Dynamic Computation Offloading for Vehicular Edge Computing Networks

Presented By :

Mohd Gulam Mohd H. Ansari

(242CS023)

National Institute of Technology, Karnataka

Under the Guidance :

Dr. Sourav Kanti Addya

& Sushama Ma'am

Introduction

Why Vehicular Edge Computing Needed

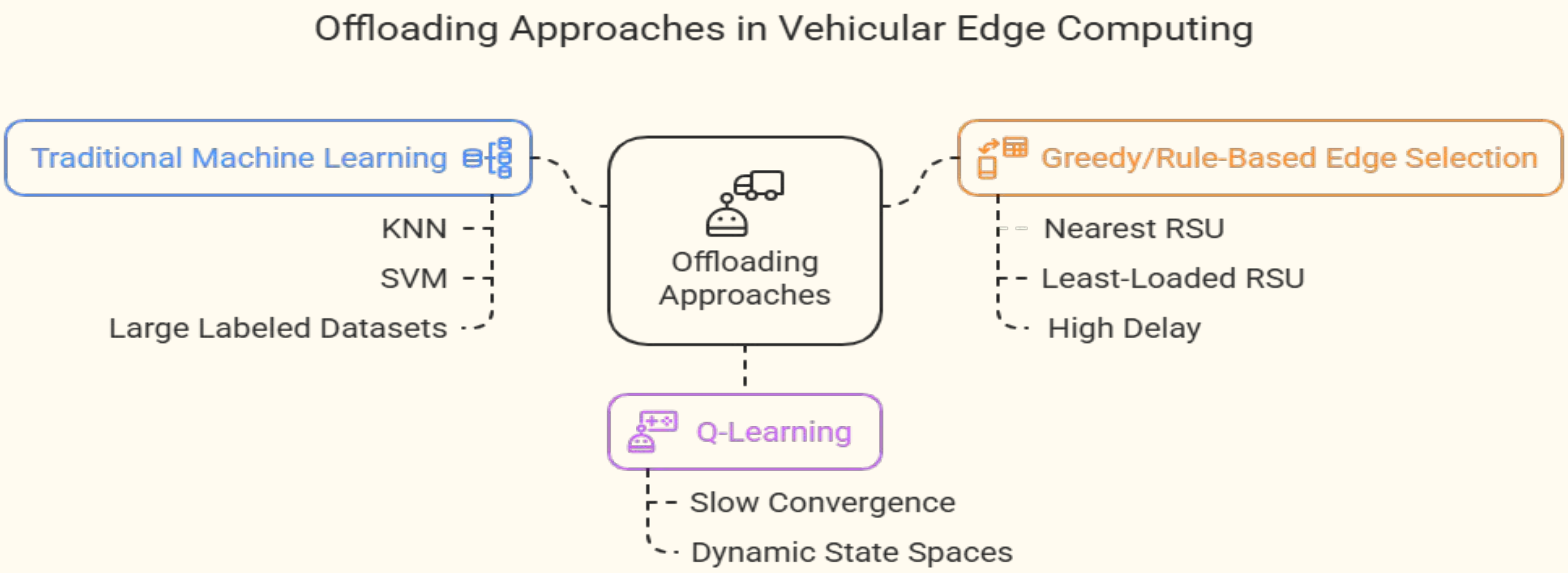
- Modern smart vehicles run compute-heavy applications (e.g., AR navigation, object detection).
- Current vehicles have limited onboard computing power and battery life, creating performance challenges.
- Vehicular Edge Computing (VEC) allows vehicles to offload computation tasks to nearby RSUs or other vehicles.
- Due to mobility, vehicle surroundings change rapidly, affecting offloading decisions, Adaptive offloading is essential to ensure quality of service (QoS) and efficiency.

Problem Statement

The problem is to design a dynamic and mobility-aware computation offloading strategy for vehicular edge computing that **minimizes task execution delay and energy consumption** .

- Vehicles must decide when and where to offload computation tasks—locally, to RSUs, or to nearby vehicles—while on the move.
- **Complexity:** Decisions must account for changing RSU availability, transmission delay, processing delay, and energy consumption — in real time.

Existing Solutions & Drawbacks



Existing methods fail to effectively handle dynamic vehicular mobility and frequent handovers, thus we need for a more stable and adaptive learning model like Double Q-Learning.

