19L720 – PROJECT WORK

EFFICIENT TASK OFFLOADING USING TRAJECTORY PREDICTION

Deepika E (20L106)
Kavinraj S (20L121)
Subhashini J (20L146)
Sureshkumar R (20L147)

Dissertation submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

Branch: ELECTRONICS AND COMMUNICATION ENGINEERING

of Anna University

November 2023

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641 004

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641 004

EFFICIENT TASK OFFLOADING USING TRAJECTORY PREDICTION

Bonafide record of work done by

DEEPIKA E (20L106)

KAVINRAJ S (20L121)

SUBHASHINI J (20L146)

SURESHKUMAR R (20L147)

Project Report (Phase1) submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

Branch: ELECTRONICS AND COMMUNICATION ENGINEERING

of Anna University

NOVEMBER 2023

.....	.....
Dr. SIVARAJ D	**Dr. KRISHNAVENI V**
Faculty guide	Head of the Department

Certified that the candidate was examined in the viva-voice examination held on .....

.....
(Internal Examiner)

CONTENTS

CHAPTER **Page No.**

ACKNOWLEDGEMENT

We would like to extend our sincere thanks to **Dr. Prakasan K,** Principal, PSG College of Technology, for his kind patronage.

We also wish to express our sincere thanks to **Dr. V. Krishnaveni**, Professor and Head-of-the Department, Electronics and Communication Engineering.

We are grateful for the support extended by our Programme Coordinator, **Dr. K. V. Anusuya**, Associate Professor (CAS), Department of Electronics and Communication Engineering.

We would like to express our deepest gratitude to our project guide **Dr. D. Sivaraj**, Assistant Professor (Sr. Gr.), Department of Electronics and Communication Engineering for his constant support, motivation, and guidance throughout the project.

We thank all the teaching and non-teaching staff members of the Department of Electronics and Communication Engineering for their support.

Finally, we would like to thank all my student colleagues who have helped us in completing this project without any hassles.

SYNOPSIS

The advent of Artificial Intelligence has led to the development of intelligent vehicles with diverse functionalities and complex system architectures, resulting in an increase in computational tasks within vehicles. However, vehicles often face limitations in terms of local computing resources. To address this challenge, mobile edge computing has emerged as a solution to alleviate the burden on local computing resources. In the context of Telematics, when a vehicle offloads a computational task to an edge server, the communication time between the vehicle and the base station is reduced due to the high-speed movement of the vehicle. However, if the vehicle moves away from the current base station before the computation is completed, it may not receive the computation results in a timely manner.

The project's main idea is to enhance the decision-making process for task offloading by incorporating trajectory prediction of vehicles to solve the problem of short communication time between vehicles and base stations due to high-speed movement of vehicles. The proposed framework uses reinforcement techniques such as Q learning and Deep Q learning to predict the future locations and movements of vehicles, allowing for proactive and context-aware task offloading decisions. The decision-making process involves determining whether a task should be processed locally on a vehicle or offloaded to a Roadside Unit (RSU). The framework consists of an agent, which represents the vehicle, and an RSU. The agent observes the current state, including vehicle resources, network conditions, and task characteristics. Based on this information, the agent decides whether to offload the task to the RSU or process it locally. The decision is made by estimating the expected long-term rewards using the reinforcement learning algorithm. Additionally, trajectory prediction is utilized to estimate the future movement patterns of vehicles enabling efficient task offloading.

The proposed approach is evaluated through simulations and experiments using real-world vehicle edge computing scenarios. The results demonstrate that the combination of DQN-based decision-making and trajectory prediction significantly improves the performance of task offloading. The approach achieves higher offloading accuracy, reduced latency and improved resource utilization. Ultimately, this work contributes to the development of intelligent and efficient task offloading techniques in vehicle edge computing.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The advancement of Telematics has brought about a significant increase in the demand for computational resources, particularly for time-sensitive tasks such as pattern recognition, image and video processing, target detection, and route planning. Such tasks need to get the results in the shortest time to help the vehicle make the next decision quickly. However, due to the limited computing resources of the vehicle, it is impossible to complete the task computation locally in a short time when facing computationally intensive and latency-sensitive tasks. Facing this challenge, Vehicle Edge Computing (VEC) provides a solution. Offloading tasks to the edge server allows the computational tasks to be completed using the computational resources of the edge server. In VEC, roadside units (RSUs) can provide richer computational and storage resources. In the context of mobile edge computing, different task offloading algorithms have different effects in different application scenarios. The merit of task offloading algorithms will directly determine the delay and energy consumption of task offloading.

In our project, we have successfully developed an algorithm called DQN (Deep Reinforcement Q Network) for efficient task offloading from vehicles to Roadside Units (RSUs). A key aspect of our algorithm is considering delay factors before deciding whether to offload a task to an RSU. This means that the vehicle itself can make an informed decision based on various factors to determine whether it can handle the computation internally or if offloading is necessary. By incorporating delay considerations into the decision-making process, we aim to optimize the overall performance and resource utilization in the system. This approach allows vehicles to intelligently and effectively manage their computing tasks, ensuring timely and efficient execution while minimizing communication delays.

Also, the traditional schemes only consider that the user does not leave the communication range of the base station. However, in the context of the Internet of Vehicles (IoV), intelligent vehicles travel very fast on the highway with speeds up to 85 mph, which brings new challenges to the communication between vehicle and base station. Due to the short communication range of 5G communication, the communication time between vehicles and the base station may be very short, which directly limits the data transmission. Moreover, if the

vehicle has left the communication range of the current base station before the RSU completes the task calculation, the results cannot be returned to the vehicle. Therefore, the traditional offloading scheme will become ineffective in the IoV.

1.1 NEED FOR THE WORK

With the development of Artificial Intelligence, the intelligent vehicle with diverse functions and complex system architectures brings more computing tasks to vehicles. Due to insufficient local computing resources for the vehicles edge computing is seen as the solution to relieve local computing pressure. When the vehicle offloads the computation task to the edge server, the communication time between the vehicle and the base station becomes shorter due to the high-speed movement of the vehicle. If the vehicle leaves the current base station before the computation is completed, the vehicle will not be able to obtain the computation results in time. Therefore, a task offloading scheme based on trajectory prediction solves the problem of short communication time between vehicles and base stations.

1.2 PROBLEM DEFINITION

The computational demand for delay-sensitive tasks, such as route planning in platoons and video streaming between vehicles, has increased. These tasks require obtaining results in the shortest time possible to facilitate quick decision-making by the vehicles. However, the limited computing resources of the vehicles make it impossible to perform the required task computations locally within the desired timeframe, especially for computationally intensive and latency-sensitive tasks.

1.3 OBJECTIVES

The objective of this project is to decide the task computation locally or offloaded to nearby RSU. To assign the offloaded task computation to the nearby RSU or through future trajectory prediction. To transfer the result computed in the local RSU to the future RSU or the complete task is offloaded to the future RSU. To send the results from the future RSU to the respective vehicle.

1.4 REPORT ORGANISATION

The report is organized as follows. Chapter 2 reviews the literatures on various task offloading schemes and their limitations. Chapter 3 gives an overview of the methodologies used and flowchart of the proposed methodology. Chapter 4 explains the algorithms considered, real time task used in vehicular edge computing and the channel model used. Chapter 5 deals with the implementation of the proposed algorithm. Chapter 6 presents the discussion on the results obtained. Chapter 7 concludes with the summary of the project, followed by a discussion on its future scope and limitations.

CHAPTER 2

LITERATURE SURVEY

Paper [1] discusses the insufficient local computing resources for the vehicles mobile edge computing as a solution to relieve local computing pressure. The communication time betweenthe vehicle and the base station becomes shorter due to the high-speed movement of the vehicle. Therefore, trajectory prediction solves the problem of short communication time between vehicles and base stations.

Paper [2] discusses the partial task offloading problem in vehicular edge computing, where the vehicle computes some part of a task locally, and offload the remaining task to a nearby vehicle and to VEC server subject to the maximum tolerable delay and vehicle's stay time. This includes the cost of the required communication and computing resources, we consider to fully exploit the vehicular available resources, but also minimizes the overall system cost in comparison to baseline schemes.

Paper [3] discusses the trajectory prediction methods in Intelligent Transportation Systems, deep learning methods prove to improve the prediction accuracy. Therefore, STA-LSTM model is implemented which explains the influence of historical trajectories and neighboring vehicles on the target vehicle. Also, this paper compares with the already existing models.

Paper [4] considers the situation of multiple MEC servers when modeling and proposes a dynamic task offloading scheme based on deep reinforcement learning. It improves the traditional Q-Learning algorithm and combines deep learning with reinforcement learning to avoid dimensional disaster in the Q-Learning algorithm. Simulation results show that the proposed algorithm has better performance on delay, energy consumption, and total system overhead under the different number of tasks and wireless channel bandwidth.

Paper [5] focuses on the task offloading in edge computing, from task migration in multi-user scenarios and edge server resource management expansion, and proposes a multi-agent load balancing distribution based on deep reinforcement learning DTOMALB, a distributed task allocation algorithm, can perform a reasonable offload method for this scenario to improve user experience and balance resource utilization.

Paper [6] proposed 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. To improve the estimation of the long-term cost in the algorithm, we incorporate the long short-term memory (LSTM), dueling deep Q-network (DQN), and double-DQN techniques.

Paper [7] proposes a task offload strategy based on reinforcement learning computing in edge computing architecture of Internet of vehicles. Firstly, the system architecture of Internet of vehicles is designed. The Road Side Unit receives the vehicle data in community and transmits it to Mobile Edge Computing server for data analysis, while the control center collects all vehicle information. Then, the calculation model, communication model, interference model and privacy issues are constructed to ensure the rationality of task offloading in Internet of vehicles. Finally, the user cost function is minimized as objective function, and double-layer deep Q-network in deep reinforcement learning algorithm is used to solve the problem for real-time change of network state caused by user movement.

Paper [8] considers an MEC enabled multi-user multi-input multi-output (MIMO) system with stochastic wireless channels and task arrivals. In order to minimize long-term average computation cost in terms of power consumption and buffering delay at each user, a deep reinforcement learning (DRL)-based dynamic computation offloading strategy is investigated to build a scalable system with limited feedback. Specifically, a continuous action space-based DRL approach named deep deterministic policy gradient (DDPG) is adopted to learn decentralized computation offloading policies at all users respectively, where local execution and task offloading powers will be adaptively allocated according to each user’s local observation.

CHAPTER 3

METHODOLOGY

3.1 REAL TIME TASKS IN VEHICULAR EDGE COMPUTING

Some specific tasks in vehicles in real-time vehicular edge computing scenarios that need task offloading strategies:

1. Advanced driver assistance systems (ADAS): ADAS applications, such as automatic emergency braking and lane departure warning, require real-time processing of sensor data to detect and respond to hazards. These applications can be offloaded to edge servers to improve their performance and reduce the latency.
2. High-definition (HD) mapping and localization: HD maps are essential for autonomous driving and other advanced vehicle applications. However, generating and updating HD maps requires significant computational resources. These tasks can be offloaded to edge servers to free up the vehicle's onboard resources for other tasks.
3. Real-time traffic monitoring and prediction: Real-time traffic monitoring and prediction applications can help to improve traffic flow and reduce congestion. However, these applications require processing large amounts of data from vehicles and other sources. These tasks can be offloaded to edge servers to improve their performance and scalability.
4. Vehicle-to-everything (V2X) communication: V2X communication allows vehicles to communicate with each other and with infrastructure. This communication can be used for a variety of applications, such as cooperative collision avoidance and platooning. However, V2X communication can generate a significant amount of data, which can overwhelm the vehicle's onboard resources. These tasks can be offloaded to edge servers to reduce the load on the vehicle.
5. In-vehicle infotainment (IVI): IVI applications, such as navigation and entertainment, can require significant computational resources. These tasks can be offloaded to edge servers to improve the performance of IVI applications and reduce the drain on the vehicle's battery.

3.2 PROPOSED METHODOLOGY

The node will try to compute the task, within itself, with the available local resources. Once the node fails to provide the enough resource for that particular task, it choose to offload it. The RSUs with high computational resources will aid vehicles to get the output. Parameters like Local computation delay, RSU Computation Delay, Communication Delay between vehicle and RSU and the Sojourn Time of the vehicle within the RSU coverage are considered. LSTM Algorithm is used for the trajectory prediction, as it considers the position of the vehicle at the previous time instants and current time instant. Along with this, it also considers history of trajectories chosen by other vehicles gone through that rate which increases the accuracy of the prediction of the future location of the vehicle.

