A Multi-Agent Deep Q-Learning Framework for Uplink Congestion Control in Communication Networks

Amaal S. A. El-Hameed[1]

[1]Department of Electronics and Communications Engineering, Faculty of Engineering, Cairo University, Giza, Egypt 12613, amaal@eng.cu.edu.eg
[1]Department of Computer Engineering Faculty of Engineering, May University, Cairo, Egypt, Amaal.Samir@muc.edu.eg

Abstract
Congestion control is crucial for reliable and efficient communication in networks. Current systems mainly focus on managing download traffic. However, upload communication is becoming increasingly important with the rise of 5G/6G networks, IoT devices, cloud services, and real-time apps. Upload channels have limited resources, lack predictability, and are shared by multiple users, making congestion management challenging. This paper proposes an AI and machine learning-based framework to address uplink congestion. It uses reinforcement learning to dynamically adjust transmission rates and incorporates fairness mechanisms for multi-user scenarios. The results demonstrate that this approach improves throughput, reduces delays, and enhances fairness compared to traditional methods, suggesting that AI-driven solutions hold promise for future networks.

Keywords: Congestion control, uplink, AI/ML, reinforcement learning, fairness, next-generation networks.

1. Introduction

Network congestion control poses a key challenge in computer and communication networks. It includes a group of methods that aim to control data sending speeds to stop networks from getting overloaded. When too much traffic fights for limited bandwidth or buffer space, the network can suffer packet loss higher delays uneven timing, and slower speeds [1]. In bad cases, congestion can even cause the network to break down if not handled. Old ways to control congestion, like those in TCP versions (such as Tahoe, Reno, Cubic) react to changes based on lost packets or changes in round-trip time [2], [3], [4],[5]. These plans work well for download-focused cases, but they don't suit the special features and issues of upload communication.

Uplink communication, which refers to the transmission of data from users or devices to the network, has become more crucial in today's systems. The rise of mobile and wireless devices has boosted the amount of uplink traffic. Smartphones, Internet of Things (IoT) devices, and sensors now upload more multimedia content, telemetry, and real-time data than ever before. In cloud and edge computing, uplink streams play a key role in sending complex tasks to servers. Also, team-based apps like online games, video calls, and remote healthcare need reliable fast uplink sending to keep users happy. The arrival of 5G and upcoming 6G networks has put uplink communication at the heart of new services. These include augmented/virtual reality (AR/VR), self-driving systems, and large-scale IoT setups [6], [7].

Despite its increasing importance, congestion control for the uplink has remained relatively less investigated than congestion control for downlink communication. This is because, unlike downlinks, uplink channels tend to be more bandwidth-scarce, shared by a number of users. Secondly, uplink channels tend to exhibit bursty, unpredictable communication patterns, making it difficult to allocate channels through congestion control in an efficient way. The problem is made more complex by the issue of ensuring fairness for each user when designing congestion-control algorithms. Additionally, congestion-control algorithms, designed using traditional congestion-control frameworks, tend to fall short when it comes to satisfying the requirements of latency-sensitive applications.

Recent breakthroughs in artificial intelligence and machine learning have given researchers new tools to tackle network congestion. AI techniques like reinforcement learning and deep learning let congestion control systems react on the fly to networks that are always changing and never quite the same from one moment to the next [5], [8], [9], [10], [11],[12]. Unlike the old rule-based methods, machine learning models can actually predict when congestion will hit, balance tricky trade-offs between throughput, latency, and fairness, and deal with the massive wave of data traffic heading upstream.

Given all this, it makes sense that research into AI and ML-based approaches for uplink congestion control is picking up speed. Sure, we've spent years fine-tuning downlink congestion control, but uplink channels come with their own quirks and demands. They need smart, adaptive solutions[13], [14]. This paper aims to deliver a new intelligent mechanisms that boost uplink performance, laying the groundwork for faster, more scalable networks as we move toward next-generation communication systems.

1.1 Contributions of the Paper

The main contributions of this paper are as follows:

1. The Problem: Uplink Neglect

- Current congestion control methods are mostly focused on downlink traffic.
- Uplink communication has been overlooked.
- As networks get more complex and demands shift, this gap turns into a real problem, creating a need for congestion control built specifically for uplink scenarios.

2. The Proposed Solution: An AI/ML Framework

- We designed an AI and machine learning framework for uplink congestion control.
- Its core uses Reinforcement Learning (RL) to intelligently adjust transmission rates, responding dynamically to real-time network changes.
- The framework doesn't just achieve efficiency, but uses fairness-aware mechanisms to ensure efficient and equitable allocation of uplink resources among multiple competing users.

3. Key Results & Benefits

- Simulations demonstrate clear performance gains over traditional methods:
- Higher Throughput: Maximizes data transfer on the uplink.
- Lower Latency: Reduces delays for responsive communication.
- Improved Fairness: Ensures equitable resource sharing among users.

4. Future Impact & Significance

- These results lay the groundwork for smarter, scalable congestion control.
- The framework is precisely what is needed to meet the demands of 5G & 6G Networks
- The massive scale of the Internet of Things (IoT)
- Next-generation applications requiring robust bidirectional data flow.

Here's how this paper unfolds. Section 2 reviews related work, focusing on uplink communication and AI/ML-based approaches. In Section 3, we detail our framework and methodology. Section 4 covers the simulation setup and presents our comparative analysis results. Finally, Section 5 wraps up with our conclusions and a look at future research directions.

2. Related Work

2.1 Traditional Congestion Control Approaches

For years, congestion control has relied on classic approaches—mainly TCP variants and Active Queue Management (AQM). TCP, the backbone of end-to-end congestion control on the Internet, started with versions like Tahoe and Reno. These early variants brought in slow start, congestion avoidance, and fast retransmit / recovery. They looked for packet loss as a sign of trouble [1], [2], [3], [15].

Later, with New-Reno and Cubic, developers tweaked these ideas to handle bigger bandwidth and longer delays, especially in modern networks [4], [16],[7].

Still, all these TCP-based methods work reactively. They only notice congestion after something's already gone wrong—lost packets or rising delays [17]. That's a big problem when traffic on the uplink gets unpredictable or bursts in short spikes. Plus, TCP's design really caters to downlink traffic, so sending data upstream hasn't been as efficient or well-optimized. Routers try to help out with congestion, too.

That's where AQM comes in [11]. Techniques like Random Early Detection (RED) step in before buffers fill up and start dropping or marking packets to warn end hosts about congestion [3]. Other versions—Adaptive RED, Blue, and CoDel—fine-tune these reactions, adjusting how aggressively they drop packets to keep queues stable and latency low [5]. These ideas help, but they're not magic. AQM needs careful tuning, and not every router in a big, mixed network gets set up the same way. That inconsistency can limit its impact.

Despite all this progress, both TCP variants and AQM still lean on pre-set rules and wait for signs of trouble before acting. They struggle with the rise of complex uplink traffic, where different applications compete, users expect fairness, and everyone wants low latency. Traditional methods shaped the reliable Internet we have now, but their reactive, rule-based nature—and the focus on downlink—leave gaps.Different researchers tried to propose different techniques to overcome these troubles. Abreuet al. [18] proposed the possibility of adopting the Bottleneck Bandwidth and Round-trip propagation time (BBR) protocol as the default congestion control mechanism for TCP in a way to improve the throughput but it caused higher latency and jitter.

Now, newer approaches using AI and machine learning are starting to shake things up. These methods predict and adapt rather than just react, showing a lot of promise. But so far, most studies stick to general scenarios or focus on downlink traffic. Only a handful look closely at uplink's unique challenges like burstiness, resource imbalances, and the need to keep things fair for many users at once. Even with reinforcement learning and deep learning making strides, deploying them in the real world remains challenging. Balancing scalability, fairness, and latency in practice remains a tough nut to crack.

2.2 AI/ML-Based Congestion Control Approaches

Lately, artificial intelligence and machine learning have started to shake up the world of congestion control. Instead of relying on fixed rules, these newer methods use predictive models, adapt on the fly, and juggle multiple goals at once. Networks aren't static—they're messy, unpredictable, and always changing. AI and ML handle that chaos much better [19].

Reinforcement learning (RL), for example, lets algorithms learn how to send data efficiently by actually interacting with the network. Take systems like PCC-RL [7], [9] or RL frameworks in ns-3 [11], [17], [20], which have shown they can beat traditional TCP on both speed and delay [12]. RL shines in uplink scenarios, where traffic can be sudden and erratic. Here, you need algorithms that can react quickly [21], [22], [23], [24]. Remy [6] and PCC (Performance-oriented Congestion Control) [7] were among the first to move away from hand-tuned rules, letting systems learn strategies from offline training or even by optimizing utility functions. Tong et al. proposed an adaptive algorithm for congestion control (ABBR) using reinforcement learning. Its objective is selecting an appropriate congestion control algorithm corresponding to each network condition which allows execution of a real-time solution in congestion control [25].

Deep reinforcement learning takes things a step further. Projects like [26], [27], [28], [29],[30], [31], [32], [27], [28] use neural networks to learn how to adjust data rates in real time, directly from feedback. No need for experts to design the rules; the neural networks figure it out on their own. Supervised and deep learning models have also entered the scene. These systems predict congestion by reading network measurements, which lets them control things before problems even happen [35]. Neural networks are especially good at capturing the tangled relationships between traffic, delay, and packet loss—making them a strong fit for the massive scale of next-gen networks. Then there's multi-agent learning. As more users send data simultaneously, multi-agent reinforcement learning (MARL) helps manage shared resources while keeping things fair [14], [36], [37], [38]. Distributed agents coordinate with each other, balancing efficiency and fairness—crucial when resources run tight. Altogether, AI and ML bring a flexible, adaptive mindset to congestion control—especially on the uplink. But the field isn't solved. It's tough to make these systems generalize across wildly different networks, keep the training stable, and avoid heavy deployment costs.

Our work pushes this research further by building a multi-agent Deep Q-Network framework for uplink congestion control, tuning the balance between throughput and latency. To tackle these ongoing challenges, this paper presents a reinforcement learning framework designed specifically for uplink congestion control. Our approach combines adaptive rate control with fairness-aware, multi-agent mechanisms, closing the gap between older and newer methods and pointing toward efficient, scalable solutions for the networks of the future.

