy_queue, Tasks_dropped,
Tasks_computed, Objective, Optimal]
array_names = ["Tasks remaining in the computation queue over time in frames", "Tasks
remaining in the offload queue over time in frames", "Energy Queue over time in frames",
"Tasks Dropped over time in frames", "Tasks Computed over time in frames", "Thoughput over
time in frame", "Optimal ratio over time in frames"]
x_names = ["Tasks remaining", "Tasks remaining", "Energy (J)", "Tasks Dropped", "Tasks
Computed", "Thoughput (Bits)", "Throughout/(Optimal Throughput)"]

# Define line styles and markers
line_styles = ['-', '--', '-.', ':', '-', '--']
markers = ['o', 's', 'D', '^', '*', 'x']

# Plotting
for i, array in enumerate(array_list):
    rows, cols = array.shape

    plt.figure(figsize=(18, 6))
    for j in range(rows):
        # plt.plot(range(cols), array[j], linestyle=line_styles[j % len(line_styles)],
        #          marker=markers[j % len(markers)], label=f'MTCD {j + 1}')
        plt.plot(range(cols), array[j],marker=markers[j % len(markers)], label=f'MTCD {j +
1}')

    plt.title(f'{array_names[i]}')
    plt.xlabel('Frame')
    plt.ylabel(x_names[i])
    plt.legend()
    plt.grid()
    ymin = 0
    ymax = 1.1*np.max(array)
    plt.ylim(ymin, ymax)