In this work, a typical VEC-based single-vehicle multi-cell (SVMC) scenario is considered, where a vehicle is moving on a certain section of the unidirectional highway, and a row of RSUs are deployed along the roadside. It is assumed that the coverage diameter of each RSU is L meters. Each RSU is equipped with a VEC server, providing computing and storage resources for vehicles. It is assumed that the vehicle is equipped with a global positioning system (GPS) module to obtain its own location information. The location information can be encapsulated in beacon messages that are periodically transmitted to the RSU at a fixed or variable beacon rate.

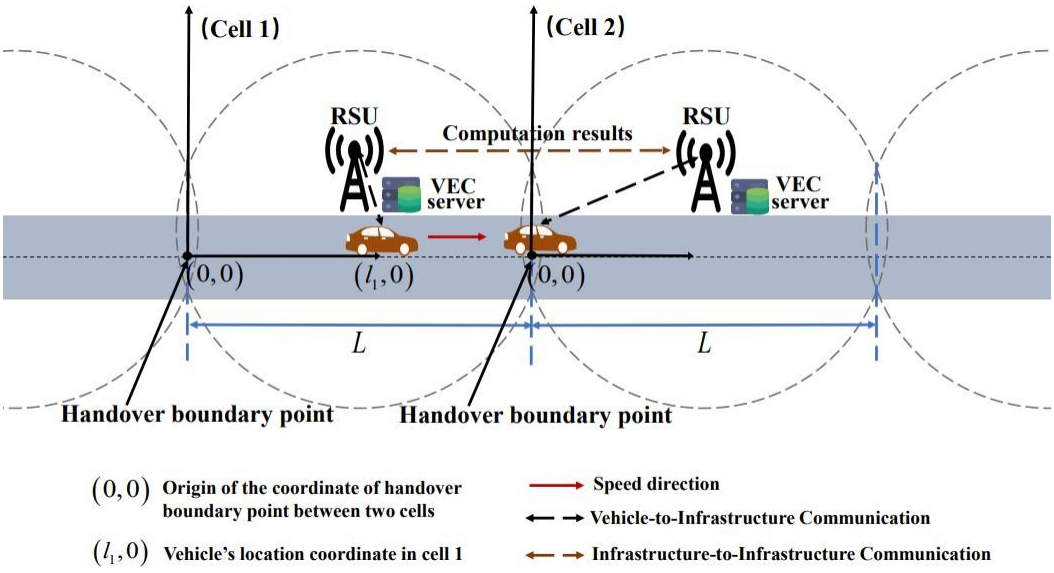

Figure 3.1 Vehicular Edge computing scenario

Assuming that the task is divisible, it can be divided evenly into multiple independent subtasks. The vehicle sends a task offloading request to the RSU. The RSU receives the vehicle status information (task description, vehicle's speed and vehicle's location) via V2I communication. The task offloading strategy is made by the VEC server according to the vehicle status information and the network status information (computing capacity of the VEC server and communication resources of the RSU). To start, the vehicle does local computation and at the same time offloads some subtasks to the VEC server for processing based on the task offloading strategy. It is assumed that the subtasks are handled one by one. Once a subtask arrives at the VEC buffer, the VEC server starts to process it immediately. Upon finishing, the computation result of the subtask is sentback to the vehicle and integrated with the local computation results.

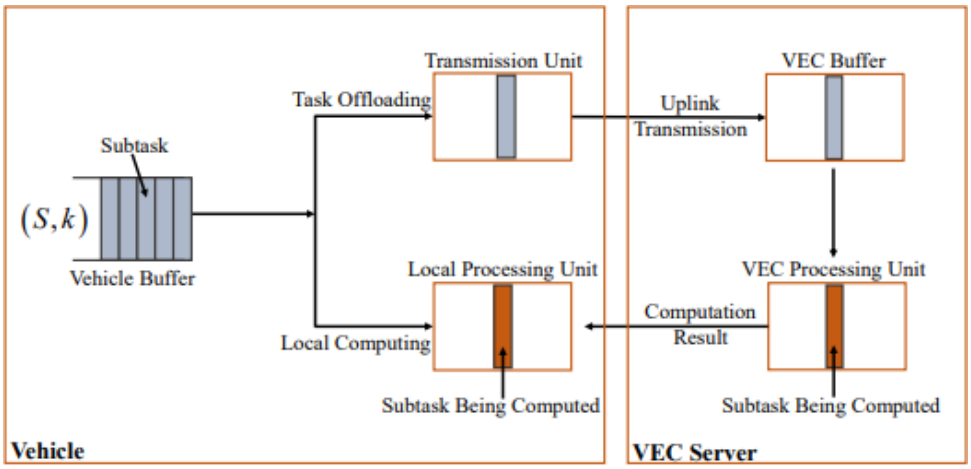

Figure 3.2 Task Offloading scenario as subtasks

3.2.1 TASK OFFLOADING DETERMINATION ALGORITHM

STEP 1:

Calculating total stay time (sojourn time) of the vehicle within RSU's coverage area:

e - vertical distance of RSU from road level, r - radius of coverage area, d_n - totaldistance that vehicle stays in RSU's coverage area.

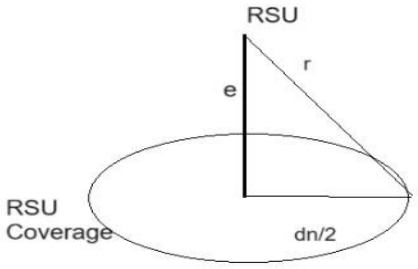

Figure 3.3 Vehicle entering the RSU coverage

Applying Pythagoras Theorem,

$$r^2 = e^2 + (d_n/2)^2$$

$$d_n = 2\sqrt{(r^2-e^2)}$$

where d_n is the total distance travelled by vehicle in RSU coverage

$$t_n, \text{ stay} = d_n / V_n \text{ [Sojourn time]}$$

where V_n - vehicle's speed.

When there is a task to be done by vehicle,

STEP 2:

If resource needed is not available in vehicle within deadline:

Task Offloading = 1

exit

Else:

Continue with next step

STEP 3 (Calculating Local Computation Delay):

- Clock frequency of local processor is identified.
- Cycles Per Instruction (CPI) required for task is calculated.

Speed at which this processor executes the task is calculated by,

MIPS = Clock frequency / (CPI* 1 million)

For a task, Local Computation Delay is calculated as,

Execution time = Instruction count / MIPS

(or)

Execution Time = Task size / Clock Frequency / (No. of clock cycles per Bit)

STEP 4 (Calculating RSU Computation Delay):

RSU Computation Delay is calculated similar to Local Computation delay calculation.

STEP 5:

If Local Computation Delay < RSU Computation Delay:

Task Offloading = 0

Exit

Else:

Continue with next step

STEP 6 (Calculating Vehicle to RSU communication delay):

$$t_{comm} = \text{Data size / Data transmission Rate}(R_{V2I})$$

where $R_{V2I} = B_{V2I} \log_2 (1 + (P_t * d_{n,rsu}^{-\sigma} |h^2|) / No)$

Pt - Tx Power

B Bandwidth

No - Noise

σ- Path loss Exponent

h- Reileigh Fading channel

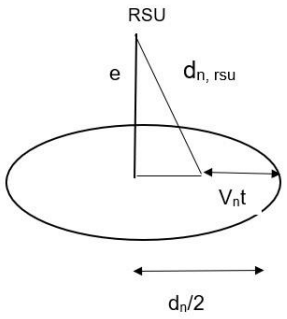

Figure 3.4 Vehicle's position after t times in the RSU coverage

where $d_{n, rsu}$ can be calculated by applying Pythagoras Theorem,

$$d_{n, rsu} = \sqrt{(e^2 + ([d_n/2] - V_n t)^2)}$$

STEP 7:

Remaining time that vehicle stays in RSU's coverage is,

$$t_{rem,\ stay} = t_{n,\ stay} - t_i$$

STEP 8:

If Vehicle to RSU communication delay > Remaining stay time of vehicle:

Task Offloading = 0

Exit

Else:

Continue with next step

STEP 9:

If Vehicle to RSU communication delay + RSU computation delay + Tolerance > Local Computation Delay:

Task Offloading = 0

exit

Else:

Task Offloading = 1

exit

Here, Tolerance is constant used to compensate the Ready Time Delay in RSU.

3.2.2 TRAJECTORY PREDICTION DETERMINATION ALGORITHM

When task is offloaded in RSU,

If RSU computation delay + RSU ready time < Deadline:

Computation is done in this RSU itself.

If Remaining vehicle stay time > 0:

Trajectory Prediction = 0

Else:

Trajectory Prediction = 1 (only for final output forwarding)

Else:

Trajectory Prediction = 1 (computation and output forwarding)