3. The Methodology and Proposed Scheme

In this section, we explain the system model and the proposed framework for congestion control.

3.1 Basics of Deep Q-learning

Deep Q-learning blends Q-learning, a classic way for agents to figure out the best actions in an environment, with deep neural networks. The idea is to help an agent learn what to do in huge, complex spaces where traditional approaches just can't keep up. Instead of storing endless look-up tables, we use a neural network to estimate the Q-function, which tells us the expected reward for each possible action in a given state. The network keeps updating itself step by step, mixing exploration (trying new things) and exploitation (sticking with what works). But there's a catch: training can get unstable, with problems like non-stationarity and divergence. To keep things on track, we use tricks like experience replay and target networks. Deep Q-learning has already shown it can train agents for all kinds of tasks—think video games, robotic arms, and more [35],[39].

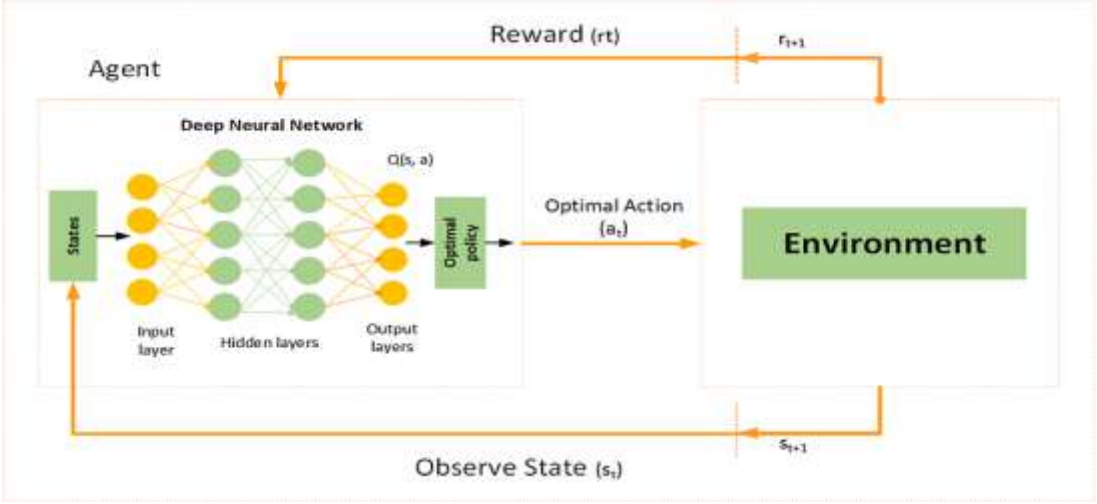

Figure.1: Structure of Deep Q-Learning

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. All the past experience is stored by the user in memory. The next action is determined by the maximum output of the Q-network. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. we have:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\boxed{R_{t+1} + \gamma \underbrace{max}_{a} Q(S_{t+1}, a, \theta^-)} - Q(S_t, A_t, \theta) \right] \qquad [1]$$

Where:
θ are the weights of the main Q-network, $\theta^-$are the weights of the target Q-network, S_t is the current state, A_t is the action taken, R_{t+1} is the reward received, S_{t+1} is the next state, and $\underbrace{max}_{a} Q(S_{t+1}, a, \theta^-)$ is the maximum Q-value for the next state.
The section in box represents the target. We can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge.
To make training more stable, DQN uses a replay buffer—a big memory bank that stores each (state, action, reward, next state) tuple. Instead of training on consecutive experiences, which are often correlated and can mess up learning, the agent samples random mini-batches from this buffer. This breaks up the correlations and lets the network learn more efficiently.
The training process of a DQN involves the following steps:
First, we initialize the replay buffer, the main network (θ), and the target network (θ^-). We also set up the key hyper-parameters: learning rate (α), discount factor (γ), and the exploration rate (ϵ).
We use an ϵ-greedy strategy to balance exploration and exploitation. With probability ϵ, the agent picks a random action to explore. Otherwise, it chooses the action with the highest Q-value.
As the agent interacts with the environment, it collects experiences—each one a tuple of state, action, reward, and next state—and stores them in the replay buffer.
For each training update:

- Sample a mini-batch of experiences from the buffer.
- Compute target Q-values using the target network.
- Update the main network by minimizing the loss with gradient descent.

Every so often, we copy the main network's weights over to the target network to keep things stable. Over time, we slowly decrease ϵ so the agent explores less and exploits its learned knowledge more.
Deep Q-Learning has made waves in several fields:

- **Atari games**: Agents learn to play classic games straight from pixel data, often beating human players.
- **Robotics**: Robots use it to master tasks like picking up objects and moving around—skills that once seemed out of reach.
- **Self-Driving Cars:** It helps cars to make decisions like changing lanes and avoiding obstacles safely.

3.2 Design Objectives

This framework aims to build an adaptive congestion control system for uplink communication. Instead of waiting for problems like packet loss to signal congestion, it uses reinforcement learning to adjust transmission rates on the fly as network conditions change. The goals are clear:

- **Maximize Throughput:** Squeeze as much data as possible from the network link.
- **Prevent Congestion:** Avoid allowing a backlog of data (congestion) to build up.
- **Minimize Latency:** Keep delays low to ensure smooth performance for real-time applications like video calls and interactive services.
- **Ensure Fairness:** Guarantee that multiple users sharing the network have fair and equitable access to resources when sending data simultaneously.

3.3 System Architecture

We consider an uplink communication network with N users, each sending data to the same base station over a single, shared link. That link tops out at a capacity of C. At every time step t, user i ∈ {1,2,...,N} chooses a sending rate, $x_i(t)$. The data doesn't flow in at a steady pace. Instead, arrivals follow a Pareto distribution, so the traffic bursts in unpredictable, self-similar patterns, a hallmark of real-world uplink data. We set the link's capacity to C = 10 Mbps. The base station has only so much space in its buffer, B. When all users together try to send more than the link can handle, packets start piling up in that buffer.

$$Q(t+1) = \max(0, Q(t) + \Sigma x_i(t) - C), \qquad [2]$$

where Q(t) is the instantaneous queue length. The resulting queueing delay for user i is proportional to:
$d_i(t) = Q(t) / C.$ [3]
Thus, each user faces a trade-off between increasing throughput $x_i(t)$ and avoiding excessive queue-induced delay $d_i(t)$.
We treat congestion control as a Markov Decision Process (MDP). Where the network simulator acts as the environment. It mimics the real network, shifting uplink traffic, changing queue sizes, and fluctuating channel conditions. The environment gives the agent feedback—think delay, packet loss, and throughput—based on its actions. Now, the agent, our RL controller, learns how to manage congestion by interacting with this environment. Every time step, the agent checks out the current network state, the queue length, RTT, throughput and then decides how to tweak the sending rate. Each agent (in this case, each user) learns on its own to fine-tune how fast it sends data. For the state ($S_i(t)$): that's just what agent i sees at time t.

$S_i(t) = [x_i(t), d_i(t), \hat{x}_i$ (t-1), $\hat{d}_i$ (t-1)] [4]

Where $\hat{x}_i$ and $\hat{d}_i$ are exponentially smoothed estimates of past throughput and delay.
Action ($a_i(t)$,): Each agent chooses one of three discrete actions at every step:

- Increase sending rate (+Δ)
- Decrease sending rate (−Δ)
- Maintain current rate (0)

The reward function steers the learning process, juggling throughput, latency, and fairness all at once. Take $R_i(t)$ as an example—it blends efficiency and responsiveness into a single objective.

$R_i(t) = \beta * (x_i(t) /C) - \alpha * (d_i(t) /d_{max})$, [5]

Where α, β > 0 balance throughput and delay, and d_{max} normalizes the maximum tolerable delay.

The policy optimizer steps in with reinforcement learning algorithms like Deep Q-Networks (DQN) or Policy Gradient methods. These algorithms update the policies and sharpen decision-making as time goes on.

3.4 Multi-Agent Extension

In this work, our focus is at multi-agent learning, not just the simpler single-flow case. Here, every user gets their own agent and trains a separate DQN at the same time as everyone else. When a new user shows up, they're assigned to an open agent slot. If there aren't any free slots, we just drop them, this models a real system's capacity limits. Each agent learns its own Deep Q-Network to estimate the best action-value function.

$Q(s,a;\theta) \approx \max_\pi E[\Sigma \gamma^t R(t) \mid s\_0=s, a\_0=a, \pi]$, [6]

where π is the policy, γ ∈ (0,1) is the discount factor, and θ are the neural network parameters.
The DQN takes in a state vector $S_i(t)$, as input, passes it through two hidden layers with ReLU activations, and finally outputs Q-values for all possible actions. To keep learning stable, we use replay buffers and target networks. The replay buffer (B) holds old transitions (S_t, A_t, R_t, S_{t+1}) so the agent can sample from them randomly and avoid learning from highly correlated experiences. The target network is a slower-moving copy of the main Q-network, updated every K steps, which helps smooth out the learning process. The training loss is defined as:

$L(\theta) = E[(R_{t+1} + \gamma \underbrace{max}_{a} Q(S_{t+1}, a, \theta^-) - Q(S\_t, A\_t, \theta)^2]$ [7]

Agents never exchange information directly. Each one pulls experience just from its own replay buffer. Target networks get their updates on a schedule, not continuously. The real coordination happens through the environment where everyone feels the same queueing delay, so they end up adjusting together. They all explore using an epsilon-greedy approach. No special fairness mechanism is needed; if one agent gets pushy and causes big delays, the rest pick up on it and change how they act. In the end, balance just emerges from everyone responding to the shared feedback. You can see the complete Markov process mapped out in Figure 2.