    plt.show()
```








Tasks Dropped over time in frames
Tasks Dropped
Frame
MTCD 1
MTCD 2
MTCD 3
Tasks Computed over time in frames
Tasks Computed
Frame
MTCD 1
MTCD 2
MTCD 3
Thoughput over time in frame
1e6
Thoughput (Bits)
Frame
MTCD 1
MTCD 2
MTCD 3



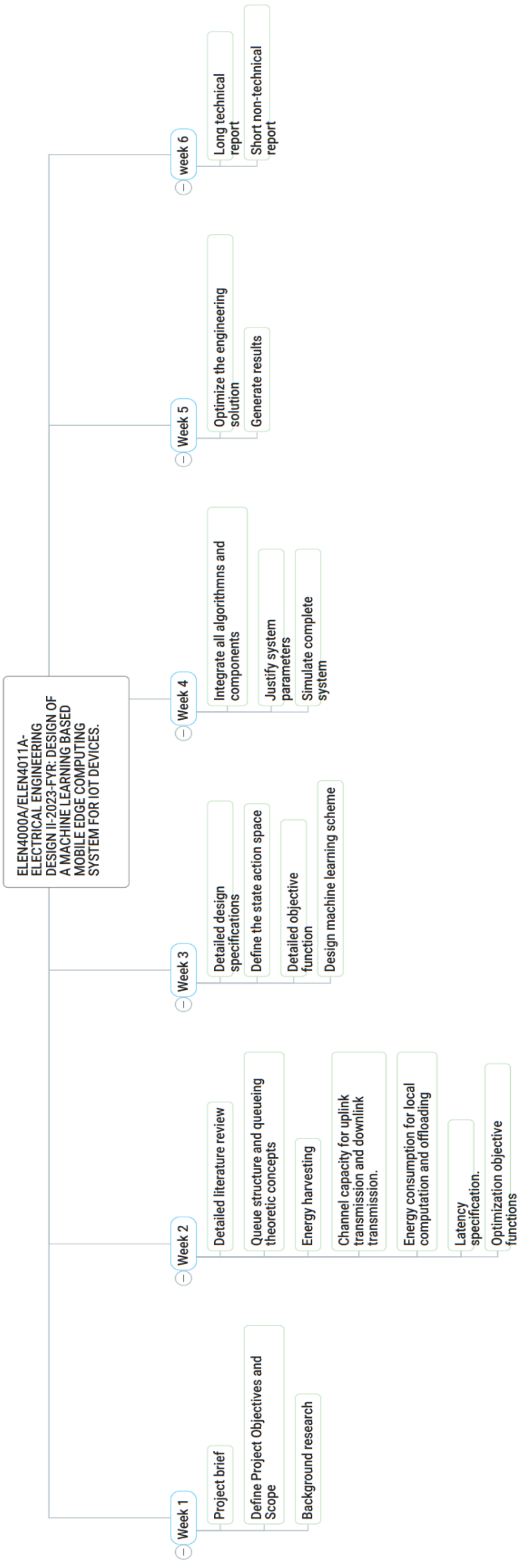
Save model and scaler

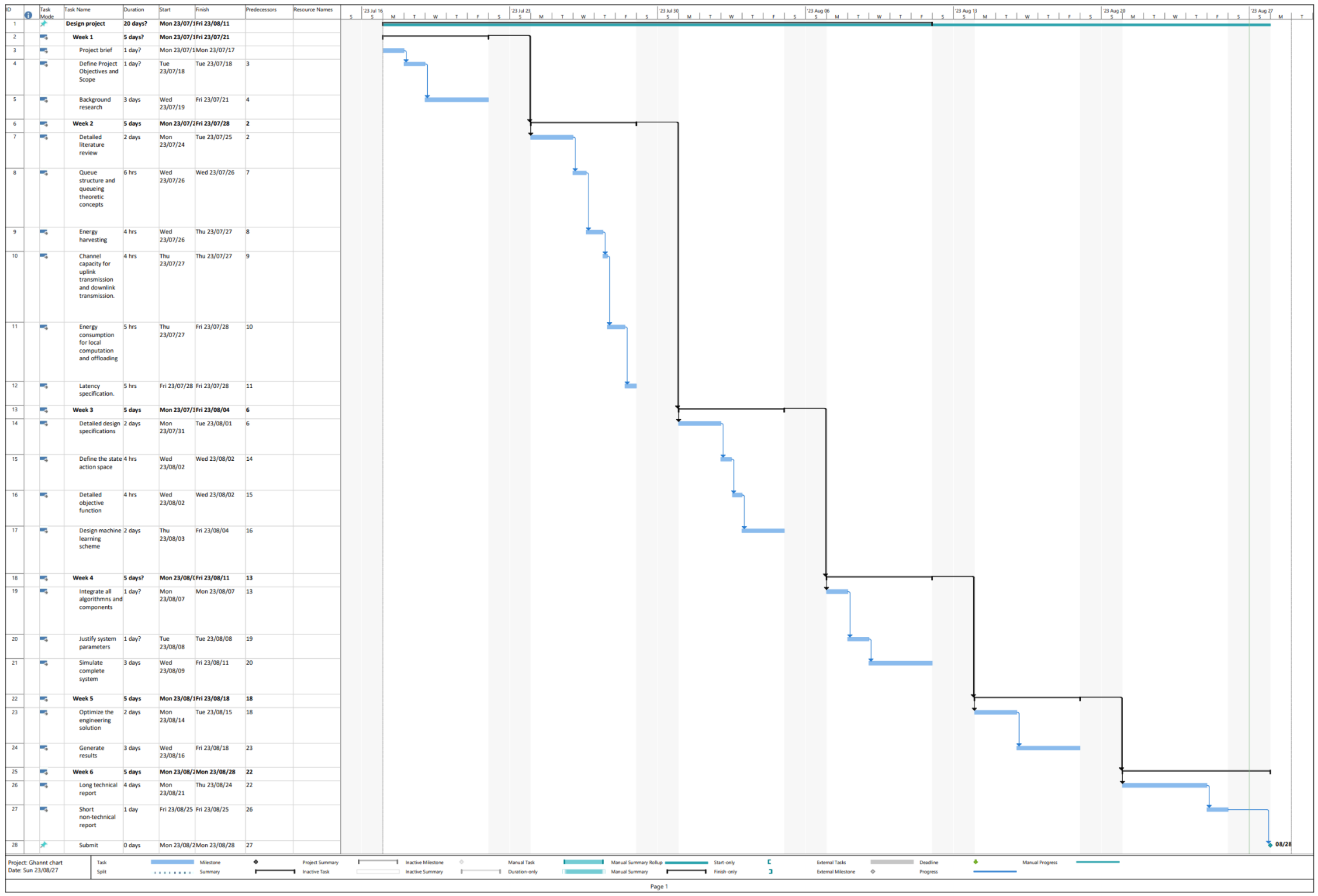
```
[ ]
import joblib
# # Save the scaler
# scaler_filename = "scaler.pkl"
# joblib.dump(scaler, scaler_filename)
# # Save the trained DNN model
# model_filename = "trained_model.h5"  # You can use a different extension if you're using
a different model format
# modelTuned.save(model_filename)