3.2.3 FLOWCHART OF PROPOSED MODEL

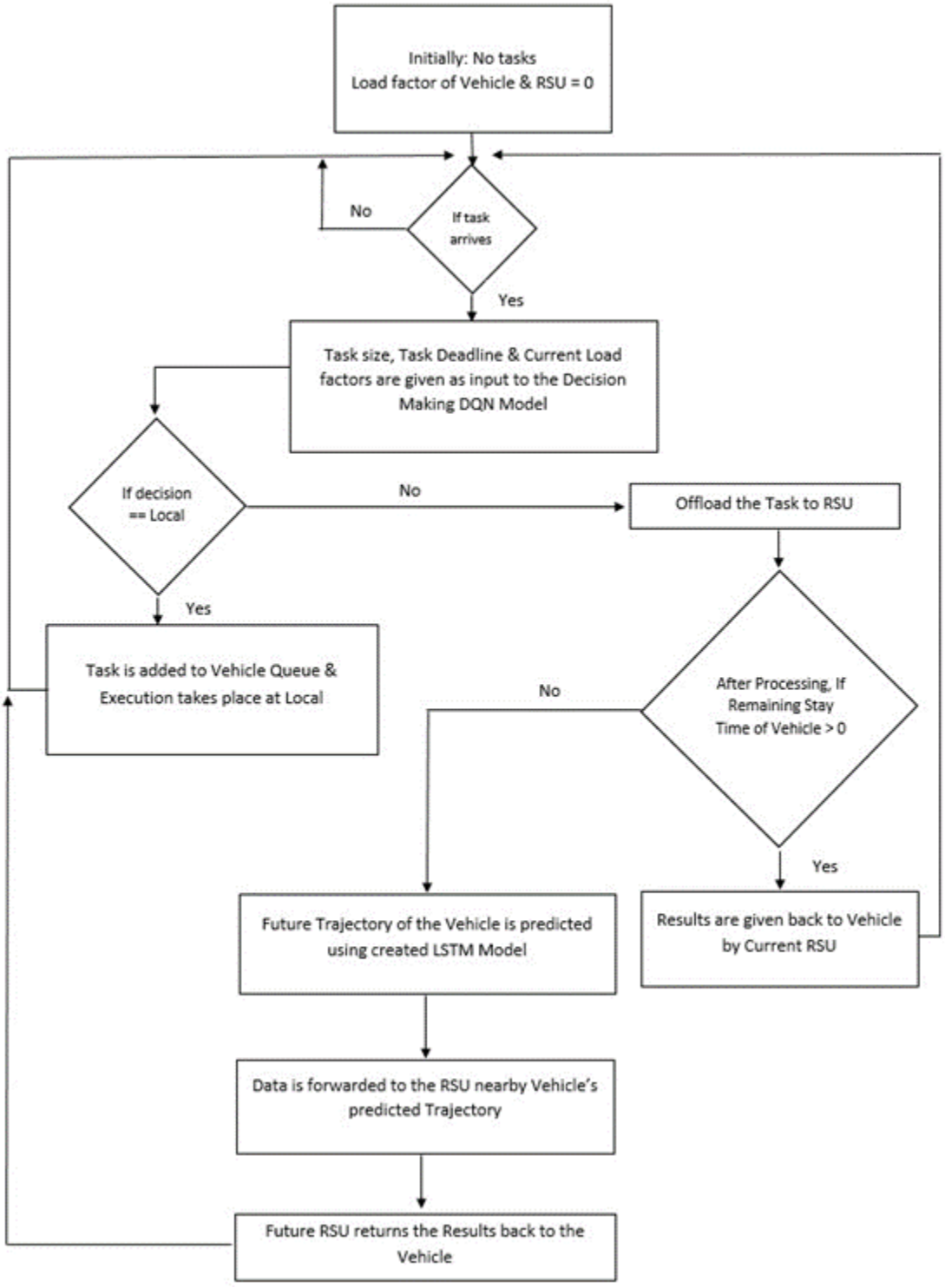

Figure 3.5 Flowchart of proposed model

3.2.4 FLOWCHART OF DQN TASK OFFLOADING DECISION MAKING MODEL

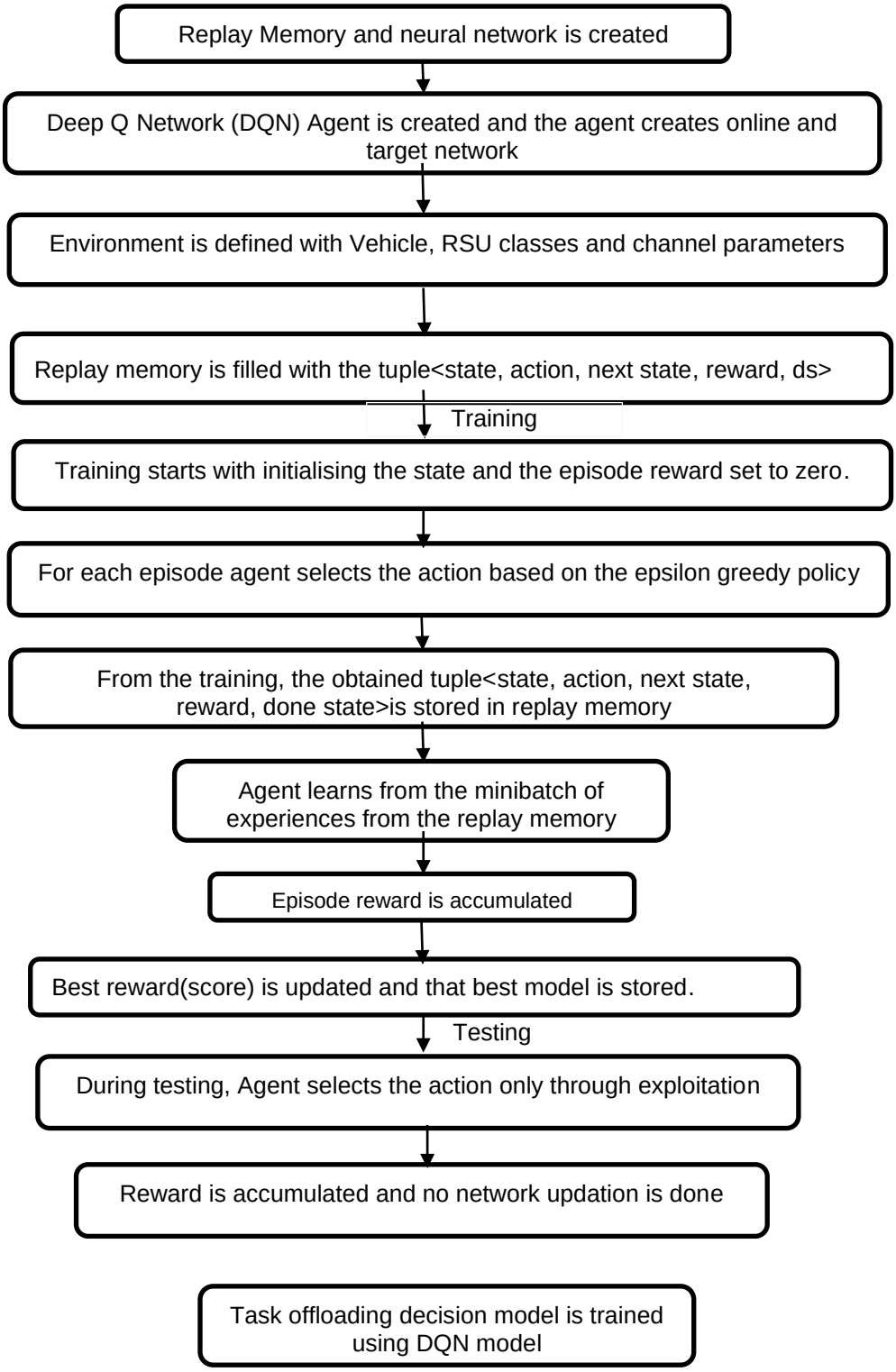

Figure 3.6 Flowchart of DQN task offloading decision making model

3.2.5 FLOWCHART OF LSTM TRAJECTORY PREDICTION MODEL

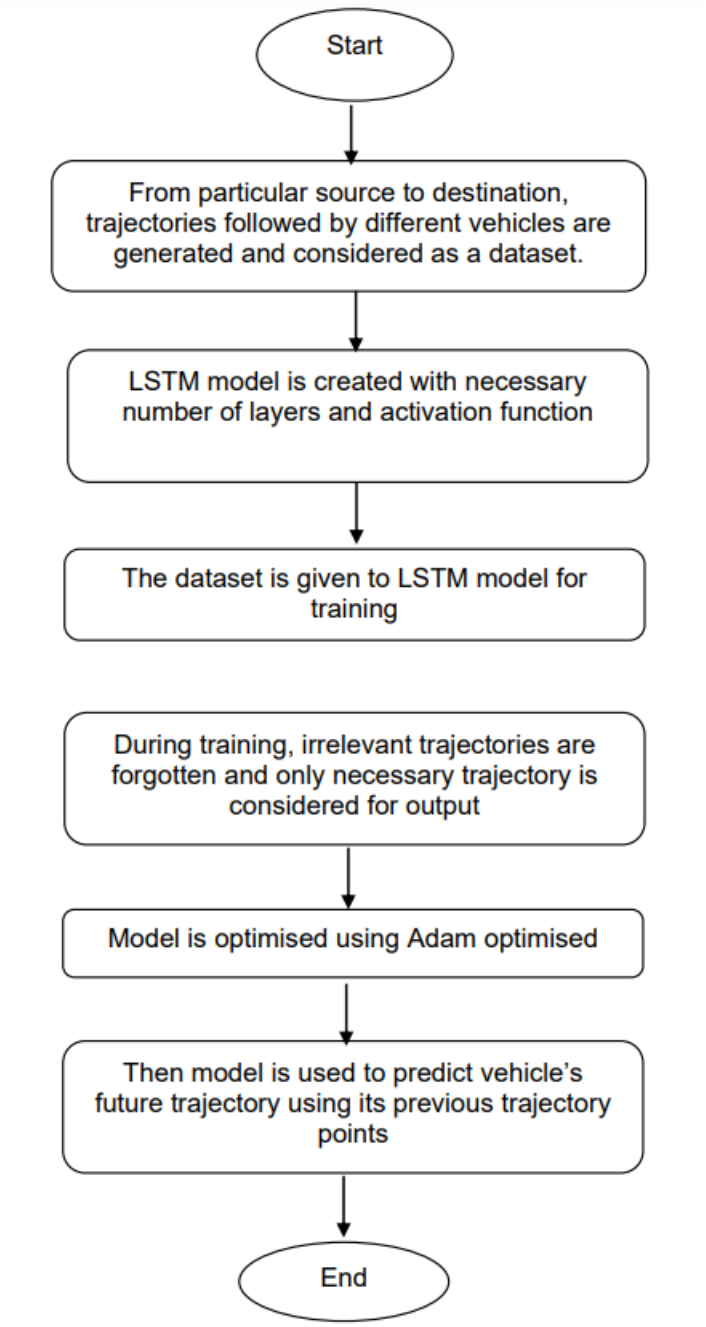

Figure 3.7 Flowchart of LSTM trajectory prediction model

CHAPTER 4

ALGORITHMS AND STANDARDS

4.1 DEEP Q NETWORK ALGORITHM

Deep Q-Network (DQN) is a powerful algorithm in reinforcement learning (RL) that combines the Q-learning algorithm with deep neural networks. Its application in RL has shown great success, especially in tasks that involve high-dimensional state spaces, such as playing video games or controlling autonomous systems. The architecture of a DQN consists of several key components. It typically includes an input layer, one or more hidden layers, and an output layer. The input layer receives the state representation, which can be raw sensory inputs like pixels or pre-processed features. The hidden layers are responsible for learning and representing the complex relationships between the input and the output. Finally, the output layer produces the Q-values for each possible action. The Q-function is the primary component of the DQN architecture. It represents the expected future rewards for each state-action pair. The neural network approximates the Q-function, taking the state as input and outputting the Q-values for all possible actions. During training, the network's weights are updated to minimize the difference between the predicted Q-values and the target Q-values. During training, the agent updates its Q-values using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha * (r + \gamma * \max(Q(s', a')) - Q(s, a))$$

Here, Q(s, a) represents the Q-value for state s and action a, r is the immediate reward, s' is the next state, a' is the next action, α is the learning rate, and γ is the discount factor.

Experience replay is another crucial component of DQN. It addresses the issue of temporal correlations between consecutive experiences in RL. Instead of training the network with experiences sequentially, DQN stores experiences (state, action, reward, next state) in a replay buffer. During training, mini-batches of experiences are sampled randomly from the replay buffer, breaking temporal correlations and providing more diverse training samples.The epsilon-greedy strategy is employed to balance exploration and exploitation. It determines whether the agent will choose a random action (exploration) or the action with the highest Q-value (exploitation). Initially, the agent explores more to gather information about the

environment, gradually reducing exploration over time to exploit the learned knowledge.

DQN addresses the problems of instability and overestimation in traditional Q-learning using two key techniques. First, the introduction of a separate target network with delayed updates helps stabilize the learning process. The target network is a copy of the main network and is used to calculate the target Q-values during training. The target network's weights are updated periodically, reducing the non-stationarity of the target values. Second, experience replay mitigates instability and overestimation by breaking the correlation between consecutive experiences. Randomly sampling mini-batches from the replay buffer provides a more diverse training dataset, making the learning process more stable and reducing the impact of noisy updates.

Here is a simplified diagram to visualize the architecture of a Deep Q-Network (DQN):

Input (State) -> Hidden Layers -> Output Layer (Q-Values)

↑

Target Q-Values

The state representation is fed into the input layer, which connects to one or more hidden layers responsible for learning and representing complex relationships. The output layer produces the Q-values for each possible action. The target Q-values, calculated using the target network, are used to update the main network's weights and guide the learning process. The training data comes from the experience replay buffer, which stores and samples experiences during training. Overall, the architecture of a DQN combines Q-learning with deep neural networks, enabling effective learning of optimal policies in RL tasks with high-dimensional state spaces.

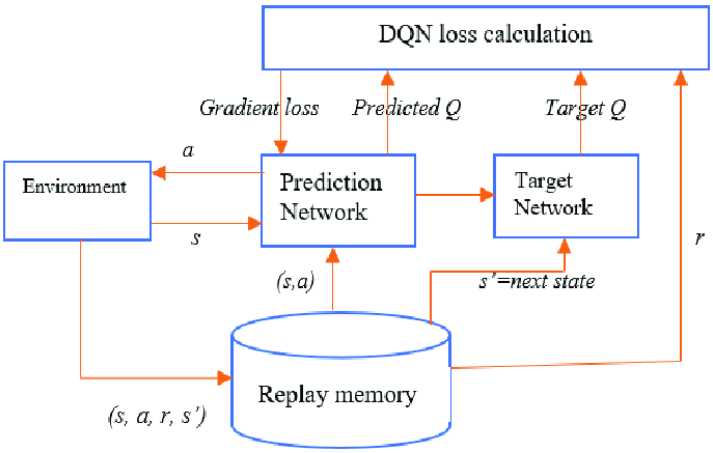

Figure 4.1 DQN working

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Figure 4.2 DQN Algorithm

4.2 LONG SHORT TERM MEMORY(LSTM) ALGORITHM

An LSTM (Long Short Term Memory) network is a type of recurrent neural network (RNN) architecture that is specifically designed to handle long-term dependencies in sequential data. Unlike traditional RNNs, LSTM networks have a unique memory cell structure that allows them to retain information over long periods, making them well-suited for tasks that involve processing and predicting sequences of data. The key difference between LSTM networks and other neural network architectures lies in their ability to handle the vanishing and exploding gradient problems. These issues arise when training RNNs on long sequences, as the gradients can become extremely small or large, hindering learning. LSTM networks overcome this problem by using a memory cell and a set of gates that regulate the flow of information, allowing for effective long-term memory storage and retrieval.

The LSTM cell comprises several key components: the input gate, the forget gate, the output gate, and the memory cell. These components work together to manage the information flow within the LSTM cell.

Input Gate: The input gate determines how much information from the current input and the previous cell state should be stored in the memory cell. It takes the current input and the previous hidden state as inputs and passes them through a sigmoid activation function, producing a value between 0 and 1 for each element in the memory cell.

Forget Gate: The forget gate controls the amount of information to be discarded from the memory cell. It takes the current input and the previous hidden state as inputs, processes them through a sigmoid activation function, and outputs a forget vector. This vector, when multiplied element-wise with the previous cell state, determines which information to forget.

Memory Cell: The memory cell stores the information over time. It takes the output from the input gate and combines it with the output from the forget gate, updating the cell state accordingly. This allows the LSTM cell to retain important information while discarding irrelevant or outdated information.

Output Gate: The output gate determines how much of the current memory cell state should be output as the hidden state of the LSTM cell. It takes the current input and the previous hidden state as inputs, passes them through a sigmoid activation function, and combines the result with the memory cell state. The output gate output is then passed through a tanh activation function to produce the current hidden state.