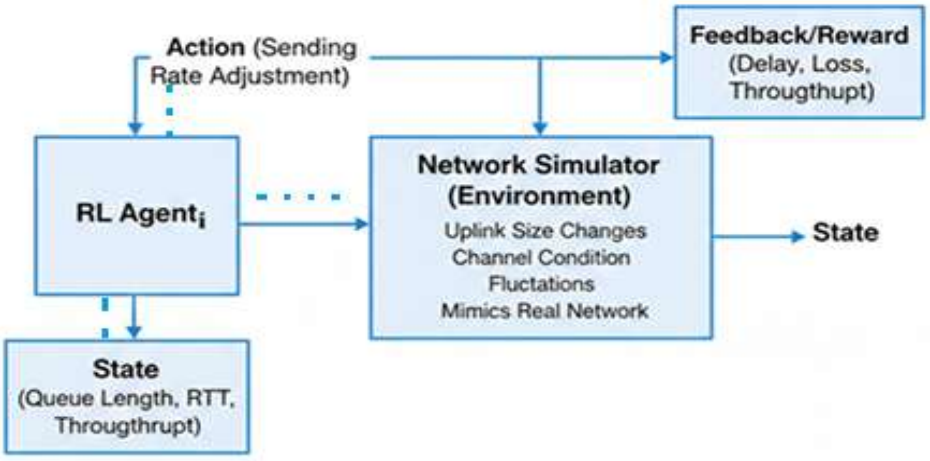

Figure .2: Markov Decision Process for Multi-Agent Congestion Control

Multiple Independent RL Agents Interact with a Shared Environment.
Environment. Each Agent Latent Choses an Action (Sending Rates),
'Affecting' Sent State and Environment. Agents Learn Optimal Policies Individually.

3.5 Implementation Workflow

In this section, we explain the proposed framework step by step. First, we set up the simulation environment: we lay out the network topology, define traffic sources, and set buffer sizes to match real-world uplink scenarios. Once the environment's ready, the reinforcement learning agents get to work. They watch the network closely, tracking things like queue occupancy, delay, and throughput. At every decision point, each agent tweaks its transmission rate based on what it sees.

Now, the environment doesn't just sit back. It pushes back with feedback: numbers for throughput, latency, and packet loss. These feed directly into the reward signal, which steers the agents as they update their policies via the chosen reinforcement learning algorithm—in our case, DQN.

We start by initializing the DQNs and replay buffers for every agent. At each time step t, an agent observes the current state S(t) and picks an action using the epsilon-greedy approach. The environment processes this action, updates the throughput and delay, and sends back the reward. We record these transitions in replay buffers. Then, by sampling mini-batches from these buffers, we update the networks using gradient descent. We also update the target networks at regular intervals. As training runs over multiple episodes, the agents keep honing their strategies, zeroing in on stable and efficient congestion control—specifically tuned for uplink communication. The entire system flow appears in Figure 3.

To see how our method stacks up, we pit it against TCP Reno, and CoDel AQM. We use synthetic traffic models in MATLAB and focus on three main metrics: throughput (Mbps), delay (ms), and the packet loss. Also, we show the superior of our work by showing the adaptability of the proposed scheme for large traffic.

To calculate the average throughput, we measures total successfully delivered bits per second across all users

$$T = \frac{\sum_i^N Packets_{i,delivered} \times Packet\ Size}{T_{Sim}} \quad [11]$$

For the average End-to-End Delay (D), we calculate the packet latency experienced during transmission.

$$D = \frac{\sum_i^N Arrival\ Time_i - Send\ Time_i}{N} \quad [12]$$

As for the Packet loss, it occurs when one or more packets of data traveling across a network fail to reach their destination **[40]**.

Packet loss percentage is typically defined as:

Packet Loss Rate (%) = ((Total Packets Sent - Packets Received) / Total Packets Sent) * 100% [13]

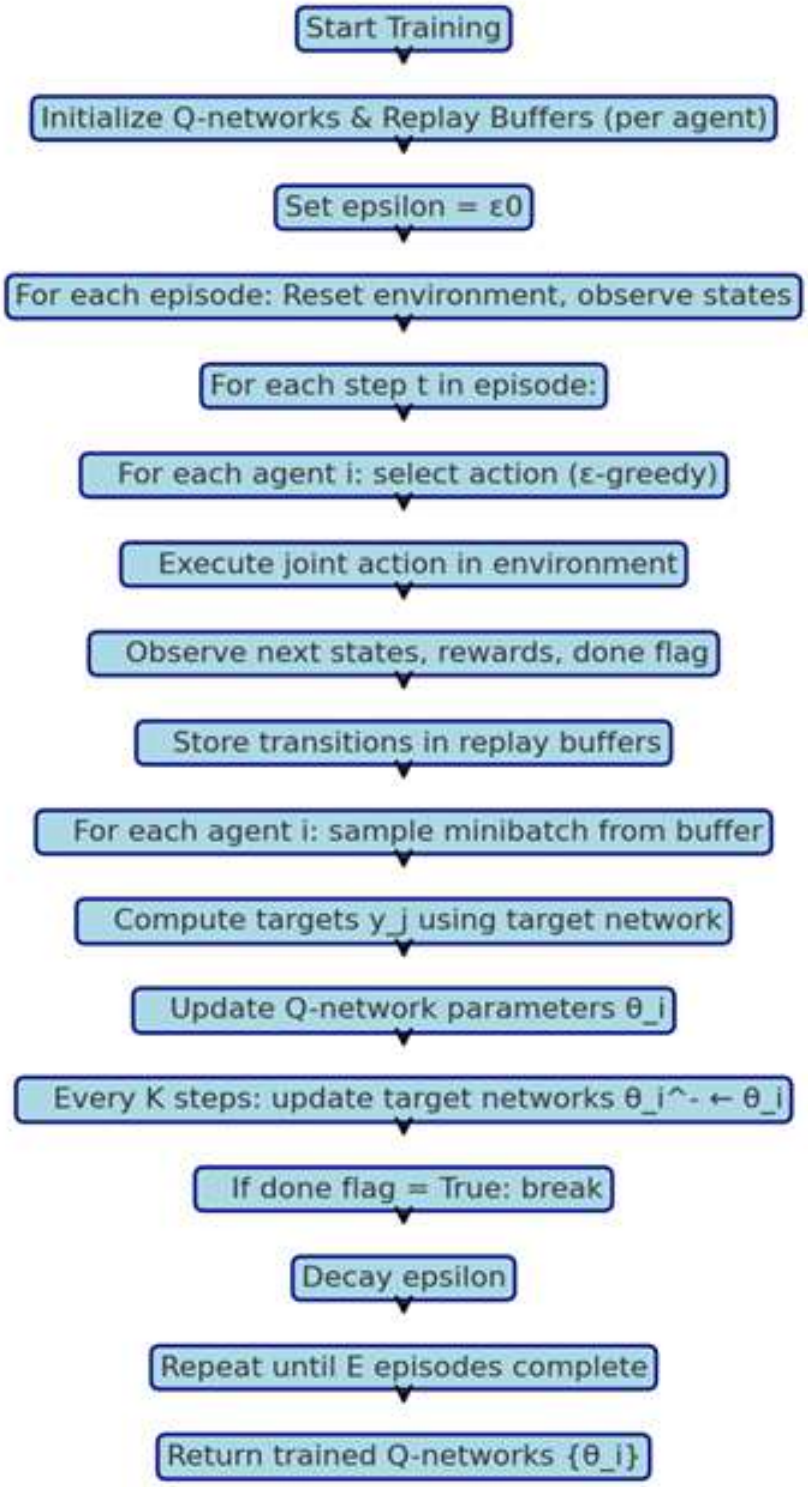

Figure.3: illustrates the methodological flowchart used in this study.

4. Performance Evaluation Results

This evaluation assess how well the proposed Multi-Agent Deep Q-Learning (MADQN) framework handles uplink congestion control. The main focus is on three things: can the model boost throughput, cut down packet delay and loss, and keep things fair, especially when traffic patterns get messy and unpredictable? To get a real sense of its strengths, I put the DQN approach head-to-head with some classic congestion control methods—TCP Reno and CoDel AQM. These two have been staples in network performance research for a long time [1],[2].

This section also examines how the framework scales. What happens as more users jump into the network? Can the MADQN agents keep up when network loads swing up and down, or when traffic comes in random bursts? These are the kinds of challenges real-world networks face every day.
For testing, a MATLAB-based synthetic traffic setup that mimics a standard uplink communication network is used. The environment models a Heterogeneous Cellular Network (HCN) with one macro eNB (MeNB) and several pico eNBs (PeNBs), all sharing the same frequency bandwidth. There are two main types of devices in play M2M and H2H each with their own quality-of-service demands when connecting to the eNBs, as shown in Figure 4. The system features multiple users (N) sending data to a single access point, where congestion pops up when bandwidth runs tight or buffers fill up.