Objective

- **Reduces latency** through efficient RSU/V2V selection by dynamically prioritizing the fastest available resources, optimizing task allocation based on real-time network conditions.

- **Minimizes energy consumption** by distributing computational tasks across local, RSU, and V2V resources based on real-time energy efficiency metrics and network conditions

- Automatically **balances energy-latency** tradeoffs using reinforcement learning

Problem Formulation

Objective : Minimize the weighted sum of total delay and total energy for task

$$\mathcal{C}(t) = \delta^D \cdot D_{tot}(t) + \delta^E \cdot E_{tot}(t), \quad 0 < \delta^D, \delta^E < 1$$

In order to find $\mathcal{C}_{min}(t)$, we need to minimize D_{tot} and E_{tot} at the same time as:

$$\mathcal{C}_{min}(t) = \underset{D_{tot},\, E_{tot}}{\arg\min} \left(\frac{1}{T} \sum_{t=1}^{T} \delta^D \cdot D_{tot}(t) + \delta^E \cdot E_{tot}(t) \right).$$

$$\begin{aligned}
\text{s.t.}\quad & C_1 : 0 \le \delta^D \le 1,\ 0 \le \delta^E \le 1,\ \delta^E + \delta^D = 1 \\
& C_2 : f_{B_j,min}\ (GHz) \le f_{B_j} \le f_{B_j,max}\ (GHz) \\
& C_3 : f_{V_i,min}\ (GHz) \le f_{V_i} \le f_{V_i,max}\ (GHz) \\
& C_4 : d_{min}\ (m) \le d \le d_{max}\ (m) \\
& C_5 : D_{tot}(t) \le D_{loc}(t) = D_{deadline} \\
& C_6 : 0 \le \mu_{V_i}, \mu_{B_j}, \mu_{loc} \le 1 \\
& C_7 : \sum_{i=1}^{I} \mu_{V_i} + \sum_{j=1}^{J} \mu_{B_j} + n \cdot \mu_{loc} = 1
\end{aligned}$$

Our Approach

- **Dynamic Offloading:** Vehicles explore neighboring vehicles and RSUs after every small distance and make offloading decisions.
- **Two Q-tables** are used to decouple **action selection from value estimation**, which reduces overestimation bias common in traditional Q-learning.
- **Smart Learning:** A Double Deep Q Network (DDQN) learns from the environment (server CPU, distance, speeds, etc.) to make smarter offloading decisions.
- **Partial Task Execution:** Vehicles split their tasks and offload remaining parts in each new area.

System Model

- **Entities :**
  Vehicles (dynamic) and **RSUs** (static).

- **Mobility :**
  Follows a modified Manhattan model — vehicles move at random speeds.

- **Communication :**
  Wireless with Rayleigh fading, depends on distance and transmit power.

- **Task :**
  Tasks are large and can be partially offloaded. Each task defined by data size and computing complexity.

DDQN-based Offloading Decision

Agent : The vehicle with a large task to execute with varying computational requirement from **[6to 16 MBits]**.

State : Nearby Available RSUs/vehicles, vehicle position,velocity, direction, CPU frequencies, task size, remaining task size etc.- **Total : 11 parameters in state vector**