In simple terms, an LSTM network is a specialized type of neural network that can understand and remember patterns in sequences of data. It differs from other neural networks by having a memory cell and gates that control the flow of information. These gates determine which information to keep, forget, and output, allowing the LSTM network to retain important information over long periods. LSTM networks have various applications, including language translation, speech recognition, time series analysis, gesture recognition, music generation, and health monitoring. They are particularly useful when the order of data is important, allowing them to make predictions, classify, or generate new sequences. LSTM networks have been instrumental in advancing AI capabilities in understanding and processing sequential data, leading to significant breakthroughs in several fields.

4.3 CHANNEL MODEL – DSRC STANDARD

Dedicated Short-Range Communications (DSRC) is a suite of standards at the heart of the communication of vehicular safety messages. The fast exchanging of safety messages, combined with knowledge about other moving vehicles that may not be visible to drivers in a timely manner extend the safety concepts beyond the dreams of most of the public. Wireless Access in Vehicular Environment (WAVE) is a term used to describe the suite of IEEE P1609.x standards that are focused on MAC and network layers. WAVE is fairly complex and is built over the IEEE 802.11 standards by amending many tweaks to guarantee fast reliable exchange of safety messages. WAVE is the core part of DSRC.

In some cases, the term DSRC is used as a more general term compared to WAVE. DSRC is currently considered the most promising wireless standard that can be used to connect infrastructure (like roadside) to vehicle (I2V) and vehicle-to-vehicle (V2V). DSRC standard is based on the Wi-Fi architecture. The DSRC network is built over two basic units; Road-Side Unit (RSU) and On-Board Unit (OBU).

The RSU is, typically, a stationary unit that connects roaming vehicles to the access network, which, in turn, is connected to a larger infrastructure or to a core network. The OBU is, typically, a network device fixed in a roaming vehicle and is connected to both the DSRC wireless network and to an in-vehicle network.

The communication zone covered by each IEEE 802.11p RSU is limited to a maximum of 1 Km diameter and uses 5.9 GHz radio transmission. OBUs are expected to join the WBSS of the closest RSU, exchange information, and typically leave within limited time (mean estimate 3.6 seconds). The limited lifetime of an OBU within specific WBSS communication zone imposes hard requirements on the design of the WAVE standards suite and on the nature of future DSRC. DSRC networks use WAVE Short Messages Protocol (WSMP or WSM for short) to exchange safety information between vehicles and roadside or just between vehicles.

CHAPTER 5

IMPLEMENTATION

5.1 GENERATION OF TASK OFFLOADING DATASET

A task offloading dataset for delay-sensitive tasks involving vehicle and RSU (Roadside Unit) load factors has been generated which includes the following parameters for each task entry:

1. Task Size: The task size represents the computational complexity or the amount of computation required to complete the task. It could be measured in terms of the number of instructions, CPU cycles, or any other relevant metric that quantifies the computational workload. {5,10 Mbits}
2. Task Deadline: The task deadline refers to the maximum allowable time for completing the task. It represents the time limit within which the task results should be obtained to meet the requirements of the application or system. The deadline could be expressed in absolute time units or relative to the task initiation time. [Range – 1 to 10 s]
3. Vehicle Load Factor: The vehicle load factor indicates the current computational load or utilization level of a vehicle. It can be measured by considering factors such as CPU utilization, memory usage, or other relevant metrics that indicate the workload on the vehicle's resources. The load factor provides insights into the available capacity for executing additional tasks on the vehicle. [Range 0-10 corresponds to 0%-100%]
4. RSU Load Factor: The RSU load factor represents the current computational load or utilization level of an RSU (Roadside Unit). Similar to the vehicle load factor, it measures the workload on the RSU's resources, such as CPU utilization, memory usage, or other relevant metrics. The RSU load factor serves as an indicator of the available capacity for executing tasks offloaded from vehicles. [Range 0-10 corresponds to 0%-100%]

5.2 GENERATION OF TRAJECTORY PREDICTION DATASET

A trajectory prediction dataset is with generated x, y coordinates of a vehicle's trajectory typically captures the movement and path of a vehicle over a specific time period. It can serve as input data for various tasks like trajectory analysis, route planning, or vehicle behavior prediction. Here it is used to predict future trajectory of the vehicle for forwarding tasks to the future RSU in the vehicle's path.

Description of the key components in a trajectory dataset:

1. Vehicle ID: Each vehicle in the dataset is assigned a unique identifier. This ID serves to differentiate and track individual vehicles' trajectories.
2. Timestamp: The dataset includes timestamps that indicate the specific points in time when the x, y coordinates were recorded. These timestamps provide temporal information about the trajectory and can be used to study the vehicle's movement patterns over time.
3. X, Y Coordinates: The x and y coordinates represent the spatial positions of the vehicle at different timestamps. These coordinates can be in a specific coordinate system, such as a Cartesian coordinate system. It resembles latitude and longitude coordinates (i.e., GPS coordinates for real-world scenarios).

Using these trajectory values, LSTM model is trained and when current trajectory is given, future trajectory is being predicted.

5.3 BASIC MATHEMATICAL MODEL IMPLEMENTATION

5.3.1 PARAMETERS

- Height of the RSU : 100m
- Radius of RSU coverage : 200m
- Speed of the vehicle : 100 km/hr
- Local Computation Frequency : 1MHz
- RSU Computation Frequency : 10GHz
- Tolerance : 0.1s

5.3.2 ANALYSIS OF BASIC MATHEMATICAL MODEL IMPLEMENTATION

Case 1

```
Enter 1 if there is a task: 1
Enter total no. of instructions in task: 50000
Average CPI required: 2
Enter deadline for the task: 5
Enter the resources needed for this task: r2 r3
Enter the time needed for each resource: 3 2
This task is not offloaded, It can be done by vehicle itself
```

Figure 5.1 Lesser Task size and lesser CPI value

If both the task value and CPI value is less, then task is not offloaded, it is computed inside the vehicle itself.

Case 2

```
Enter 1 if there is a task: 1
Enter total no. of instructions in task: 50000
Average CPI required: 5
Enter deadline for the task: 7
Enter the resources needed for this task: r3 r2
Enter the time needed for each resource: 3 5
This task has been offloaded
```

Figure 5.2 Lesser Task size and higher CPI value

If the task size is less and CPI value is high, the task is offloaded to the RSU

Case 3

```
Enter 1 if there is a task: 1
Enter total no. of instructions in task: 200000
Average CPI required: 1
Enter deadline for the task: 7
Enter the resources needed for this task: r3 r2
Enter the time needed for each resource: 5 4
This task is not offloaded, It can be done by vehicle itself
```

Figure 5.3 Higher Task size and lesser CPI value

If the task size is greater and CPI is less, task is computed inside the vehicle

Case 4

```
Enter 1 if there is a task: 1
Enter total no. of instructions in task: 200000
Average CPI required: 2
Enter deadline for the task: 5
Enter the resources needed for this task: r3 r1
Enter the time needed for each resource: 3 1
This task has been offloaded
```

Figure 5.4 Higher Task size and higher CPI value

If both the task size and CPI value is greater, the task is offloaded to the RSU

After offloading the task to the RSU, the decision is made on whether to predict trajectory or not. Therefore, the remaining stay time of the vehicle inside the RSU coverage also the deadline to compute the task is compared to decide whether to predict the trajectory of the vehicle.

Case 1

```
= RESTART: C:/Users/subhashini/Desktop/subha/college/phase 1/computationcode.py
Enter 1 if there is a task: 1
Enter total no. of instructions in task: 200000
Average CPI required: 2
Enter deadline for the task: 10
Enter the resources needed for this task: r1 r2
Enter the time needed for each resource: 1 2
This task has been offloaded
Predict Trajectory
```

Figure 5.5 More deadline than stay time

From the result, if the instructions are more and deadline of the task is higher, the trajectory is predicted as the remaining stay time of the vehicle will be less than the deadline, the result or the complete task is offloaded to the future RSU.

Case 2

```
Enter 1 if there is a task: 1
Enter total no. of instructions in task: 200000
Average CPI required: 2
Enter deadline for the task: 5
Enter the resources needed for this task: r1 r2
Enter the time needed for each resource: 2 4
This task has been offloaded
Task computed in the RSU itself
```

Figure 5.6 Less deadline than stay time

If the deadline of the task is less than the remaining existing stay of the vehicle, then the task is computed in RSU and given to the vehicle in the RSU coverage.

5.4 Q-LEARNING TASK OFFLOADING DECISION MAKING MODEL

For making efficient task offloading decision making model, dynamic and complex environments is considered and to make the right decision, instead of normal decision-making approach, Q-learning (Reinforcement learning model) approach is implemented.

5.4.1 ENVIRONMENT PARAMETERS

- Height of RSU - 15m
- Radius of coverage - 200m
- Vehicle speed - 100km/hr
- Communication Bandwidth - 10MHz
- Transmission Power – [1,2] W
- Vehicle's computing capability – [10, 8, 4, 2] MHz
- RSU's computing capability – [30, 20] MHz

5.4.2 SYSTEM PARAMETERS

- State Space – [Vehicle's clock frequency, RSU's clock frequency]
- Action Space – [local, offload]
- Reward Function -

$$C_{all} = \theta T_{all} + (1 - \theta)\, E_{all}$$

where,

T_{all} - Total time taken for the action taken

E_{all} - Total energy taken for the action taken

Θ - optimization parameter

Model has to be trained to find minimum C_{all} value to get maximum reward.

5.4.3 TRAINING PARAMETERS

- Episodes - 1000
- Epsilon - 0.2
- Discount Factor - 0.9
- Learning rate - 0.9

5.4.4 ANALYSIS OF Q-LEARNING MODEL IMPLEMENTATION

In get_decision function there are two arguments passed. First argument is index of clock frequency of the vehicle list. Second argument is the index of clock frequency of the RSU list.

Figure 5.7 With respect to 30MHz

With respect to 30MHz RSU and for the vehicles with 10MHz, 8MHz, 4MHz and 2MHz clock frequency, the respective decision made by Q-learning model is shown.

Figure 5.8 With respect to 20MHz

With respect to 20MHz RSU and for the vehicles with 10MHz, 8MHz, 4MHz and 2MHz clock frequency, the respective decision made by Q-learning model is shown.

By comparing both the cases, it can be inferred that the decision is made as offload in the case of 4MHz vehicle and 30MHz RSU whereas the decision is made as local in the case of 4MHz vehicle and 20MHz RSU. Hence it is understood that the balancing between vehicle and RSU is needed.

5.5 DQN TASK OFFLOADING DECISION MAKING MODEL IMPLEMENTATION

To improve the efficiency in task offloading decision making strategy, load factor of both vehicle and RSU is considered in this approach and instead of Q-learning model, DQN model is implemented. DQN approach is used since it can accept high-dimensional inputs and it has discrete action space.

5.5.1 ENVIRONMENT PARAMETERS

RSU

- Clock Frequency = 10 MHz
- Height = 8 m
- Radius = 200 m

Vehicle

- Clock frequency = 2 MHz
- Speed of the vehicle = 60km/hr

Channel

- Bandwidth = 10 MHz (By IEEE 802.11p standard protocol - WAVE)
  (DSRC - 5.85-5.925 GHz, 7 channels having 10MHz each)

5.5.2 SYSTEM PARAMETERS

- State Space : {Task size, Task Deadline, Loadfactor}
- Action Space : {Local, Offload}
- Reward : - (delay + loadfactor)

5.5.3 TRAINING PARAMETERS

- Training episodes : 1000
- Timesteps : 10
- Discount factor : 0.99
- Epsilon_minimum : 0.01
- Epsilon_maximum : 1
- Epsilon_decay value : 0.995
- Memory capacity : 5000 units
- Minibatch size : 64
- Testing episodes : 100