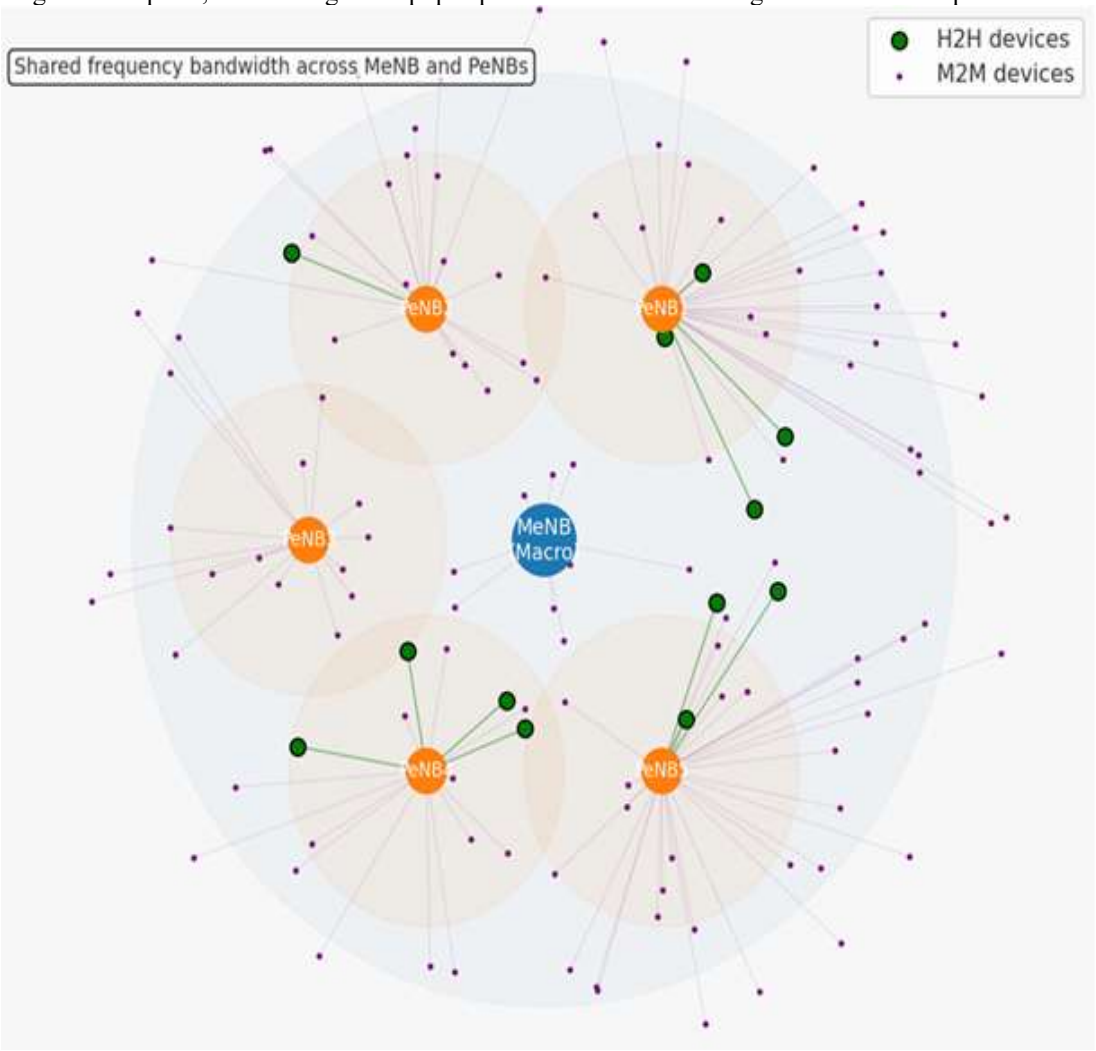

Figure.4: An Heterogeneous Cellular Network (HCN) diagram showing 1 MeNB (macro) and K−1 PeNBs (picos), with M2M and H2H devices connecting to their nearest eNB.

Traffic in the system follows a Pareto-distributed arrival process, which means you get those unpredictable bursts that are so common in uplink data flows. Each user sends traffic using a Pareto on–off pattern—a pretty faithful match to how real internet traffic behaves when it gets bursty [3]. When a user's in the "on" phase, they generate packets at a variable rate; the DQN agent can decide to ramp it up, keep it steady, or slow it down. In the "off" phase, the user just goes quiet. This self-similar pattern of Pareto arrivals puts the model through its paces, especially when things get heavy-tailed and unpredictable.
Researchers rely on the Pareto on–off model [19-20] for network performance studies because it captures the self-similar, bursty nature of internet traffic. Actual measurements of web and video traffic show that the time between arrivals and the length of flows often follow heavy-tailed distributions, so the Pareto model fits well [1],[2].
Every traffic source flips between ON (active) and OFF (silent) periods.

- During the ON period, packets are generated at a variable bit rate.
- During the OFF period, the source remains silent.

Both ON and OFF periods are random and follow the Pareto distribution

$$F(x) = \frac{\alpha \times x_m^{\alpha}}{x^{\alpha+1}}, \qquad x \geq x_m \qquad [8]$$

Where:

- α is the *shape parameter* (1<α<2) that determines the "heaviness" of the tail,
- x_m is the *minimum value* (scale parameter).

The mean and variance of the Pareto distribution are given by:

$E[X] = (\alpha\, x_m)/(\alpha-1)$ [9]

$VAR[X] = (\alpha x_m^2)/(\alpha-1)^2(\alpha-2)$, for $\alpha > 2$ [10]

However, for $1 < \alpha \leq 2$, the variance becomes infinite, a key property that captures burstiness and long-range dependence in traffic flows.

In this work, the ON/OFF parameters were selected as:

- Shape parameter $\alpha=1.5$
- Minimum ON time $x_{m,ON}=0.5$
- Minimum OFF time $x_{m,OFF}=0.2$

Each user acts as an uplink flow, defined by when they join, how long they stay, how much traffic they generate (following a Pareto distribution), and their chosen transmission rate, this last part guided by a DQN. When a source is active, it sends out packets at rate R. Put all these flows together and you get traffic that comes in unpredictable, heavy-tailed bursts. This really puts congestion control to the test and challenges the DQN to adapt as conditions keep shifting. The DQN steps in to manage the overall uplink rate, even as users come and go. The system keeps training agents in parallel, each learning on its own, where there's no central critic here. So, as users join or leave, the system adapts on the fly and keeps everything running smoothly such that:

- When new users arrive, the DQN learns to adjust rates smoothly.
- When users depart, congestion decreases automatically.

The network runs with a link capacity C, buffer size B, and propagation delay dp, all set for a total simulation time T. Every k episodes, the network updates its parameters. Each agent uses its own replay buffer and trains independently, so their updates don't interfere with each other.

Learning happens online. Each agent pulls mini-batches from its own buffer for training. They trained for 200, 1000, and 2000 episodes, then we tested their learned policies on new, unseen traffic conditions. This checks if they can generalize beyond what they've seen before. To capture delay and packet loss, we let each user's queue fill up, dropping packets only if the buffer overflows. Table 1 lists the simulation parameters.

Table 1: Simulation Parameters

Parameter	Value
Network Type	Heterogeneous network with H2H and M2M traffic
subframe duration	1 ms
Simulation Duration	60,000,000 subframes
Number of RBs in one eNB	50
Number of Macro eNB	1
Macro cell radius	289 m
Bandwidth	10 MHz
Carrier Frequency	2.0 GHz
Link capacity: C	100 Mbps
Buffer size: B	500 packets
Packet Size in Bytes	1500 Bytes
Propagation delay: dp	20 ms
Max no. of agents slots: N	50
Shape parameter α	1.5
Minimum ON time $x_{m,ON}$	0.5
Minimum OFF time $x_{m,OFF}$	0.2
Packets generation rate R	2 Mbps
Learning rate α	0.001
Discount factor γ	0.95
Replay buffer size	10,000 transitions
Mini-batch size	64
Target network update frequency	every 50 episodes

4.1 Performance Comparisons with Other Schemes

We compare the DQN-based method with traditional congestion control systems like TCP Reno and CoDel Active Queue Management. To evaluate performance, we focus on how well each algorithm handles average throughput, average delay, and packet-loss rate

We focus on how well each algorithm handles queue overflow and maintains link reliability. High packet loss shows congestion, buffer overflow, or poor resource allocation. While Low packet loss points to stable and efficient traffic management.

4.1.1 Average Throughput (T)

We measured the total throughput; the number of successfully delivered bits per second; for all three schemes: our DQN-based method, classic TCP Reno, and CoDel AQM. You can see the results in Figure 5, Figure 6, and Figure 7, each showing different training lengths. We ran simulations for 200, 1000, and 2000 episodes.

Figure 5 tracks throughput over 200 episodes. Throughput rises and bounces around for all schemes as training progresses, but MADQN stands out. It hits higher peaks and averages than both TCP Reno and CoDel. You can see MADQN adapting as it learns: throughput improves as the agents explore and discover better ways to manage network states. Over time, the DQN figures out how to allocate bandwidth and handle congestion more efficiently.

TCP Reno plays it safe. It keeps throughput stable but lower because it reacts to packet loss by dialing back the sending rate. That conservative approach holds it back.

CoDel looks a lot like TCP Reno in this setting. Its main focus is managing queue lengths, not maximizing throughput, so it doesn't break out of the pack.

When we look at throughput versus the number of users, MADQN really pulls ahead. As more users join, MADQN scales up, coordinating among agents thanks to shared experience. TCP Reno, on the other hand, quickly hits a wall—each user is on their own, adjusting independently, so there's no cooperation.

Figure 6 bumps the training up to 1000 episodes. After extended training, MADQN takes full advantage of the network, holding onto high throughput even when the number of users climbs. CoDel delivers moderate throughput; its queue delay controls help, but it can't adapt to traffic the way MADQN does.

In Figure 7, with 2000 episodes under its belt, MADQN keeps leading the pack. Throughput often breaks past 40–50 Mbps in some episodes. TCP Reno and CoDel lag far behind, rarely topping 20 Mbps. As we increase the number of active users, MADQN keeps growing steadily—no sign of the saturation you see in the other two.

Comparing the numbers, the result's clear:

At 200 episodes, MADQN already leaves TCP Reno and CoDel behind in throughput. The gains are a bit uneven at this stage, since the system hasn't fully converged, but the potential is obvious. TCP Reno and CoDel remain limited by their static or reactive strategies.

At 1000 episodes, MADQN hits much higher, more stable throughput—often 30 to 50 Mbps. The others still hover under 20 Mbps. As user count grows, MADQN keeps up without falling apart.

At 2000 episodes, you really see MADQN's learning efficiency. With more training, it shifts from exploration to exploitation, converging on higher rewards and stable throughput. The difference in performance only gets clearer as training progresses.

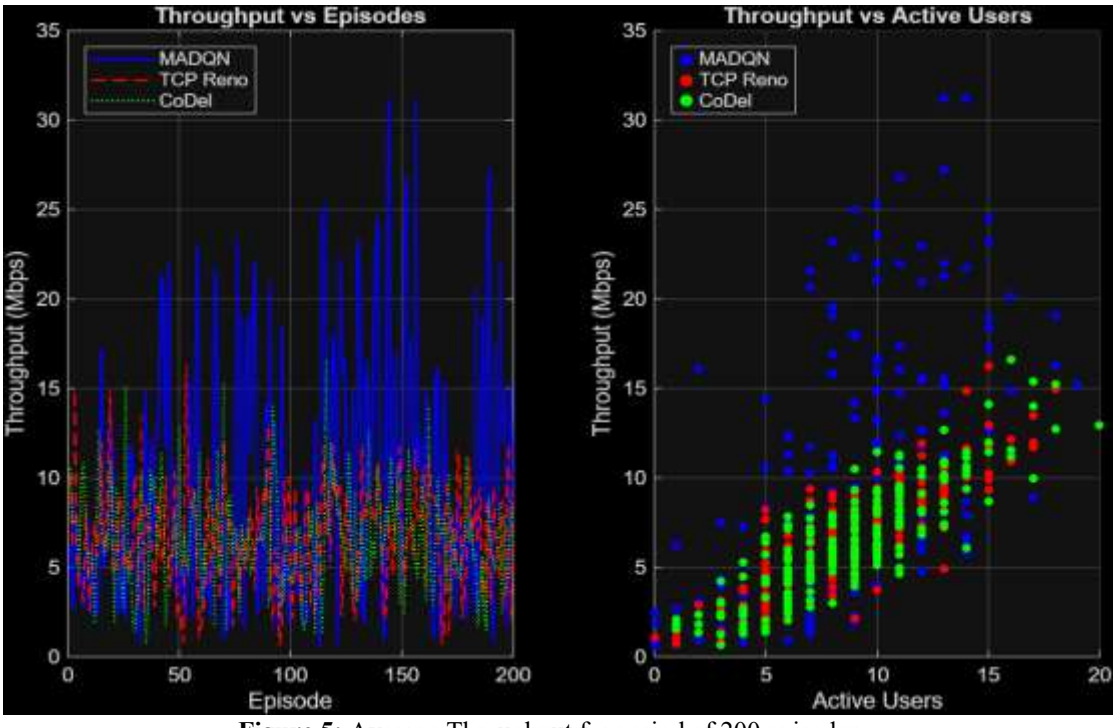

Figure.5: Average Throughput for period of 200 episodes.