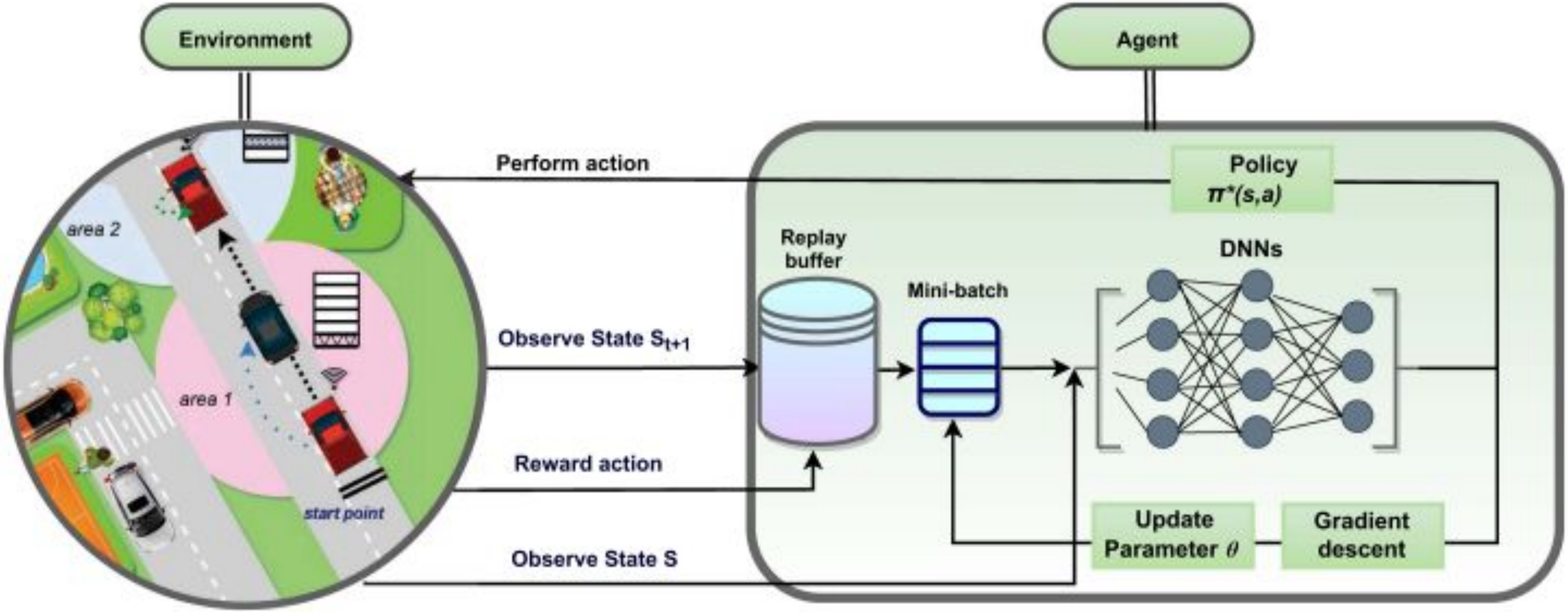
Actions : Offload to RSU, offload to another vehicle, or compute locally and partial offloading combinations- Total: **7 Actions Possible.**

Reward : Inverse of a cost function **(cost = weighted sum of delay + weighted sum of energy)**.

Learning : DDQN uses two neural networks (evaluation and target) and experience replay to stabilize training and avoid overestimation.

System Architecture

Our RL agent learns optimal offloading by interacting with the environment, optimizing actions through continuous policy updates based on latency/energy rewards



Algorithm :

Algorithm 1 DDQN-Based Task Offloading Algorithm

Input: Initialize evaluation network Q_θ, target network $Q_{\theta'}$, replay memory $\mathcal{M}$, mini-batch $\mathcal{B}$, exploration probability ϵ, discount rate γ, learning episodes X

for episode $x = 1$ to X **do**
  if $L > 0$ **then**
    Explore environment for candidates
    if no candidate found in range **then**
      Execute the task locally, $L = L$-executed task
    else
      for each evaluation step **do**
        Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
        Execute a_t, $L = L$-executed task, observe s_{t+1} and reward r_t
        Store (s_t, a_t, r_t, s_{t+1}) in experience memory $\mathcal{M}$
      end for

Algorithm :

for each target step **do**

  Sample random mini-batch

  $\mathcal{B}_\tau = (s_\tau, a_\tau, r_\tau, s_{\tau+1}) \sim \mathcal{M}$

  for each $\tau \in \mathcal{B}$ **do**

    $Y_\tau = r_\tau + \gamma \cdot \max_{a \in \mathcal{A}} Q_{\theta'}(s_{\tau+1}, a)$