Colab paid products - Cancel contracts here
```

Appendix H: Github repo

https://github.com/bcgrahn/DESIGN-OF-A-MACHINE-LEARNING-BASED-MOBILE-EDGE-COMPUTING-SYSTEM-FOR-IOT-DEVICES

Appendix I: Work breakdown structure and Gantt chart


ELEN4000A/ELEN4011A-ELECTRICAL ENGINEERING DESIGN II-2023-FYR: DESIGN OF A MACHINE LEARNING BASED MOBILE EDGE COMPUTING SYSTEM FOR IOT DEVICES.
Week 1
Project brief
Define Project Objectives and Scope
Background research
Week 2
Detailed literature review
Queue structure and queueing theoretic concepts
Energy harvesting
Channel capacity for uplink transmission and downlink transmission.
Energy consumption for local computation and offloading
Latency specification.
Optimization objective functions
Week 3
Detailed design specifications
Define the state action space
Detailed objective function
Design machine learning scheme
Week 4
Integrate all algorithms and components
Justify system parameters
Simulate complete system
Week 5
Optimize the engineering solution
Generate results
week 6
Long technical report
Short non-technical report



ID		Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resource Names
1			**Design project**	**20 days?**	**Mon 23/07/1**	**Fri 23/08/11**		
2			**Week 1**	**5 days?**	**Mon 23/07/1**	**Fri 23/07/21**		
3			Project brief	1 day?	Mon 23/07/1	Mon 23/07/17		
4			Define Project Objectives and Scope	1 day?	Tue 23/07/18	Tue 23/07/18	3	
5			Background research	3 days	Wed 23/07/19	Fri 23/07/21	4	
6			**Week 2**	**5 days**	**Mon 23/07/2**	**Fri 23/07/28**	**2**	
7			Detailed literature review	2 days	Mon 23/07/24	Tue 23/07/25	2	
8			Queue structure and queueing theoretic concepts	6 hrs	Wed 23/07/26	Wed 23/07/26	7	
9			Energy harvesting	4 hrs	Wed 23/07/26	Thu 23/07/27	8	
10			Channel capacity for uplink transmission and downlink transmission.	4 hrs	Thu 23/07/27	Thu 23/07/27	9	
11			Energy consumption for local computation and offloading	5 hrs	Thu 23/07/27	Fri 23/07/28	10	
12			Latency specification.	5 hrs	Fri 23/07/28	Fri 23/07/28	11	
13			**Week 3**	**5 days**	**Mon 23/07/3**	**Fri 23/08/04**	**6**	
14			Detailed design specifications	2 days	Mon 23/07/31	Tue 23/08/01	6	
15			Define the state action space	4 hrs	Wed 23/08/02	Wed 23/08/02	14	
16			Detailed objective function	4 hrs	Wed 23/08/02	Wed 23/08/02	15	
17			Design machine learning scheme	2 days	Thu 23/08/03	Fri 23/08/04	16	
18			**Week 4**	**5 days?**	**Mon 23/08/0**	**Fri 23/08/11**	**13**	
19			Integrate all algorithmns and components	1 day?	Mon 23/08/07	Mon 23/08/07	13	
20			Justify system parameters	1 day?	Tue 23/08/08	Tue 23/08/08	19	
21			Simulate complete system	3 days	Wed 23/08/09	Fri 23/08/11	20	
22			**Week 5**	**5 days**	**Mon 23/08/1**	**Fri 23/08/18**	**18**	
23			Optimize the engineering solution	2 days	Mon 23/08/14	Tue 23/08/15	18	
24			Generate results	3 days	Wed 23/08/16	Fri 23/08/18	23	
25			**Week 6**	**5 days**	**Mon 23/08/2**	**Mon 23/08/28**	**22**	
26			Long technical report	4 days	Mon 23/08/21	Thu 23/08/24	22	
27			Short non-technical report	1 day	Fri 23/08/25	Fri 23/08/25	26	
28			Submit	0 days	Mon 23/08/2	Mon 23/08/28	27	

Project: Ghannt chart
Date: Sun 23/08/27