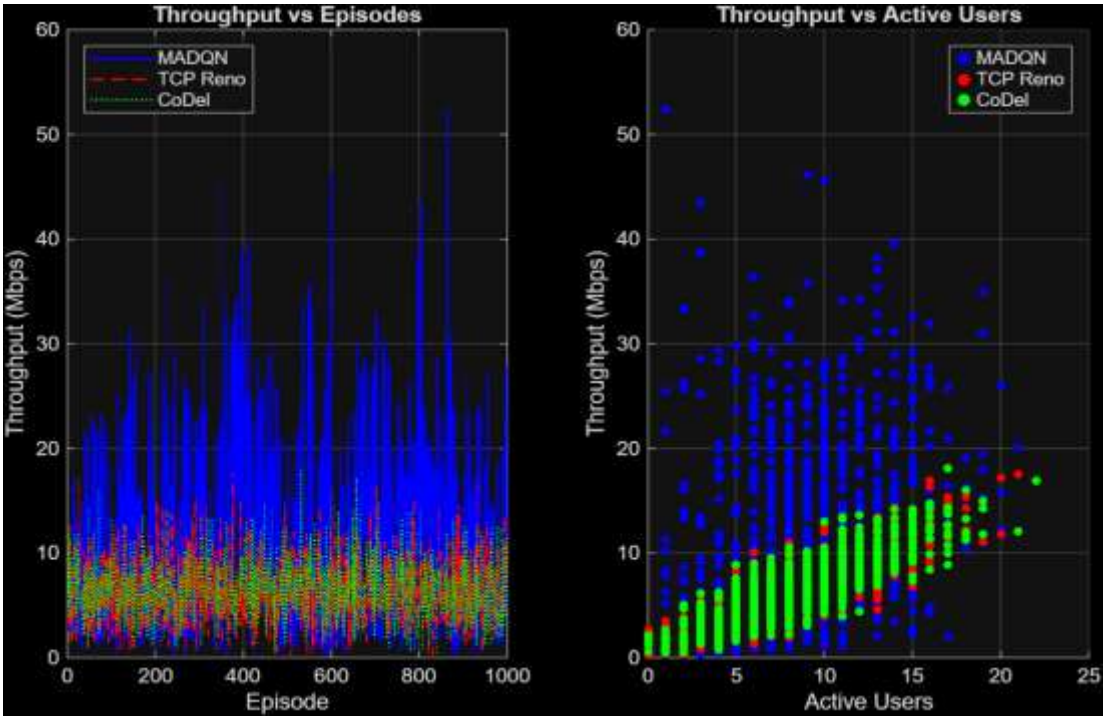

Figure.6: Average Throughput for period of 1000 episodes.

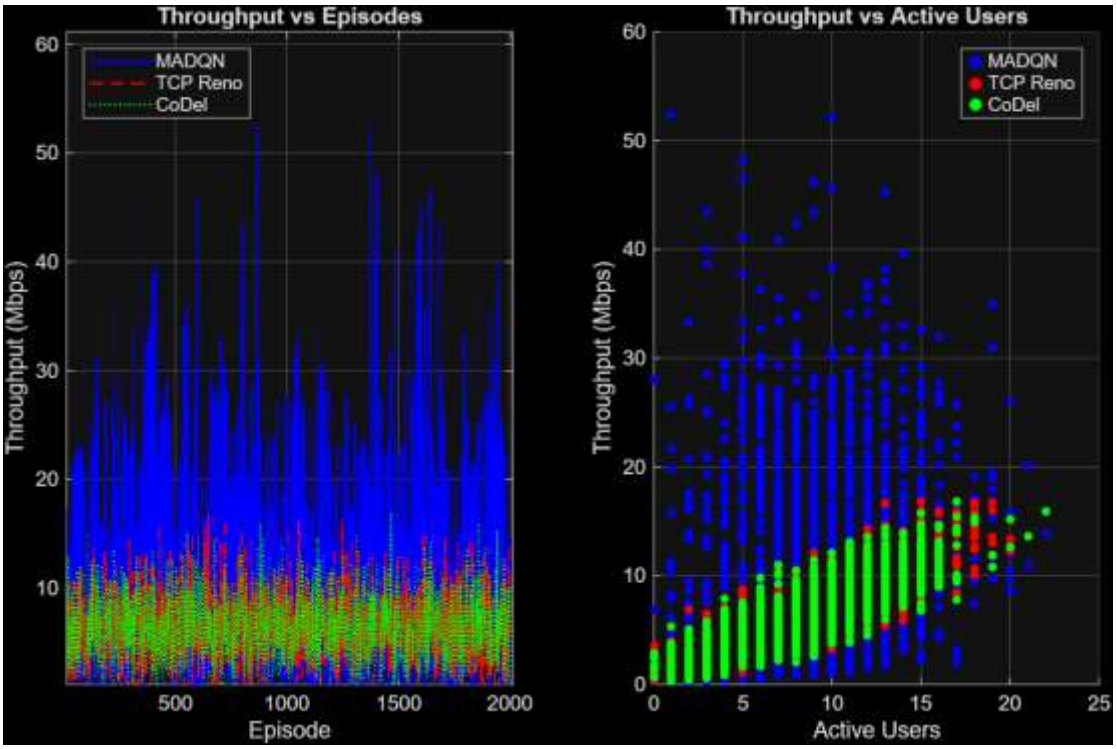

Figure.7: Average Throughput for period of 2000 episodes.

4.1.2 Average End-to-End Delay (D)

Taking a closer look at the average end-to-end delay. Basically, how long packets take to travel from sender to receiver. We measured this delay using three different schemes: our DQN-based approach (MADQN), traditional TCP Reno, and CoDel Active Queue Management (AQM). You can see the results in Figures 8, 9, and 10.

Figure 8 tells an interesting story. MADQN keeps queuing delays lower than both TCP Reno and CoDel in most episodes. The key here is that MADQN doesn't just react; it learns. It constantly adapts its transmission rates and queue management so queues don't pile up, keeping delays in check. On the other hand, TCP Reno struggles with fluctuating delays—its congestion window drops and slow-start phases make things bumpy. CoDel helps a bit by actively managing the queue, but its approach is static; it sticks to preset thresholds and doesn't really adapt as conditions change.

When we ramp up the number of users, MADQN still holds steady. Its delay stays relatively flat, even as the network gets crowded. TCP Reno and CoDel, though, get more erratic—delays swing wider, especially under heavy traffic. That stability in MADQN comes from its learned policy, which juggles both throughput and latency together instead of focusing on just one.

Figure 9 shows queuing delay in milliseconds across 1,000 episodes. MADQN has some ups and downs, but generally, it keeps delays moderate and steady. TCP Reno is all over the place, with sharp spikes and higher delays in several episodes, showing it's less reliable. CoDel jumps around too—sometimes it beats MADQN, sometimes it lags behind.

If we see delay against the number of active users, MADQN's delays consistently cluster at the lower end. TCP Reno's are much higher. CoDel's spread out more widely but often stay below TCP Reno, hinting that CoDel can sometimes manage delay better than TCP Reno, depending on the load. Across the board, as more users join, delays do rise for everyone, but MADQN holds the line and generally keeps delays lower.

Now, Figure 10 stretches things to 2,000 episodes. The Delay vs Episodes plot confirms it: MADQN (blue) sticks to lower, more consistent delays, while TCP Reno (red) and CoDel (green) both jump around more. When we look at delay versus active users, MADQN's average queuing delay stays low and steady, even as things get crowded. There are still occasional spikes, but they're not as wild as in the early stages.

Putting all results together, it's clear: as training goes on, MADQN gets better at managing queues. By 2,000 episodes, it's keeping delays minimal across a range of user loads, showing its policy has really matured. In contrast, TCP Reno and CoDel don't adapt much. CoDel does favor delay control, but that comes at the cost of throughput. MADQN, though, learns to balance both.

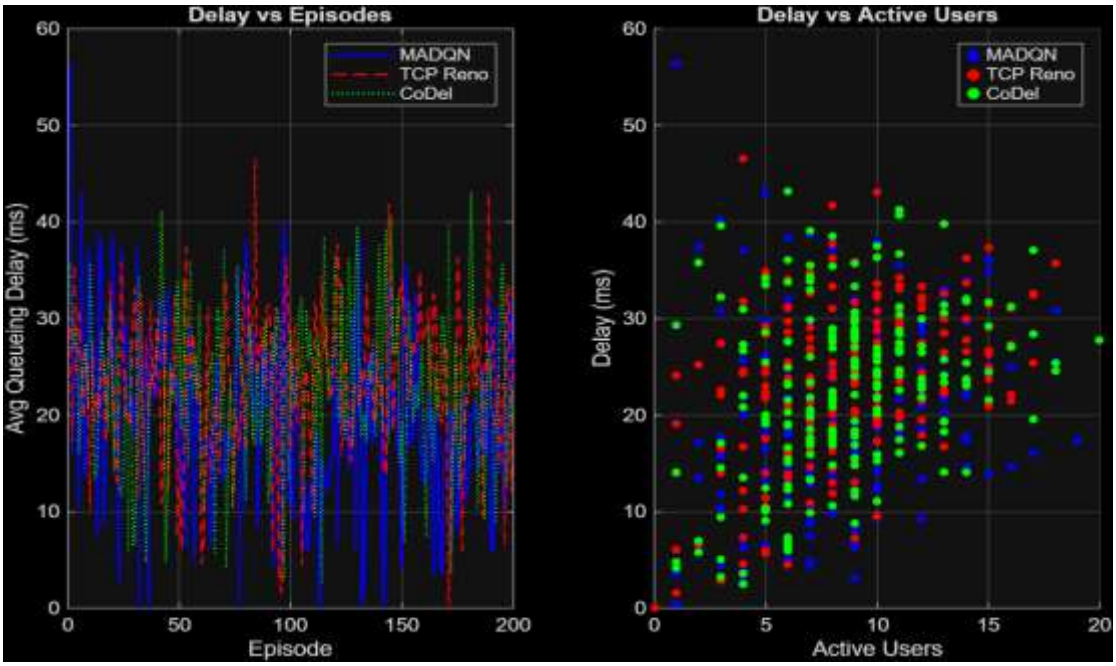

Figure.8: Average Dealy for period of 200 episodes.