  end for

  Update Q_θ via gradient descent on

    $[Y_\tau - Q_\theta(s_\tau, a_\tau)]^2$

  Update target network parameters: $\theta' \leftarrow \theta$

end for

=0

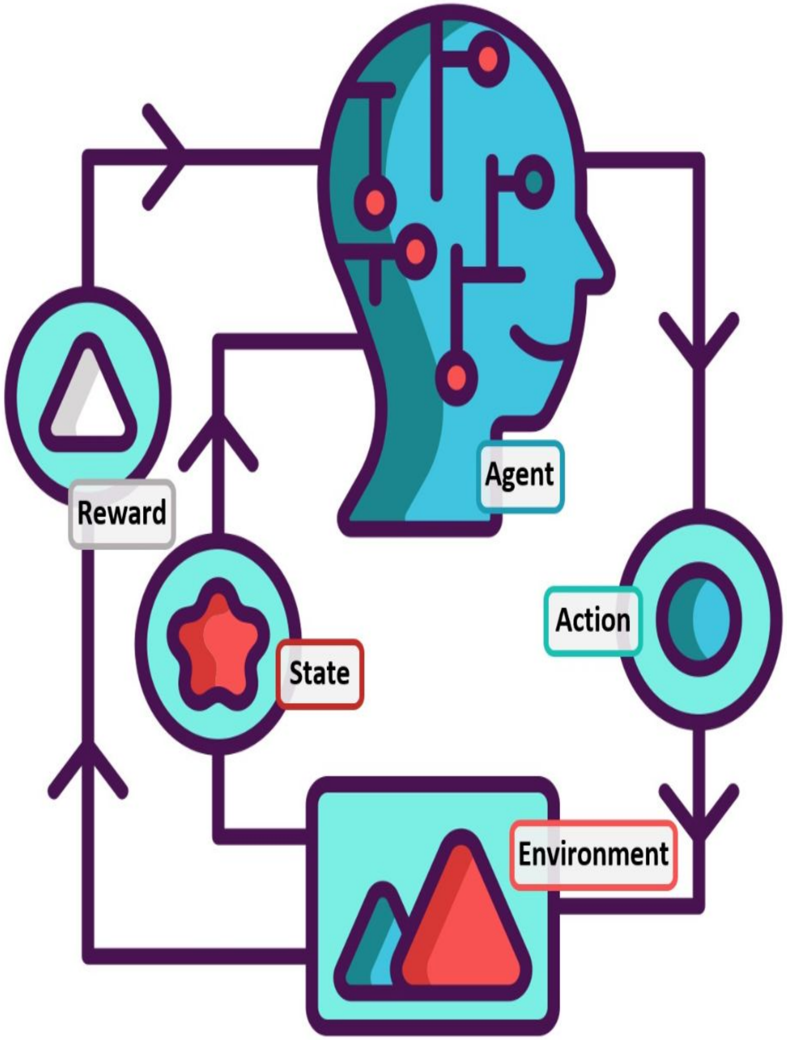
Tools Used

- **DDQN Model :**
  TensorFlow for building and training the DDQN agent

- **Programming Language :** Python to interact with model and computations of matrice

- **Markov Model:** Used to represent the decision-making process in task offloading

- **Matplotlib :** for plotting graph such as Task Offloading Distribution (e.g., % Local, RSU, V2V)



Simulation Parameters

- **80 Vehicles** with varying velocity
- **30 fixed RSUs** distributed over 10Km Road Segment
- The speed of vehicles is assumed to be different and distributed uniformly between **[30,120] km/h.**
- The communication range is considered as **200 meters**.
- CPU frequencies of vehicles and RSUs are randomly distributed in the range **[2,8] and [8,16] GHz, respectively.**

Component	Details
Number of Vehicles	80
Number of RSUs	30
Route Length	10 km
Vehicle Speed	30–120 km/h
Communication Range	200 meters
CPU Frequency (Vehicles)	2–8 GHz
CPU Frequency (RSUs)	8–16 GHz
Task Size (L)	16 Mbits
Processing Complexity (ρ)	1000 cycles/bit
Processing Constant (κ)	10^{-27} Watt·s^3/cycles^3
Bandwidth (W)	10 MHz
Vehicle Power (P_V)	0.1 W
Noise Power (N_0)	1
Parameter α	1
Parameter β	0.01
Discount Factor (γ)	0.9
Batch Size (B)	32
Learning Rate	0.05
Episodes	400

Evaluation Metrics

Average Task Delay : The mean time taken (in seconds) to complete a computational task, averaged across all tasks in the evaluation

The end-to-end latency for task completion is calculated as:

$$D_{off} = D_{UL} + D_{edge} + D_{DL}$$

where:

$$D_{edge} = \frac{\rho \cdot L}{f_{edge}}, \quad D_{UL} = \alpha \cdot \frac{L}{R_{UL}}, \quad D_{DL} = \beta \cdot \frac{L}{R_{DL}}$$