5.5.4 ANALYSIS OF DQN MODEL HYPER PARAMETERS

Case 1: Learning Rate : 0.001 and No of neurons = 32

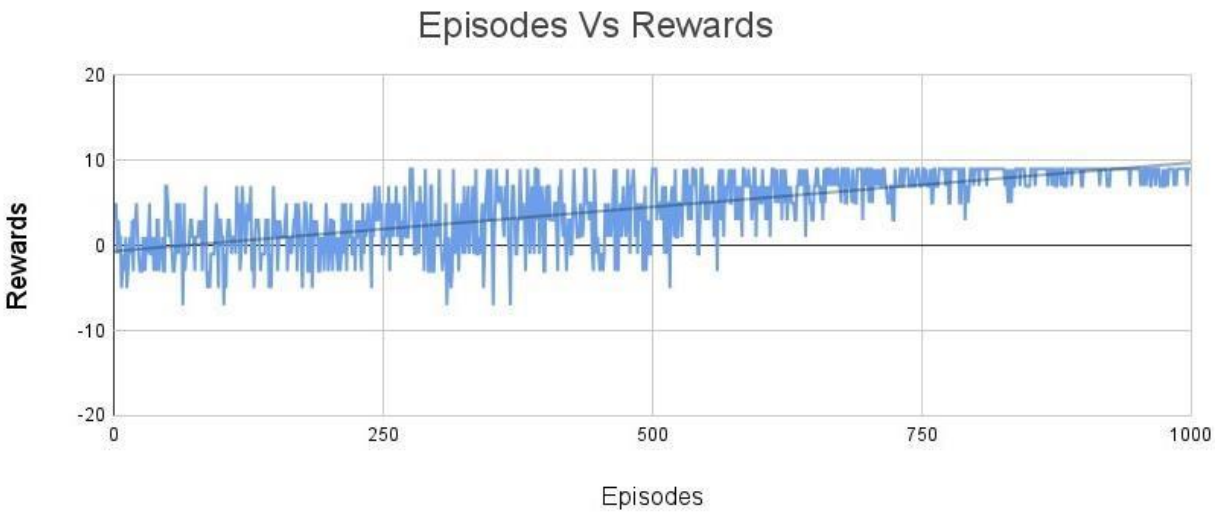

Figure 5.9 Episodes vs Rewards (Learning Rate - 0.001)

At learning rate 0.001 convergence starts at episode no. 750

Case 2 : Learning Rate : 0.01 , one hidden layer and No of neurons = 32

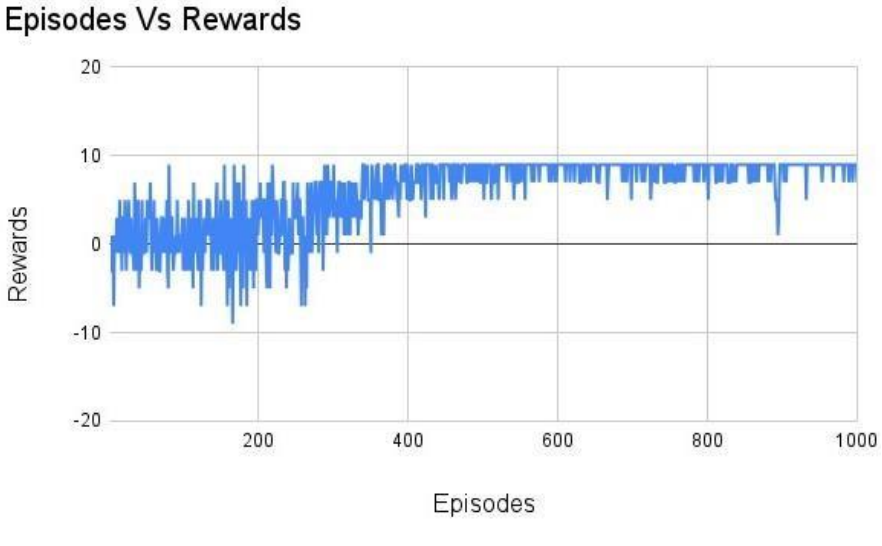

Figure 5.10 Episodes vs Rewards (Learning Rate - 0.01)

At learning rate 0.01 convergence starts at episode no. 400

Case 3 : No of neurons = 64

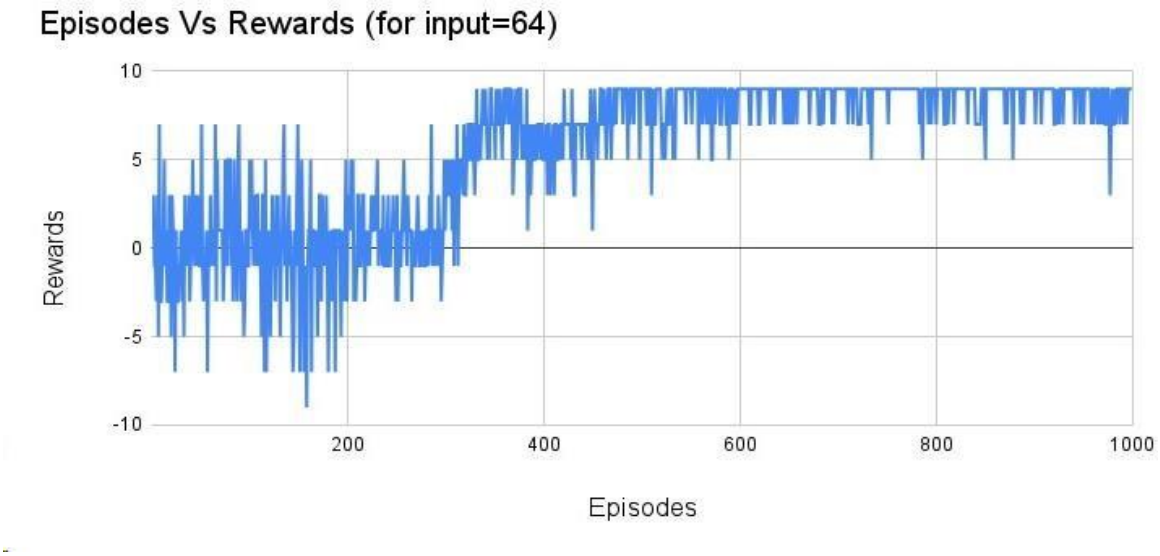

Figure 5.11 Episodes vs Rewards (No. of neurons - 64)

At learning rate = 0.01 when no of neurons is 64 the convergence starts at episode no. 600

Case 4 : No of neurons = 16

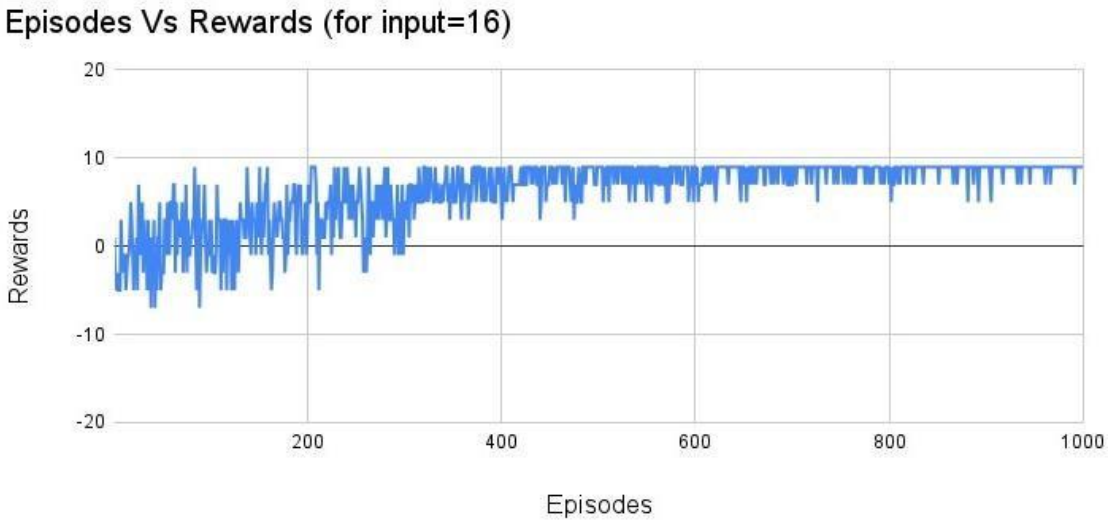

Figure 5.12 Episodes vs Rewards (No. of neurons - 16)

At learning rate = 0.01 when no of neurons is 16 the convergence starts at episode no. 300

Case 5 : No. of hidden layers = 0

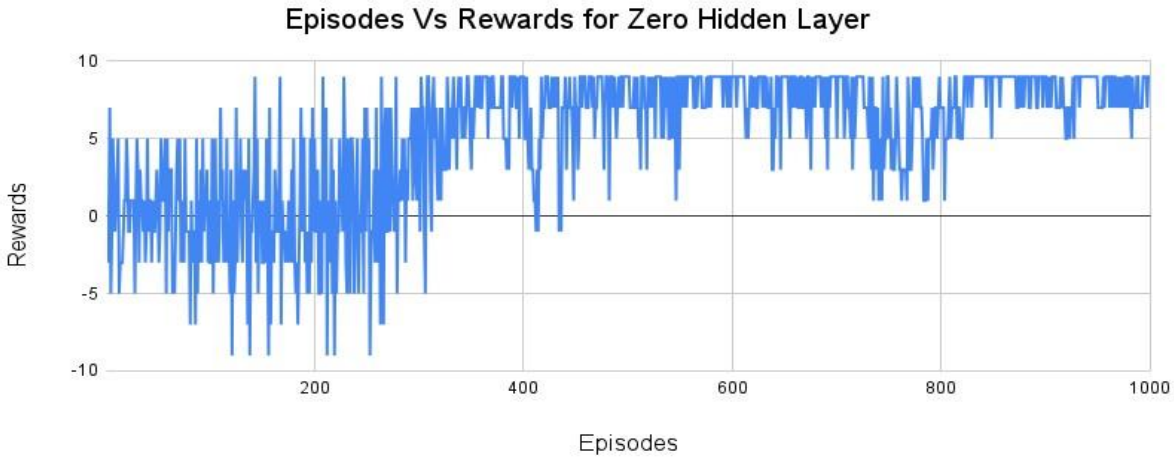

Figure 5.13 Episodes vs Rewards (No. of hidden layers - 0)

When no of hidden layers is 0, convergence happens at more than 1000 episodes and more oscillations in the reward values.

Case 6: No of hidden layers = 2

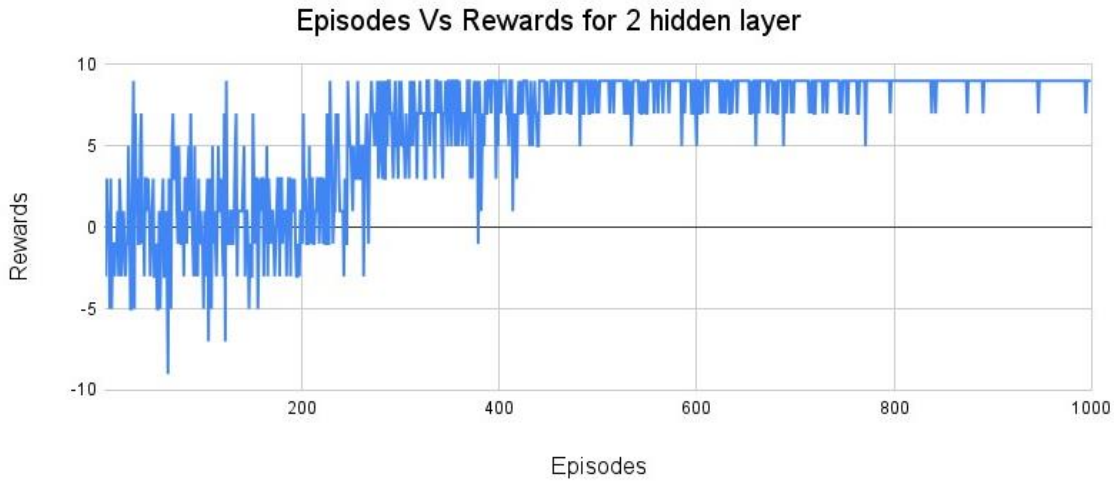

Figure 5.14 Episodes vs Rewards (No. of hidden layers - 2)

When no of hidden layers is 2, the convergence starts at episode no. 600

Case 7: No of hidden layers = 3

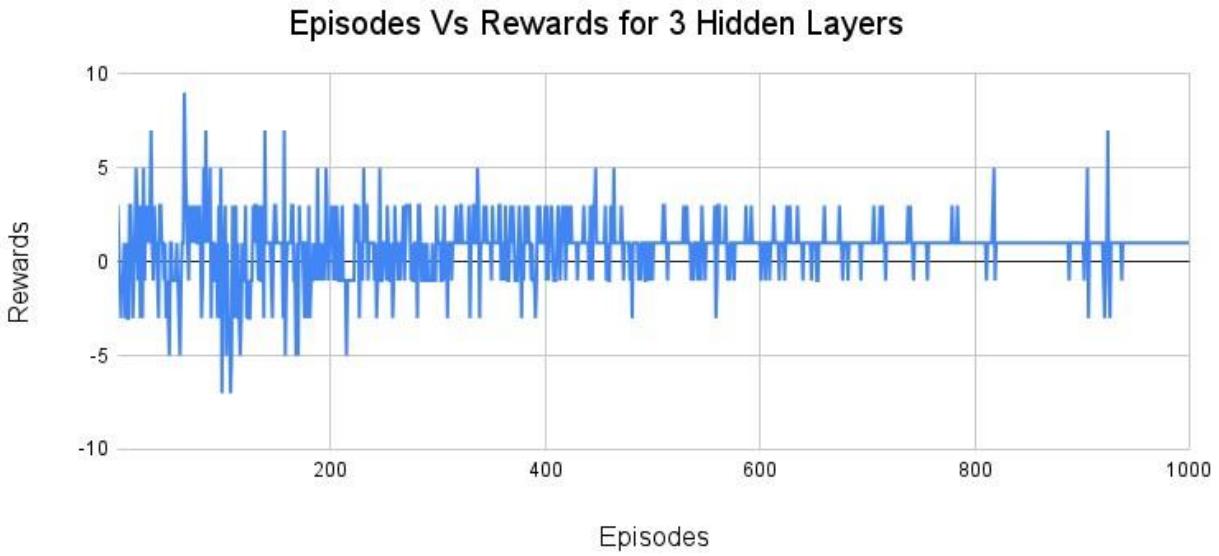

Figure 5.15 Episodes vs Rewards (No. of hidden layers - 3)