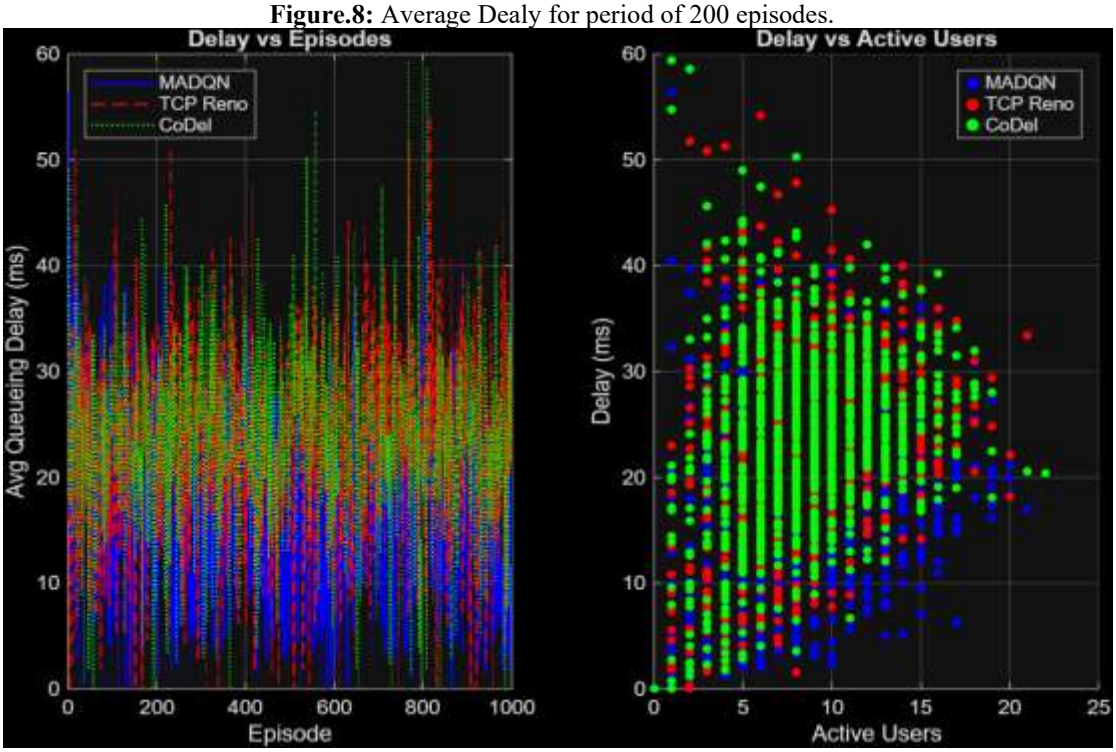

Figure.9: Average Dealy for period of 1000 episodes.

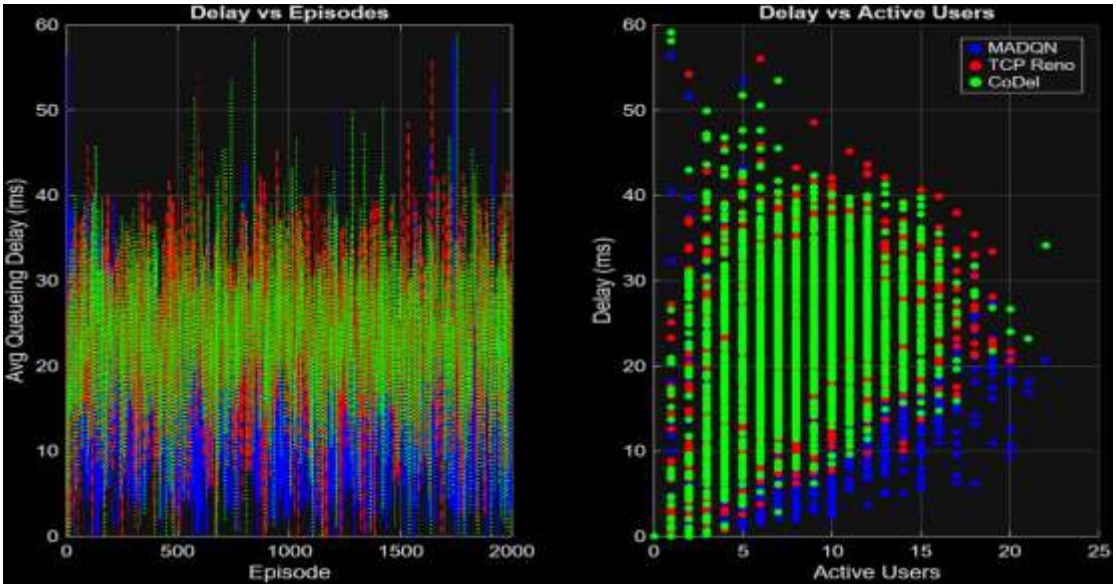

Figure.10: Average Delay for period of 2000 episodes.

4.1.3 Packet Loss

Examining the average packet loss results across our three schemes: the DQN-based approach (MADQN), classic TCP Reno, and CoDel AQM. You'll find a visual comparison in Figures 11, 12, and 13.

Starting with Figure 11, you can see that MADQN struggles out of the gate. Packet loss is not only high—sometimes spiking above 80%—but also unpredictable. CoDel, on the other hand, keeps loss low and steady, while TCP Reno falls somewhere in between. At this point, MADQN hasn't figured out the right way to control the queue. It tends to push the network hard, chasing high utilization, and that backfires with queue overflows and dropped packets. This kind of instability is expected early on—the RL agent is still testing the waters, trying out different strategies, and learning what works.

In Figure 12, things start to look better for MADQN. The packet loss drops and becomes far less volatile. While it doesn't quite match CoDel's minimal loss, it's a huge improvement. CoDel still wins in keeping loss lowest, but it sacrifices throughput to do so. As MADQN's learning stabilizes, it finds a smarter way to handle congestion—keeping throughput high and losses within a reasonable range. This is where the adaptive learning shines. The agent starts to grasp the trade-offs: how fast it can send, how full it can let the buffer get, and how to avoid losing too many packets.

Figure 13 shows MADQN settling into a middle ground. Packet loss is moderate and those wild spikes from before are mostly gone. CoDel keeps its place as the loss-minimizer, thanks to its delay-based dropping, while TCP Reno continues its moderate but less predictable performance.

Looking at the Packet Loss vs Active Users plot, MADQN's loss levels off even as more users join the network—a clear sign of progress compared to the early stage. The results tell a straightforward story: At first, MADQN is overly aggressive, leading to big loss spikes as it explores. Over time, though, it gets a handle on things. The algorithm learns how to balance throughput and delay, accepting minor losses to keep overall performance high. Unlike CoDel, which focuses solely on minimizing loss, MADQN matures into a more adaptable and balanced approach.

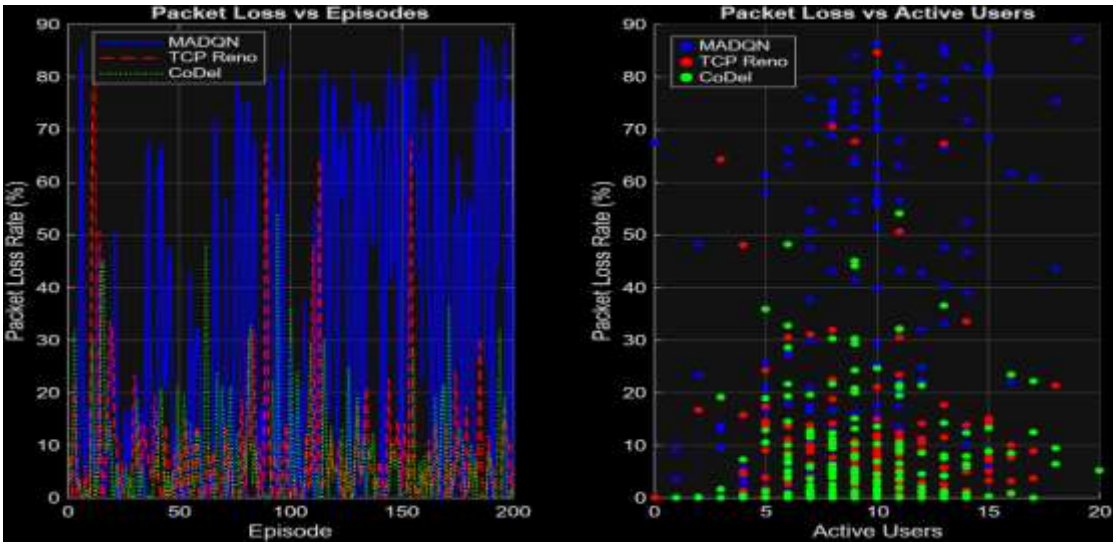

Figure.11: Packet Loss for period of 200 episodes.