- ρ: Computation intensity (cycles/bit)
- L: Task size (bits)
- f_{edge}: Edge server frequency (cycles/sec)
- R_{UL}, R_{DL}: Uplink/Downlink rates (bits/sec)
- α, β: Protocol overhead factors

Evaluation Metrics

Average Energy Per Task : The mean energy expended (in Joules) to complete a computational task, averaged across all tasks in the evaluation

Total energy per task combines computation and transmission:

$$E_{total} = E_{loc} + E_{off}$$

where local computation energy is:

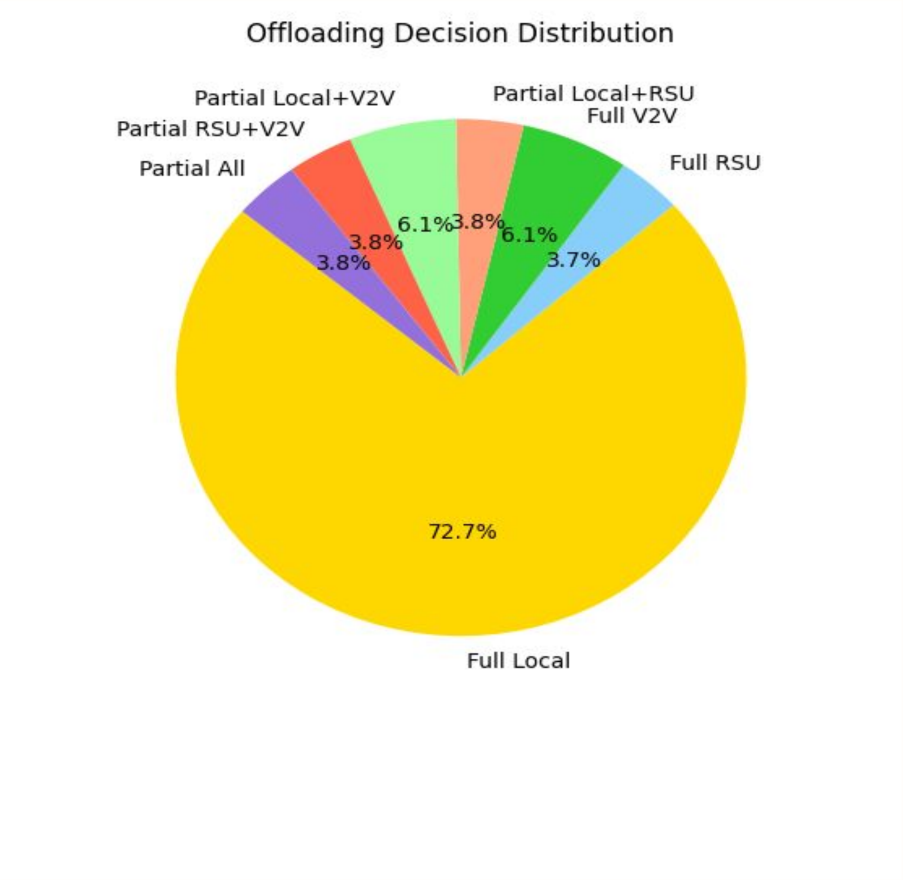
$$E_{loc} = P^{CPU} \cdot \rho \cdot L \quad \text{with} \quad P^{CPU} = \kappa \cdot f_{loc}^2$$

and offloading energy is:

$$E_{off} = \frac{P_{tx} \cdot \rho \cdot L}{R_{UL}} + \frac{P_{rx} \cdot \rho \cdot L}{R_{DL}}$$