When no of hidden layers is 3, overfitting occurs

Learning Rate vs Convergence

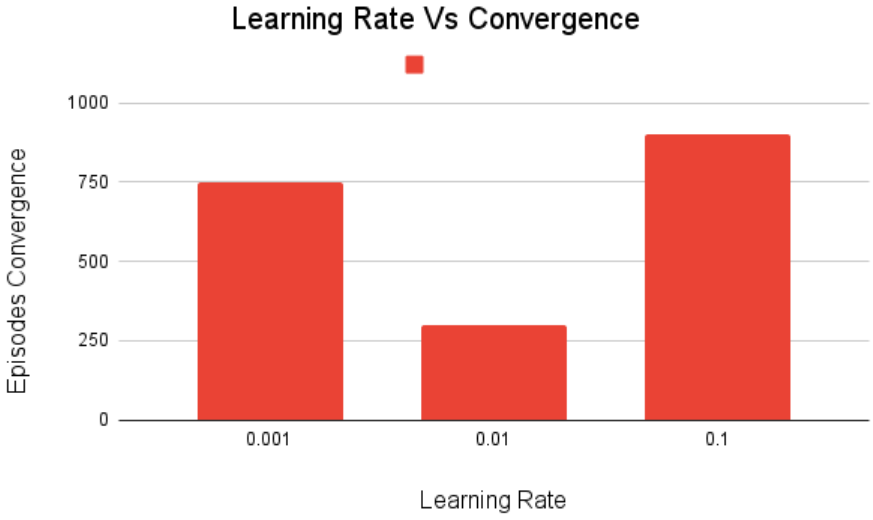

Figure 5.16 Learning Rate vs Convergence

By changing the learning rates as 0.001, 0.01 and 0.1, it is identified that the learning rate 0.01 gives the optimised output with convergence at episode no. 300.

Number of Neurons vs Convergence

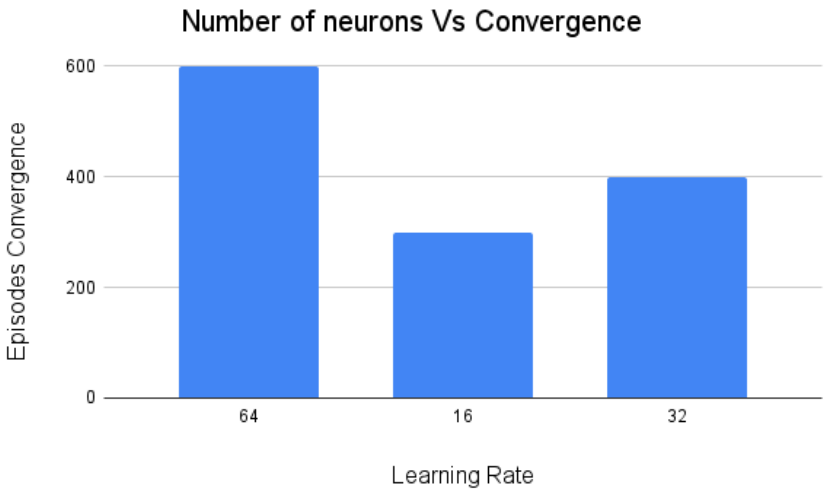

Figure 5.17 Number of Neurons vs Convergence

By fixing the learning rate as 0.01 and varying the no. of neurons as 16, 32 and 64, it is found that with 16 number of neurons in each layer, we can able to obtain the optimised model.

Number of Hidden Layers vs Convergence

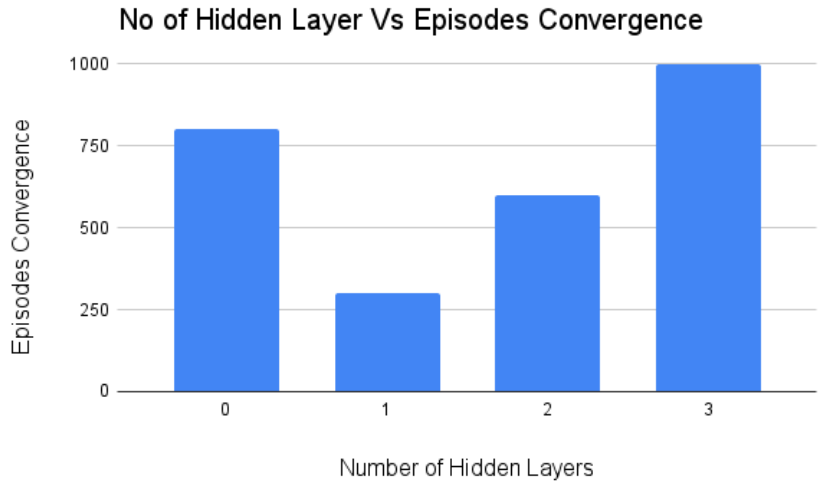

Figure 5.18 Number of hidden layers vs Convergence

By fixing the learning rate as 0.01 and number of neurons in each layer as 16, we varied the number of hidden layers from 0 to 3. It is found that with 1 hidden layer, we got the optimised model.

5.6 IMPROVED DQN MODEL IMPLEMENTATION

5.6.1 ENVIRONMENT PARAMETERS

RSU

- Clock Frequency = 1 GHz
- Height = 5 meters
- Radius = 200 meters
- Power = 2 watts

Vehicle

- Clock Frequency = 500 MHz
- Speed = 60 km/hr
- Power = 1.5 watts

Channel

- Bandwidth = 10 MHz
- Path loss exponent = -4
- Channel Fading Gain = 0.5
- Noise power = -70 dB

5.6.2 SYSTEM PARAMETERS

- State space : [Task size, Task Deadline, Vehicle Load Factor, RSU Load Factor]
- Action space : [0,1] #0 - local #1 - offload
- Reward function :

  Reward = - w1 * (Total Delay) - w2 * (Load Factor Variance)

where,

Total Delay = (1 - action) * Local Delay + Action * RSU Delay

Load Factor = (1 - action) * Vehicle Load Factor + Action * RSU Load Factor

Load Factor Variance = (Load Factor – |Mean of Load Factor|)^2 / M

where M – Total No. of Nodes in Environment

5.6.3 TRAINING PARAMETERS

- Training episodes : 1000
- Timesteps : 1400
- Discount factor : 0.99
- Epsilon_minimum : 0.01
- Epsilon_maximum : 1
- Epsilon_decay value : 0.995
- Memory capacity : 5000 units
- Minibatch size : 64
- Testing episodes : 100

5.6.4 HYPER PARAMETERS

- Learning Rate – 0.001
- No of hidden layers - 3
- No of neurons – [64, 128, 32]
- Activation function – [relu, relu, relu]

5.6.5 EPISODES vs REWARDS

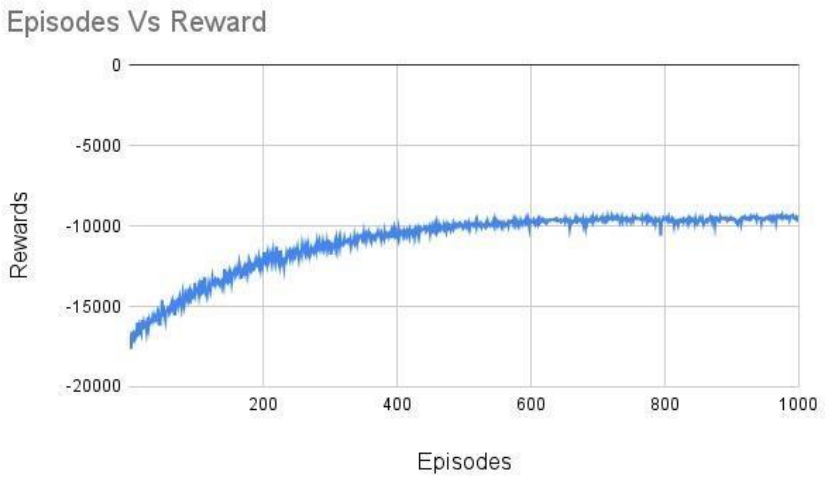

Figure 5.19 Episodes vs Rewards

It is inferred that as episode increases, reward increases. It took almost 6 hours for training and most of the complex cases gets trained well. The convergence happens after 600 episodes.

5.7 LSTM TRAJECTORY PREDICTION MODEL IMPLEMENTATION

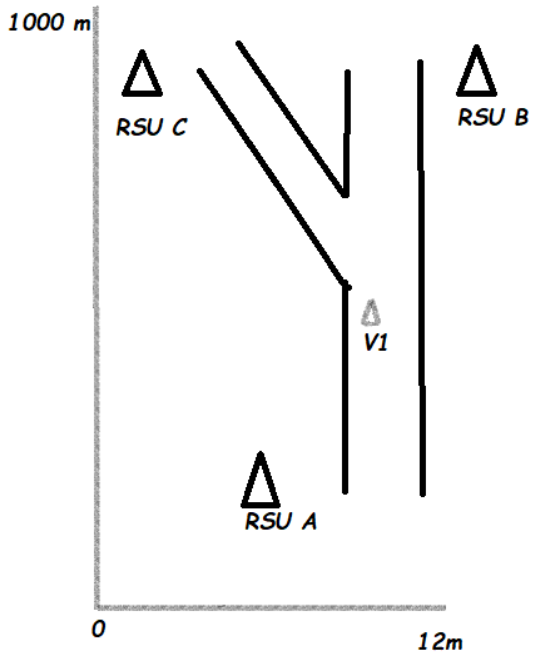

Figure 5.20 Trajectory prediction scenario

In environment, two different paths for the same destination are considered. First path is straight lane and second path cuts after 500km of the straight path. One vehicle, Current RSU and two possibilities of future RSUs are considered in the environment. Dataset is generated by collecting X, Y coordinates for 60 timesteps (i.e, for a minute). It is generated for 10 different vehicles which chooses either first path or second path for the same destination.

Now if the vehicle crosses the RSU A coverage before the results are returned, then there is a need for RSU to forward the results to future RSU. That future RSU can be either RSU B or RSU C depending on the way through which vehicle goes. Hence in order to predict the future trajectory of the vehicle, RSU collects previous trajectories of the vehicle and gives as input to the trained LSTM model. Then the model gives the predicted future trajectory based on the route chosen by most number of vehicles for the same destination.

5.7.1 PARAMETERS

- No. of LSTM layers – 2
- No. of neurons – 64
- Optimizer – ADAM
- Loss - Binary cross entropy

CHAPTER 6

RESULTS

6.1 DQN TASK OFFLOADING DECISION MAKING MODEL RESULTS

Figure 6.1 DQN results

The different states have been given to DQN agent which chooses the best action by using the trained model. The agent selects the efficient action which reduces the task failure and as well as balancing the load at each node in the environment.

6.2 LSTM TRAJECTORY PREDICTION MODEL RESULTS

```
enter the current trajectory: [(6.87,340),(7.2,356),(8.6,432)]
the future point is: 9.2 , 448
the vehicle goes through RSU B from RSU A
```

Figure 6.2 RSU A to RSU B scenario

It shows that the most vehicles have gone through the route nearby RSU B (i.e, through straight lane) when going through these current trajectory points.

```
enter the current trajectory: [(4.87,540),(3.2,556),(2.6,582)]
the future point is: 2.4 , 616
the vehicle goes through RSU C from RSU A
```

Figure 6.3 RSU A to RSU C scenario

It shows that the most vehicles have gone through the route nearby RSU C (i.e, through different lane) when going through these current trajectory points.

6.3 RESULTS OF COMPLETE MODEL

Both DQN Task offloading decision making model and LSTM Trajectory Prediction model are integrated together to show how really task offloading model is working with these trained models.

11 different states were given to the model and checked how it’s behaving in each state. The results are as follows.