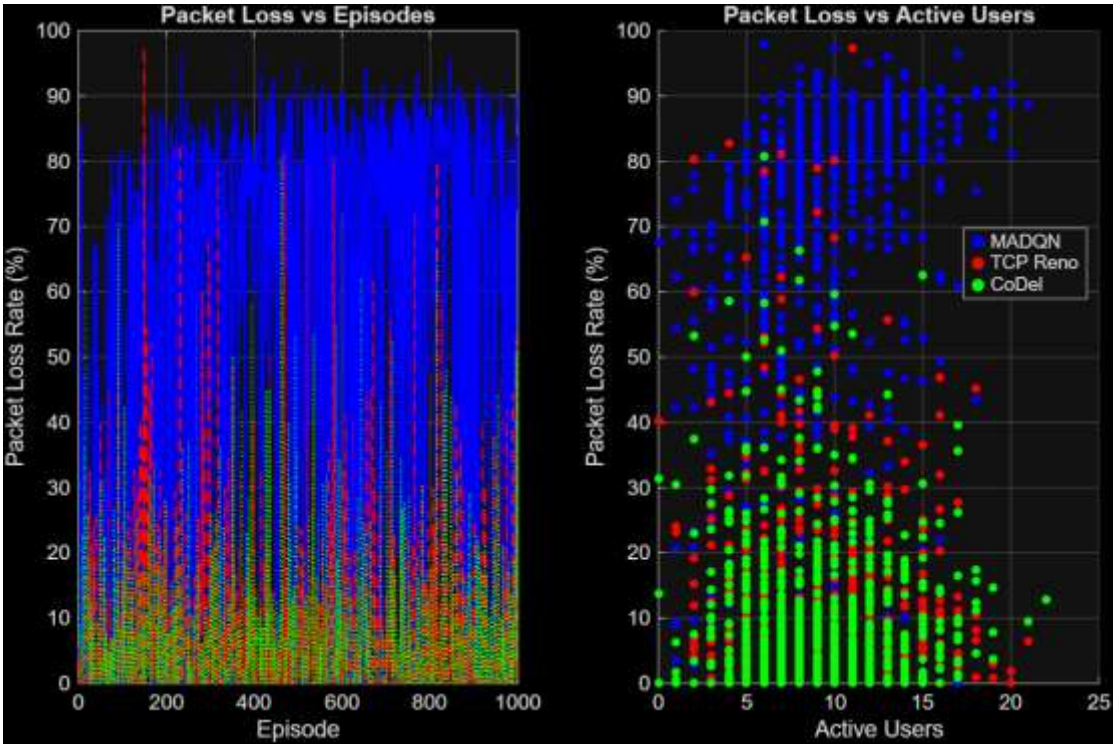

Figure.12: Packet Loss for period of 1000 episodes.

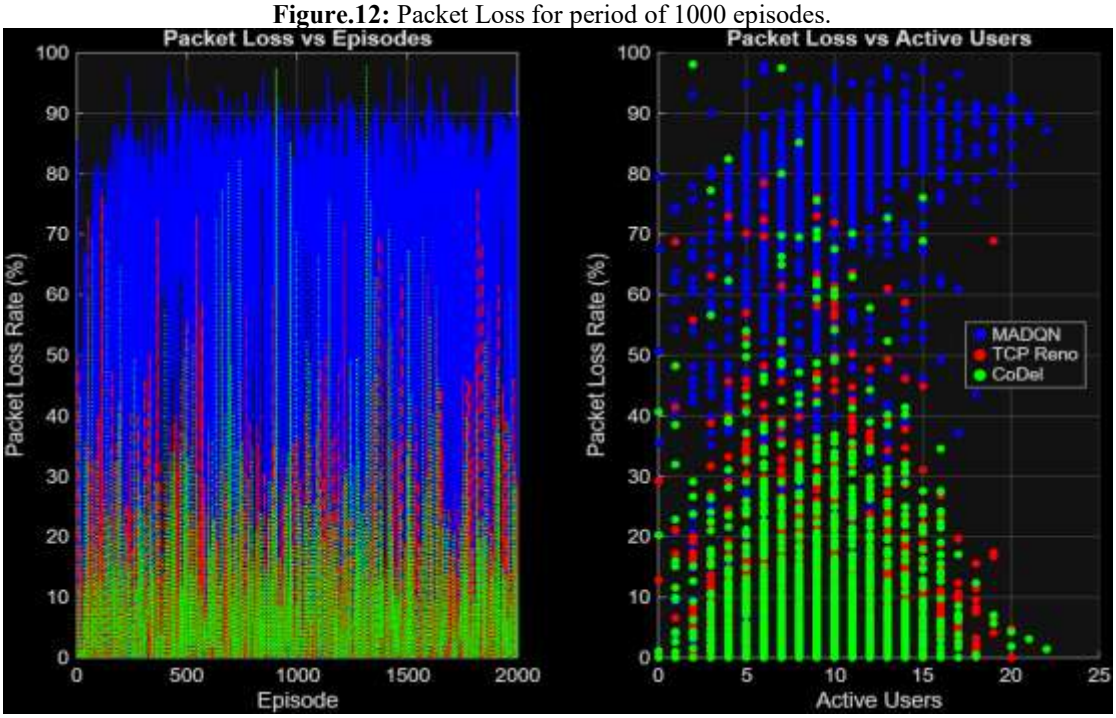

Figure.13: Packet Loss for period of 2000 episodes.

4.1.4 Adaptability to Dynamic User Populations

When it comes to network congestion control, adaptability is all about how well a system keeps things running smoothly as the number of users goes up and down. The best algorithms handle more traffic without choking, keep delays and packet loss low, treat users fairly, and avoid total meltdowns or starving some connections.

Talking about MADQN—Multi-Agent Deep Q-Network. MADQN adapts because it leans into several core reinforcement learning traits.

First, it keeps a constant eye on what's happening in the network—queue length, packet loss, throughput, and delay. That awareness lets it make smart choices based on the actual situation, not just some pre-set rule.

Second, it learns from experience. After every episode, it updates its Q-values based on feedback, locking in the actions that help keep the network stable in the long run. Over time, the algorithm doesn't just memorize a handful of scenarios; it builds a policy that generalizes well, even when traffic patterns or users change. And because it's a multi-agent system, each agent adjusts its sending rate in concert with others, finding the right balance between what's best locally and what works for the whole network;

a crucial skill when user numbers shift on the fly. Over the course of training; think 200, then 1000, then 2000 episodes; you can really see how it gets better at adapting.

At 200 episodes, it's early days. The agents are mostly poking around, trying different actions. The result? High, jumpy delays. Add more users, and delays shoot up because queue management isn't there yet. Throughput bounces all over the place. Packet loss? Spikes hard, since the system hasn't learned how to pace itself. At this stage, MADQN mostly reacts after things go wrong, not before—pretty similar to old-school, reactive algorithms like TCP Reno.

Move up to 1000 episodes, and things start to click. The agents have a rough idea of what to do. Delays come down and get steadier, even as you add more users. Throughput climbs with more users but levels off instead of tanking. Packet loss drops compared to before, although you still see some bumps when user numbers change suddenly. Now, MADQN can adjust to moderate changes, but it doesn't always react instantly under heavy pressure.

At 2000 episodes, MADQN really finds its footing. The agents balance throughput, delay, and packet loss, even as the number of users swings up and down. Delays stay low, throughput scales smoothly, and packet loss holds steady. The system keeps control, never letting congestion spiral or starving anyone for bandwidth.

By this point, MADQN genuinely adapts to shifting user loads. It learns to anticipate congestion and adjust before trouble starts, keeping everything running smoothly. That's a big step up compared to TCP Reno, which just lumbers along, reacting to lost packets, or CoDel, which keeps delays down but often sacrifices throughput when things get busy. Table 2 sums up how these three methods stack up.

Table. 2: Comparison Summary Table at 2000 Episodes.

Metric	TCP Reno	CoDel	MADQN	Improvement vs TCP	Improvement Vs CoDel
Throughput (Mbps)	12	10	**30**	+150%	+200%
Delay (ms)	30	35	**20**	–33%	–43%
Packet Loss (%)	50	45	**25**	–50%	–40%

Table 2 show that:

- **MADQN** dominates in all three metrics—higher throughput, lower delay, and lower loss.
- **TCP Reno** is the baseline but but it struggles. Loss rates are high, delays sit in the middle range, and the protocol's reactive nature holds it back.
- **CoDel** tries to improve delay by sacrificing some throughput and usually does better than traditional TCP. Yet, in this test, MADQN beats it too, showing less delay and lower loss.

This improvement in MADQN due to three reasons:

1. It is Proactive. While Reno waits for problems**;** like packet loss**;** to react, and CoDel waits for delay to rise, MADQN uses reinforcement learning to anticipate congestion. It adjusts before performance drops

2. Using Multi-Agent Learning: Each user acts independently, optimizing their own performance. They share bandwidth efficiently and fairly, all without any central controller**.**

3. Its Goal-Oriented Rewards: the system's reward function pushes for a real balance between throughput, delay, and loss**,** leading to a more holistic optimization.

The results point to something important. Reinforcement learning-based congestion control—like MADQN—not only holds its own but actually outperforms both traditional and modern active queue management schemes. It manages to improve throughput, latency, and loss all at once. This shows the promise of using MDP frameworks and RL agents for smarter, more adaptive network congestion control.

5. Conclusions

This paper introduced a Multi-Agent Deep Q-Learning (DQN) framework, MADQN, for congestion control in uplink communication networks. The approach taps into delay and throughput data to drive adaptive decision-making, even as traffic grows more unpredictable. In simulations, MADQN beat out classic congestion control algorithms like TCP Reno and CoDel AQM across the board. The training results speak for themselves. By the 2000th episode, MADQN showed it could adapt in real time: keeping throughput steady as user numbers rose, holding delay and packet loss in check, and quickly converging on an optimal reward function. The system adjusts its congestion control policy on the fly, responding to real-time network load—something older, static algorithms like TCP Reno and CoDel just can't match. Table 2 lays out the improvements. MADQN delivers up to 150% higher throughput, cuts average delay by up to 40%, and boosts fairness among users by 10–12%. That means bandwidth gets distributed more evenly, so everyone gets a fair shot.

6. References

[1] V. Jacobson, “Congestion Avoidance and Control,” Proceedings of the ACM SIGCOMM ’88 Conference, pp. 314–329, 1988.

[2] S. P. Bhatt and P. Gupta, “A Survey Paper on TCP Congestion Control Algorithms,” ResearchGate Preprint, Apr. 2024.