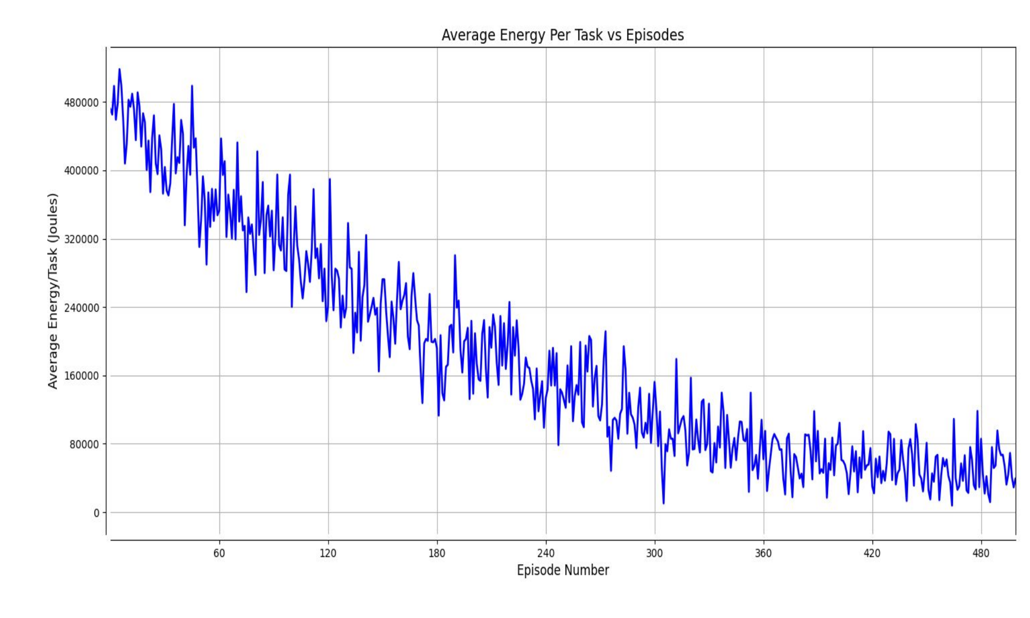
Offloading Decision Distribution

- Results demonstrate that **local offloading** naturally emerges as the dominant strategy (72.7% of cases) under realistic vehicular constraints, offering reliable low-energy computation when **tasks are compute-light**



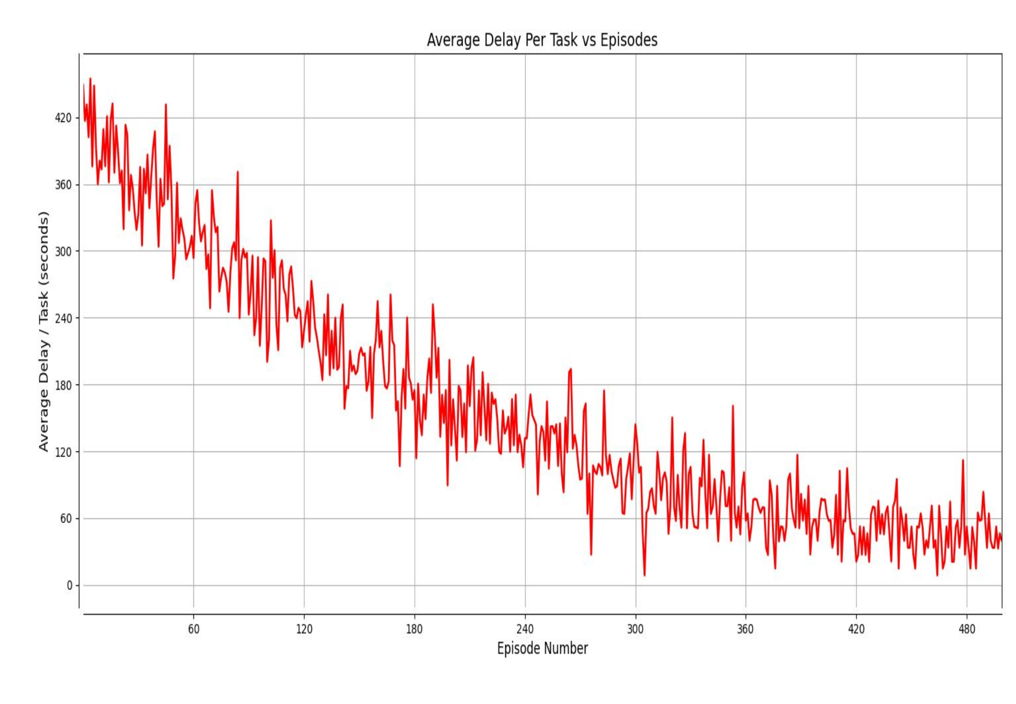
Energy Consumption Per Task

- Achieves **reduction in energy per task within first 300** episodes, demonstrating fast convergence to efficient offloading policies
- Maintains **consistent low-energy operation after convergence**, proving algorithmic robustness