Vehicle buffer initially empty

RSU buffer initially empty

Task no: 1

Task size 5

Task Deadline 5

Task values have been given to Task Offloading Decision Model

This task is done at local

Vehicle's Buffer - [5]

Vehicle's Loadfactor - 12.5 %

RSU's Buffer - []

RSU's Loadfactor - 0 %

remaining stay time 31.85

Task no: 2

Task size 5

Task Deadline 3

Task values have been given to Task Offloading Decision Model

This task is done at local

Vehicle's Buffer - [5, 5]

Vehicle's Loadfactor - 25.0 %

RSU's Buffer - []

RSU's Loadfactor - 0 %

remaining stay time 28.700000000000003

Task no: 3

Task size 10

Task Deadline 9

Task values have been given to Task Offloading Decision Model

This task has been offloaded

Vehicle's Buffer - [5, 5]

Vehicle's Loadfactor - 25.0 %

RSU's Buffer - [10]

RSU's Loadfactor - 14.285714285714285 %

remaining stay time 25.550000000000004

Task no: 4

Task size 10

Task Deadline 5

Task values have been given to Task Offloading Decision Model

This task has been offloaded

Vehicle's Buffer - [5, 5]

Vehicle's Loadfactor - 25.0 %

RSU's Buffer - [10, 10]

RSU's Loadfactor - 28.57142857142857 %

remaining stay time 22.400000000000006

Task no: 5

Task size 5

Task Deadline 2.5

Task values have been given to Task Offloading Decision Model

This task has been offloaded

Vehicle's Buffer - [5, 5]

Vehicle's Loadfactor - 25.0 %

RSU's Buffer - [10, 10, 5]

RSU's Loadfactor - 35.714285714285715 %

remaining stay time 19.250000000000007

Task no: 6

Task size 10

Task Deadline 12

Task values have been given to Task Offloading Decision Model

This task is done at local

Vehicle's Buffer - [5, 5, 10]

Vehicle's Loadfactor - 50.0 %

RSU's Buffer - [10, 10, 5]

RSU's Loadfactor - 35.714285714285715 %

remaining stay time 16.10000000000001

Task no: 7

Task size 10

Task Deadline 9

Task values have been given to Task Offloading Decision Model

This task has been offloaded

Vehicle's Buffer - [5, 5, 10]

Vehicle's Loadfactor - 50.0 %

RSU's Buffer - [10, 10, 5, 10]

RSU's Loadfactor - 50.0 %

remaining stay time 12.950000000000008

Task no: 8

Task size 5

Task Deadline 8

Task values have been given to Task Offloading Decision Model

This task is done at local

Vehicle's Buffer - [5, 5, 10, 5]

Vehicle's Loadfactor - 62.5 %

RSU's Buffer - [10, 10, 5, 10]

RSU's Loadfactor - 50.0 %

remaining stay time 9.800000000000008

Task no: 9

Task size 10

Task Deadline 15

Task values have been given to Task Offloading Decision Model

This task is done at local

Vehicle's Buffer - [5, 5, 10, 5, 10]

Vehicle's Loadfactor - 87.5 %

RSU's Buffer - [10, 10, 5, 10]

RSU's Loadfactor - 50.0 %

remaining stay time 6.6500000000000075

Task no: 10

Task size 5

Task Deadline 7

Task values have been given to Task Offloading Decision Model

This task has been offloaded

Vehicle's Buffer - [5, 5, 10, 5, 10]

Vehicle's Loadfactor - 87.5 %

RSU's Buffer - [10, 10, 5, 10, 5]

RSU's Loadfactor - 57.14285714285714 %

remaining stay time 3.5000000000000075

Remaining Stay Time of the vehicle is lesser than the execution time. So Results cannot be returned to Vehicle by Current RSU

Hence Trajectory Prediction is done to predict the future RSU
Trajectory is predicted and task is being forwarded to RSU-B

Task no: 11
Task size 10
Task Deadline 7
Task values have been given to Task Offloading Decision Model
This task has been offloaded
Vehicle's Buffer - [5, 5, 10, 5, 10]
Vehicle's Loadfactor - 87.5 %
RSU's Buffer - [10, 10, 5, 10, 5, 10]
RSU's Loadfactor - 71.42857142857143 %
remaining stay time 0.35000000000000764
Remaining Stay Time of the vehicle is lesser than the execution time. So Results cannot be returned to Vehicle by Current RSU
Hence Trajectory Prediction is done to predict the future RSU
Trajectory is predicted and task is being forwarded to RSU-B

For important cases, explanation is as follows,

CASE 1 : Local Computation
Initially both the vehicle & RSU buffer were empty.
Vehicle's load factor is at minimum, so any task can be done by vehicle itself

```
Vehicle buffer initially empty
RSU buffer initially empty
Task no: 1
Task size 5
Task Deadline 5
Task values have been given to Task Offloading Decision Model
This task is done at local
Vehicle's Buffer -  [5]
Vehicle's Loadfactor -  12.5 %
RSU's Buffer -  []
RSU's Loadfactor -  0 %
remaining stay time 31.85
```

Figure 6.4 Local computing scenario 1

CASE 2 : Task Offloading

Even though there is a less vehicle load factor, we need to offload the task.

Since local execution time > task deadline, this task can't be computed locally.

```
Task no: 4
Task size 10
Task Deadline 5
Task values have been given to Task Offloading Decision Model
This task has been offloaded
Vehicle's Buffer -  [5, 5]
Vehicle's Loadfactor -  25.0 %
RSU's Buffer -  [10, 10]
RSU's Loadfactor -  28.57142857142857 %
remaining stay time 22.400000000000006
```

Figure 6.5 Offloading scenario 1

CASE 3 : Local Computation

Though Vehicle's Load factor > RSU's Load factor

Task can be locally computed, since there is enough resource and time for its computation at vehicle itself.

```
Task no: 6
Task size 10
Task Deadline 12
Task values have been given to Task Offloading Decision Model
This task is done at local
Vehicle's Buffer -  [5, 5, 10]
Vehicle's Loadfactor -  50.0 %
RSU's Buffer -  [10, 10, 5]
RSU's Loadfactor -  35.714285714285715 %
remaining stay time 16.10000000000001
```

Figure 6.6 Local computing scenario 2

CASE 4 : Task Offloading

In order to balance the load factor of vehicle and RSU , the task has to be offloaded.

```
Task no: 7
Task size 10
Task Deadline 9
Task values have been given to Task Offloading Decision Model
This task has been offloaded
Vehicle's Buffer -  [5, 5, 10]
Vehicle's Loadfactor -  50.0 %
RSU's Buffer -  [10, 10, 5, 10]
RSU's Loadfactor -  50.0 %
remaining stay time 12.950000000000008
```

Figure 6.7 Load balancing scenario

CASE 5 : Local Computation

Vehicle’s Load factor > RSU’s Load factor

Even though there is less available resource, enough deadline time makes the vehicle to do the local computation for this task.

```
Task no: 9
Task size 10
Task Deadline 15
Task values have been given to Task Offloading Decision Model
This task is done at local
Vehicle's Buffer -  [5, 5, 10, 5, 10]
Vehicle's Loadfactor -  87.5 %
RSU's Buffer -  [10, 10, 5, 10]
RSU's Loadfactor -  50.0 %
remaining stay time 6.6500000000000075
```

Figure 6.8 Offloading scenario 2

CASE 6 : Task offloading and Trajectory Prediction

Remaining time of the vehicle within the RSU's coverage is very less.

So, Future trajectory is predicted

Then, the results are delivered through future RSU.

```
Task no: 11
Task size 10
Task Deadline 7
Task values have been given to Task Offloading Decision Model
This task has been offloaded
Vehicle's Buffer -  [5, 5, 10, 5, 10]
Vehicle's Loadfactor -  87.5 %
RSU's Buffer -  [10, 10, 5, 10, 5, 10]
RSU's Loadfactor -  71.42857142857143 %
remaining stay time 0.35000000000000764
Remaining Stay Time of the vehicle is lesser than the execution time. So Results cannot be returned to Vehicle by Current RSU
Hence Trajectory Prediction is done to predict the future RSU
Trajectory is predicted and task is being forwarded to RSU-B
```

Figure 6.9 Trajectory prediction scenario

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

The task offloading scheme of this project aims at low latency in the context of vehicular networking. To address the problem that the traditional task offloading scheme does not consider the high-speed movement of nodes, this task offloading scheme based on trajectory prediction, which fully considers the high-speed movement of vehicles and solves the problem that the task calculation results cannot be returned within the communication time between vehicles and base stations. The superiority of this scheme is proved through extensive experiments. In the highway context, it is ensured that this task offloading can be performed properly and shows stronger stability when facing a large number of computational tasks.

However, since the trajectory prediction only considers the characteristics of the forward motion of the vehicle, and does not consider the turning and U-turn of the vehicle, the change algorithm is not applicable to urban roads anymore. In future work, we will continue to improve the system framework and expand the application scenario of the system to urban roads by combining the influence of pedestrians and non-motorized vehicles when performing trajectory prediction. On urban roads, the traffic flow is more concentrated, different vehicles have different computational tasks of their own, the movement of vehicles is relatively slow, and the communication links become closer. When vehicles are offloading tasks, in addition to offloading to RSU, offloading tasks to surrounding idle vehicles is also a future research direction.

7.2 FUTURE WORK

Instead of DSRC standard, 5G can be used for channel modelling to make it more feasible. Priority based scheduling can be implemented instead of First come First serve to make the model more reliable in the real-life scenario. The advancement of mobile RSUs can be use by replacing static RSUs.

BIBLIOGRAPHY

[1] Jiachen Zeng, Fangfang Gou, Jia Wu," Task offloading scheme combining deep reinforcement learning and convolutional neural networks for vehicle trajectory prediction in smart cities", *Computer Communications*, Volume 208,2023, Pages 29-43, ISSN0140-3664.

[2] Raza, S., Liu, W., Ahmed, M. et al. "An efficient task offloading scheme in vehicular edge computing", *Journal of Cloud Computing: Advances, Systems and Applications* 9:28 (2020).

[3] L. Lin, W. Li, H. Bi and L. Qin, "Vehicle Trajectory Prediction Using LSTMs With Spatial–Temporal Attention Mechanisms," in *IEEE Intelligent Transportation Systems Magazine*, vol. 14, no. 2, pp. 197-208, March-April 2022, doi: 10.1109/MITS.2021.3049404.

[4] Zhang, D., Cao, L., Zhu, H. et al, "Task offloading method of edge computing in internet of vehicles based on deep reinforcement learning", *Cluster Comput* 25, 1175–1187 (2022).

[5] Zhang, Z., Li, C., Peng, S. et al.," A new task offloading algorithm in edge computing", *J Wireless Com Network 2021*, 17 (2021).

[6] M. Tang and V. W. S. Wong, "Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems," *in IEEE Transactions on Mobile Computing*, vol. 21, no. 6, pp. 1985-1997, 1 June 2022, doi: 10.1109/TMC.2020.3036871.

[7] K. Wang, X. Wang, X. Liu and A. Jolfaei, "Task Offloading Strategy Based on Reinforcement Learning Computing in Edge Computing Architecture of Internet of Vehicles," *in IEEE Access*, vol. 8, pp. 173779-173789, 2020, doi: 10.1109/ACCESS.2020.3023939.

[8] Chen, Z., Wang, X., "Decentralized computation offloading for multi-user mobile edge computing: a deep reinforcement learning approach", *J Wireless Com Network 2020*, 188 (2020).