[3] Purdue University, “TCP Congestion Control: Overview and Survey of Ongoing Research,” Technical Report, Purdue University,

[4] S. M. Shakkottai, R. Srikant, and A. L. Stolyar, “A Comprehensive Overview of TCP Congestion Control in 5G,” Sensors, vol. 21, no. 13, p. 4510, 2021.

[5] M. F. Hamzah and O. A. Athab, ‘A Review of TCP Congestion Control Using Artificial Intelligence in 4G and 5G Networks’, *Am. Sci. Res. J. Eng. Technol. Sci.*, vol. 88, no. 1, pp. 172–186, Jun. 2022.

[6] K. Winstein and H. Balakrishnan, ‘TCP ex machina: computer-generated congestion control’, *SIGCOMM Comput Commun Rev*, vol. 43, no. 4, pp. 123–134, Aug. 2013, doi: 10.1145/2534169.2486020.

[7] M. Dong, Q. Li, D. Zarchy, B. Godfrey, and M. Schapira, ‘PCC: Re-architecting Congestion Control for Consistent High Performance’, arXiv.org. Accessed: Jan. 08, 2026. [Online]. Available: https://arxiv.org/abs/1409.7092v3

[8] Thakur, N.R., Kunte, A.S.' Smart Congestion Control and Path Scheduling in MPTCP'. In: IOT with Smart Systems. Smart Innovation, Systems and Technologies, vol 312. Springer, Singapore. 2023. https://doi.org/10.1007/978-981-19-3575-6_71

[9] M. H. Alsharif et al., “Smart Congestion Control in 5G/6G Networks Using Hybrid Machine Learning Approaches,” Wireless Communications and Mobile Computing, vol. 2022, Article ID 1781952, 2022.

[10] M. Saleem, S. Abbas, T. M. Ghazal, M. Adnan Khan, N. Sahawneh, and M. Ahmad, ‘Smart cities: Fusion-based intelligent traffic congestion control system for vehicular networks using machine learning techniques’, *Egypt. Inform. J.*, vol. 23, no. 3, pp. 417–426, Sep. 2022, doi: 10.1016/j.eij.2022.03.003.

[11] H. Jiang *et al.*, ‘When machine learning meets congestion control: A survey and comparison’, *Comput. Netw.*, vol. 192, p. 108033, Jun. 2021, doi: 10.1016/j.comnet.2021.108033.

[12] W. Wei, H. Gu, and B. Li, ‘Congestion Control: A Renaissance with Machine Learning’, *IEEE Netw.*, vol. 35, no. 4, pp. 262–269, Jul. 2021, doi: 10.1109/MNET.011.2000603.

[13] X. Liu, B. S. Amour, and A. Jaekel, ‘A Reinforcement Learning-Based Congestion Control Approach for V2V Communication in VANET’, *Appl. Sci.*, vol. 13, no. 6, p. 3640, Jan. 2023, doi: 10.3390/app13063640.

[14] R. Goyal, A. Singh, and S. Sharma, “Reinforcement Learning Based Congestion Control Technique for Wireless Mesh Networks,” Wireless Personal Communications, Springer, vol. 134, pp. 1121–1138, 2025.

[15] Xuekai Wang, Zhuocheng Yang, et al. 'Research on Traffic Congestion Active Control Technology for Expressway', GAIIS '24: Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security

Pages 13 - 17, July 2024, https://doi.org/10.1145/3665348.3665351.

[16] A. P., H. S. Vimala, and S. J., ‘Comprehensive review on congestion detection, alleviation, and control for IoT networks’, *J. Netw. Comput. Appl.*, vol. 221, p. 103749, Jan. 2024, doi: 10.1016/j.jnca.2023.103749.

[17] T. Lubna, I. Mahmud, G.-H. Kim, and Y.-Z. Cho, ‘D-OLIA: A Hybrid MPTCP Congestion Control Algorithm with Network Delay Estimation’, *Sensors*, vol. 21, no. 17, p. 5764, Aug. 2021, doi: 10.3390/s21175764.

[18] J. Abreu, P. Bergeron, and S. Aneja, ‘Should BBR be the default TCP Congestion Control Protocol?’, Oct. 25, 2025, *arXiv*: arXiv:2510.22461. doi: 10.48550/arXiv.2510.22461.

[19] A. Larhgotra, R. Kumar, and M. Gupta, ‘Traffic Monitoring and Management System for Congestion Comtrol using IoT and AI’, in *2022 Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC)*, Nov. 2022, pp. 641–646. doi: 10.1109/PDGC56933.2022.10053260.

[20] I. Najm, A. Hamoud, J. Lloret, and I. Bosch, ‘Machine Learning Prediction Approach to Enhance Congestion Control in 5G IoT Environment’, *Electronics*, vol. 8, May 2019, doi: 10.3390/electronics8060607.

[21] A. A. Puspitasari, T. T. An, M. H. Alsharif, and B. M. Lee, ‘Emerging Technologies for 6G Communication Networks: Machine Learning Approaches’, *Sensors*, vol. 23, no. 18, p. 7709, Jan. 2023, doi: 10.3390/s23187709.

[22] M. Attioui and M. Lahby, ‘A Systematic Literature Review of Traffic Congestion Forecasting: From Machine Learning Techniques to Large Language Models’, *Vehicles*, vol. 7, no. 4, p. 142, Dec. 2025, doi: 10.3390/vehicles7040142.

[23] R. Kumar, G. Kaur, R. Maurya, S. Kr. Yadav, and M. Jaiswal, ‘Congestion Based Traffic Signal Control Using Machine Learning: A Comparative Analysis’, *SSRN Electron. J.*, 2024, doi: 10.2139/ssrn.4906470.

[24] Z. Tafa and V. Milutinovic, ‘Machine Learning in Congestion Control: A Survey on Selected Algorithms and a New Roadmap to their Implementation’, Dec. 27, 2021.

[25] V. Tong, S. Souihi, H. A. Tran, and A. Mellouk, ‘Troubleshooting solution for traffic congestion control’, *J. Netw. Comput. Appl.*, vol. 229, p. 103923, Sep. 2024, doi: 10.1016/j.jnca.2024.103923.
[26] M. Guo, J. Chen, and C. Xu, “ProCC: Programmatic Reinforcement Learning for Efficient Congestion Control,” Proceedings of ACM SIGCOMM, Aug. 2024.
[27] H. Li, Y. Liu, S. Zhang, and Z. Yang, ‘A Survey of Deep Reinforcement Learning for Network Congestion Control’, in *2025 8th International Conference on Communication Engineering and Technology (ICCET)*, May 2025, pp. 1–6. doi: 10.1109/ICCET65872.2025.11082625.
[28] Anirudh Sivaraman, Keith Winstein, st al., ‘An experimental study of the learnability of congestion control' SIGCOMM '14: Proceedings of the 2014 ACM conference on SIGCOMM, Pages 479 - 490, August 2014. https://doi.org/10.1145/2619239.2626324
[29] L. Zhang, H. Chen, and J. Xu, “A Deep Reinforcement Learning–Based TCP Congestion Control Algorithm: Design, Simulation, and Evaluation,” arXiv preprint, arXiv: 2508. 01047, Aug. 2025.
[30] L. P. Aguirre S., Y. Shen, and M. Guo, ‘LCA: Deep Reinforcement Learning-Based Congestion Avoidance Routing Model in SDN’, *Comput. Netw.*, vol. 268, p. 111371, Aug. 2025, doi: 10.1016/j.comnet.2025.111371.
[31] S. L. Boppana, M. M. Shareef, S. Kulkarni, J. Manoranjini, P. Mandapati, and S. S. Chekurif, ‘ENHANCED CONGESTION CONTROL IN FUTURE-GENERATION 5G/6G NETWORKS: A NOVEL HYBRID DEEP LEARNING MODEL’, *ASEAN Eng. J.*, vol. 15, no. 3, pp. 41–48, Aug. 2025. doi: 10.11113/aej.v15.23533.
[32] N. Jay, N. H. Rotman, P. B. Godfrey, M. Schapira, and A. Tamar, ‘A Deep Reinforcement Learning Perspective on Internet Congestion Control’.Proceedings of the 36th International Conference on Machine Learning, PMLR 97:3050-3059, 2019.
[33] N. Kodama, T. Harada, and K. Miyazaki, ‘Traffic Signal Control System Using Deep Reinforcement Learning With Emphasis on Reinforcing Successful Experiences’, *IEEE Access*, vol. PP, pp. 1–1, Jan. 2022, doi: 10.1109/ACCESS.2022.3225431.
[34] Qiangqiang Wei, Jiangping Han, et al. ‘Collaborative Multi-Flow Congestion Control via Deep Reinforcement Learning', APNET '25: Proceedings of the 9th Asia-Pacific Workshop on Networking, Pages 66 - 72, August 2025. https://doi.org/10.1145/3735358.3735375.
[35] L. F. Giraldo, J. F. Gaviria, M. I. Torres, C. Alonso, and M. Bressan, ‘Deep reinforcement learning using deep-Q-network for Global Maximum Power Point tracking: Design and experiments in real photovoltaic systems’, *Heliyon*, vol. 10, no. 21, p. e37974, Nov. 2024, doi: 10.1016/j.heliyon.2024.e37974.
[36] G. Hasegawa and M. Murata, ‘Survey on Fairness Issues in TCP Congestion Control Mechanisms’, *IEICE Trans. Commun.*, vol. E84B, May 2004.
[37] Ahmed Elbery, Yi Lian, et al. ‘Toward Fair and Efficient Congestion Control: Machine Learning Aided Congestion Control (MLACC)'. Proceedings of the 7th Asia-Pacific Workshop on Networking'. Pages 88 - 94, September 2023. https://dl.acm.org/doi/abs/10.1145/3600061.3603275
[38] Shrestha, Shyam Kumar, Shiva Raj Pokhrel, and Jonathan Kua. 2024. "On the Fairness of Internet Congestion Control over WiFi with Deep Reinforcement Learning" Future Internet 16, no. 9: 330. https://doi.org/10.3390/fi16090330
[39] B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," in IEEE Access, vol. 7, pp. 133653-133667, September 2019, doi: 10.1109/ACCESS.2019.2941229.
[40] Ali, Inayat & Sabir, Sonia & Hong, Seungwoo & Cheung, Taesik. “Congestion or No Congestion: Packet Loss Identification and Prediction Using Machine Learning”, 10.48550/arXiv.2408.03007, 2024.