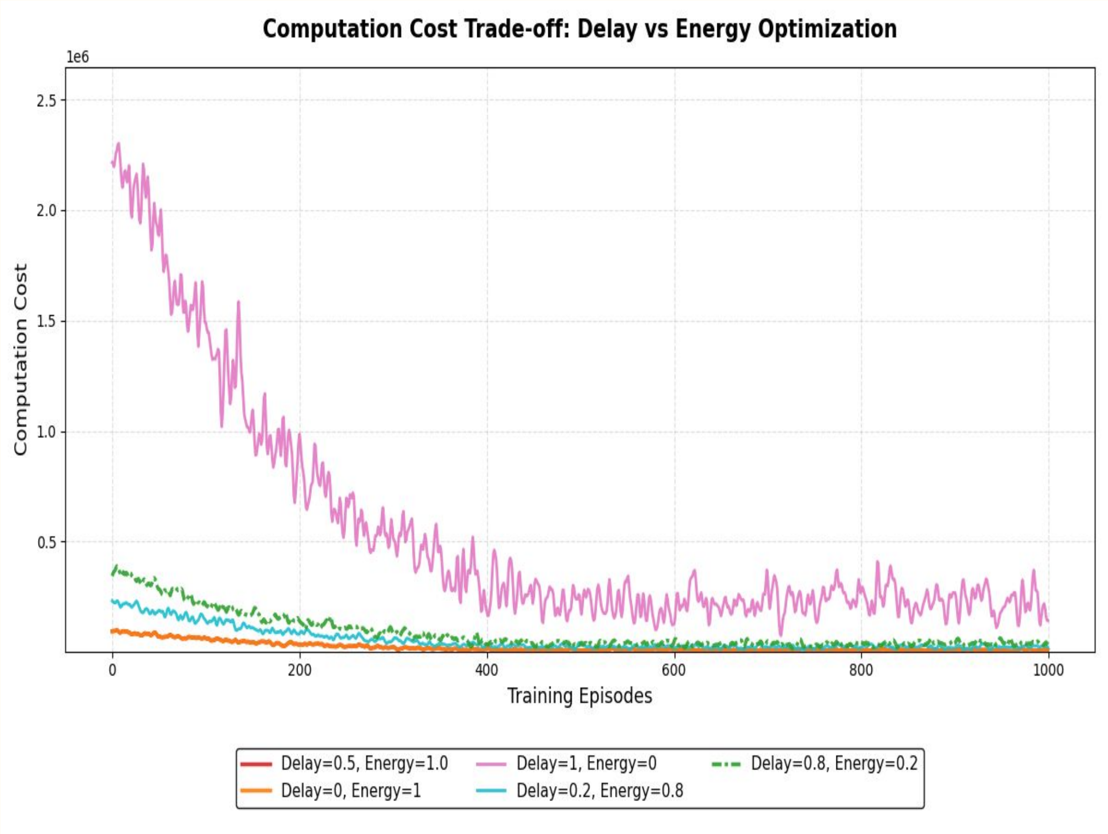
Average Latency Per Task

- Reduces average task delay through dynamic offloading to low-latency resources (RSU/V2V)
- Maintains consistent delays after convergence, enabling reliable real-time processing



Computation Cost vs Episodes

- The graph shows a direct inverse relationship between delay optimization and energy efficiency - **prioritizing lower delays (pink line) results in significantly higher computation costs**, while **prioritizing energy savings (orange line) achieves the lowest operational costs.**



Conclusion

- Our handover-enabled DDQN uses experience replay, dual neural networks, and DNN to stabilize decisions in dynamic environments.

- Our model achieves lower latency for time-sensitive tasks while intelligently balancing the energy trade-off through adaptive cost-weighted optimization.Open-source for community adoption.

Future work

- **RSU Load Awareness and Balancing :**
  Current implementation assumes RSUs are always available for task execution without considering their real-time load or capacity constraints.

Future work should incorporate a dynamic RSU load-balancing mechanism that:

- Task queue at each RSU

- Avoids overloading busy RSUs while improving task distribution efficiency
  This will lead to improved QoS, fairness, and scalability in vehicular edge computing scenarios.

Thank You !