APPENDICES

PYTHON CODE FOR DQN TASK OFFLOADING DECISION MAKING MODEL

```
import random
import numpy as np
from scipy.integrate import quad
import math
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
m = 10**6
k = 10**3
g = 10**9

class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer_state = []
        self.buffer_action = []
        self.buffer_next_state = []
        self.buffer_reward = []
        self.buffer_done = []
        self.idx = 0
    def store(self, state, action, next_state, reward, done):
        if len(self.buffer_state) < self.capacity:
            self.buffer_state.append(state)
            self.buffer_action.append(action)
            self.buffer_next_state.append(next_state)
            self.buffer_reward.append(reward)
            self.buffer_done.append(done)
        else:
            self.buffer_state[self.idx] = state
            self.buffer_action[self.idx] = action
            self.buffer_next_state[self.idx] = next_state
            self.buffer_reward[self.idx] = reward
            self.buffer_done[self.idx] = done
        self.idx = (self.idx+1)%self.capacity
    def sample(self, batch_size, device):
        indices_to_sample = random.sample(range(len(self.buffer_state)), batch_size)
        states = torch.from_numpy(np.array(self.buffer_state)[indices_to_sample]).float().to(device)
        actions = torch.from_numpy(np.array(self.buffer_action)[indices_to_sample]).to(device)
```

```
        next_states = torch.from_numpy(np.array(self.buffer_next_state)[indices_to_sample]).float().to(device)
        rewards = torch.from_numpy(np.array(self.buffer_reward)[indices_to_sample]).float().to(device)
        dones = torch.from_numpy(np.array(self.buffer_done)[indices_to_sample]).to(device)
        return states, actions, next_states, rewards, dones
    def __len__(self):
        return len(self.buffer_state)

class DQNNet(nn.Module):
    def __init__(self, input_size, output_size, lr=1e-3):
        super(DQNNet, self).__init__()
        self.dense1 = nn.Linear(input_size, 64)
        self.dense2 = nn.Linear(64, 128)
        self.dense3 = nn.Linear(128, 64)
        self.dense4 = nn.Linear(64, 32)
        self.dense5 = nn.Linear(32, output_size)
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
    def forward(self, x):
        x = F.relu(self.dense1(x))
        x = F.relu(self.dense2(x))
        x = F.relu(self.dense3(x))
        x = F.relu(self.dense4(x))
        x = (self.dense5(x))
        return x
    def save_model(self, filename):
        torch.save(self.state_dict(), filename)
    def load_model(self, filename, device):
        self.load_state_dict(torch.load(filename, map_location=device))

class DQNAgent:
    def __init__(self, device, state_size, action_size,discount=0.99,eps_max=1.0,eps_min=0.01,eps_decay=0.995,memory_capacity=25000,lr=1e-3,train_mode=True):
        self.device = device
        self.epsilon = eps_max
        self.epsilon_min = eps_min
        self.epsilon_decay = eps_decay
        self.discount = discount
        self.state_size = state_size
        self.action_size = action_size
        self.policy_net = DQNNet(self.state_size, self.action_size, lr).to(self.device)
        self.target_net = DQNNet(self.state_size, self.action_size, lr).to(self.device)
        self.target_net.eval()
```

```
        if not train_mode:
            self.policy_net.eval()
        self.memory = ReplayMemory(capacity=memory_capacity)
    def update_target_net(self):
        self.target_net.load_state_dict(self.policy_net.state_dict())
    def update_epsilon(self):
        self.epsilon = max(self.epsilon_min, self.epsilon*self.epsilon_decay)
    def select_action(self, state):
        if random.random() <= self.epsilon:
            return random.randrange(self.action_size)
        if not torch.is_tensor(state):
            state = torch.tensor([state], dtype=torch.float32).to(self.device)
        with torch.no_grad():
            action = self.policy_net.forward(state)
        return torch.argmax(action).item()
    def learn(self, batchsize):
        if len(self.memory) < batchsize:
            return
        states, actions, next_states, rewards, dones = self.memory.sample(batchsize, self.device)
        actions=actions.type(torch.int64)
        q_pred = self.policy_net.forward(states).gather(1, actions.view(-1, 1))
        q_target = self.target_net.forward(next_states).max(dim=1).values
        q_target[dones] = 0.0
        y_j = rewards + (self.discount * q_target)
        y_j = y_j.view(-1, 1)
        self.policy_net.optimizer.zero_grad()
        loss = F.mse_loss(y_j, q_pred).mean()
        loss.backward()
        self.policy_net.optimizer.step()
    def save_model(self, filename):
        self.policy_net.save_model(filename)
    def load_model(self, filename):
        self.policy_net.load_model(filename=filename, device=self.device)

class RSU:
    def __init__(self):
        self.freq = 1*g # in Hz or cycles/s
        self.height = 5 #in meters
        self.radius = 300 #in meters
        self.stay_dist = 2*math.sqrt(self.radius**2-self.height**2) #in meters
        self.power = 2 #in watts
        self.loadfactor = 0
    def compDelay(self,task_size):
        return task_size/(self.freq/297.62)
```

```
    def energy(self,task_size):
        self.energy = self.power*self.compDelay(task_size)
        return self.energy

class Vehicle:
    def __init__(self):
        self.freq = 500*m # in Hz or cycles/s
        self.speed = 60 #km/hr
        self.speed = (self.speed*5)/18 #m/s
        self.power = 1.5 #in watts
        self.loadfactor = 0
    def stayTime(self):
        rsu = RSU()
        self.stay_time = rsu.stay_dist/self.speed
        return int(self.stay_time)
    def compDelay(self,task_size):
        self.comp_delay = task_size/(self.freq/297.62)
        return self.comp_delay
    def energy(self, task_size):
        self.energy = self.power*self.compDelay(task_size)
        return self.energy

bw = 10*m #in Hz
loss = -4
fading = 0.5
noise = -70 #in dB
noise = 10**(noise/10)
rsu = RSU()
v = Vehicle()
def rate(t):
    d = math.sqrt(rsu.height**2 + (rsu.stay_dist/2 - v.speed*t)**2)
    r = bw*math.log2(1+((v.power*(d**loss)*(fading**2))/noise))
    return r
def commDelay(task_size):
    avg_rate = quad(rate,1,v.stayTime())
    avg_rate = (abs(avg_rate[0])/v.stayTime())
    return task_size/avg_rate

states = []
states.append([0,0,0,0])
for t_size in [5,10]:
  if t_size==5:
    tdeadlim=3
  else:
```

```
    tdeadlim=5
  for t_dead in range(tdeadlim,11):
    for v_lf in range(0,11):
      for r_lf in range(0,11):
        states.append([t_size, t_dead, v_lf, r_lf])
print(len(states),states)

class Environment:
    def __init__(self, v, rsu):
        self.v = v
        self.rsu = rsu
        self.cnt = 0
        self.states = states
        self.action_space = [0,1] #0-local, 1-offload
    def reset(self):
        self.delay = 0
        v.loadfactor = 0
        rsu.loadfactor = 0
        self.loadfactor = 0
        self.cnt = 0
        return self.states[0]
    def step(self,state,action):
        task_size = state[0]
        task_deadline = state[1]
        v.loadfactor = state[2]
        rsu.loadfactor = state[3]
        self.done = False
        loc_delay = v.compDelay(task_size*m)
        rsu_delay = rsu.compDelay(task_size*m) + commDelay(task_size*m)
        self.cnt+=1
        self.next_state = self.states[self.cnt]
        self.reward = 0
        w1 = 2
        w2 = 1
        self.delay = (1-action)*loc_delay + action*rsu_delay
        self.loadfactor = (1-action)*v.loadfactor + action*rsu.loadfactor
        mean_lf = (v.loadfactor + rsu.loadfactor)/2
        self.loadfactor_var = ((self.loadfactor - mean_lf)**2)/2
        self.reward = -w1*(self.delay) - w2*(self.loadfactor_var)
        if(task_deadline < self.delay):
          self.reward -= 10
        if(v.loadfactor < 2 and action == 0):
          self.reward += 10
        if(v.loadfactor > 8 and action == 1):
```

```
        self.reward += 10
      if(v.loadfactor < 2 and action == 1):
        self.reward -= 10
      if(v.loadfactor > 8 and action == 0):
        self.reward -= 10

      if(self.cnt == 1690):
         self.done = True
      return self.next_state, self.reward, self.done

def fill_memory(env, dqn_agent, num_memory_fill_eps):
   for _ in range(num_memory_fill_eps):
      done = False
      state = env.reset()
      while not done:
         action = random.choice([0,1])
         next_state, reward, done = env.step(state,action)
         dqn_agent.memory.store(state=state,
                        action=action,
                        next_state=next_state,
                        reward=reward,
                        done=done)
         state = next_state

def train(env, dqn_agent, num_train_eps, num_memory_fill_eps, update_frequency, batchsize,
model_filename):
   fill_memory(env, dqn_agent, num_memory_fill_eps)
   print('Memory filled. Current capacity: ', len(dqn_agent.memory))
   reward_history = []
   step_cnt = 0
   best_score = -np.inf
   for ep_cnt in range(num_train_eps):
      done = False
      state = env.reset()
      ep_score = 0
      while not done:
         action = dqn_agent.select_action(state)
         next_state, reward, done = env.step(state,action)
         dqn_agent.memory.store(state=state,          action=action,          next_state=next_state,
reward=reward, done=done)
         dqn_agent.learn(batchsize=batchsize)
         if step_cnt % update_frequency == 0:
            dqn_agent.update_target_net()
         state = next_state
```

```
            ep_score += reward
            step_cnt += 1
        dqn_agent.update_epsilon()
        reward_history.append(ep_score)
        current_avg_score = np.mean(reward_history[-10:])
        if current_avg_score >= best_score:
            dqn_agent.save_model(model_filename)
            best_score = current_avg_score
        print('Ep: {}, Score: {}'.format(ep_cnt, ep_score))

def test(env, dqn_agent, num_test_eps):
    step_cnt = 0
    reward_history = []
    for ep in range(num_test_eps):
        score = 0
        done = False
        state = env.reset()
        while not done:
            action = dqn_agent.select_action(state)
            print(state, action)
            next_state, reward, done = env.step(state,action)
            score += reward
            state = next_state
            step_cnt += 1
        reward_history.append(score)
        print('Ep: {}, Score: {}'.format(ep, score))

if __name__ ==  '__main__':
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    train_mode = True
    v = Vehicle()
    rsu = RSU()
    env = Environment(v,rsu)
    model_filename = 'offload_online_net'
    if train_mode:
        dqn_agent = DQNAgent(device, state_size = 4, action_size = 2,
                    discount=0.99,
                    eps_max=1.0,
                    eps_min=0.01,
                    eps_decay=0.995,
                    memory_capacity=5000,
                    lr=1e-3,
                    train_mode=True)
        train(env=env,
```

```
            dqn_agent=dqn_agent,
            num_train_eps=1000,
            num_memory_fill_eps=3,
            update_frequency=100,
            batchsize=64,
            model_filename=model_filename)

    else:
        dqn_agent = DQNAgent(device,
                state_size = 4, action_size = 2,
                discount=0.99,
                eps_max=0.0,
                eps_min=0.0,
                eps_decay=0.0,
                train_mode=False)
        dqn_agent.load_model(model_filename)
        test(env=env, dqn_agent=dqn_agent, num_test_eps=10)

action = dqn_agent.select_action([5,7,8.7,5])
print(action)
```

PYTHON CODE FOR LSTM TRAJECTORY PREDICTION MODEL

```
import numpy as np
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.model_selection import train_test_split
#data values has to be extracted from dataset
x_values=data
seq_length = 3
X = []
for i in range(len(x_values) - seq_length):
    X.append(x_values[i:i+seq_length])
    #y.append((x_values[i+seq_length], y_values[i+seq_length]))
X = np.array(X)
print(len(X))
y = np.array([i for i in range(0,57)])
print(y)
X_train = X
X_test = X
y_train = y
y_test = y
model = Sequential()
model.add(LSTM(64, activation='relu', input_shape=(3,1), return_sequences=True))  # 2 for (x, y)
model.add(LSTM(64))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test))
'''loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test loss: {loss}, Test accuracy: {accuracy}')'''
new_trajectory = np.array([4.88939655464251, 5.019276853059908, 5.140721718075084])
predicted_values = model.predict(new_trajectory.reshape(1, seq_length, 1))
print("Predicted Values:", predicted_values[0])
```

PYTHON CODE FOR TASK OFFLOADING SCENARIO

```
v_bf = []
r_bf = []
v_lf = 0
r_lf = 0
v_size = 40
r_size = 70
print('Vehicle buffer initially empty')
print('RSU buffer initially empty')
v = Vehicle()
r = RSU()
st = v.stayTime()
tasks = [[5,5],[5,3],[10,9],[10,5],[5,2.5],[10,12],[10,9],[5,8],[10,15],[5,7],[10,7]]
for i in range(len(tasks)):
    print("Task no:", i+1)
    print("Task size", tasks[i][0])
    print("Task Deadline", tasks[i][1])
    print("Task values have been given to Task Offloading Decision Model")
    action = dqn_agent.select_action([tasks[i][0],tasks[i][1],v_lf,r_lf])
    if(action == 0):
        v_bf.append(int(tasks[i][0]))
        v_lf = (sum(v_bf)/v_size)*100
        print("This task is done at local")
    else:
        r_bf.append(int(tasks[i][0]))
        r_lf = (sum(r_bf)/r_size)*100
        print("This task has been offloaded")

    print("Vehicle's Buffer - ", v_bf)
    print("Vehicle's Loadfactor - ", v_lf, "%")
    print("RSU's Buffer - ", r_bf)

    print("RSU's Loadfactor - ", r_lf, "%")
    st = st - 3.15
    print('remaining stay time', st)
    rsu_delay = r.compDelay()+r.commDelay()
    if(action == 1 and st<rsu_delay):
        print("Remaining Stay Time of the vehicle is lesser than the execution time. So Results
cannot be returned to Vehicle by Current RSU")
        print("Hence Trajectory Prediction is done to predict the future RSU")
        print("Trajectory is predicted and task is being forwarded to RSU-B")
    print()
```