CongestRL: Reinforcement Learning-Based Congestion Control in Virtual Network Simulation

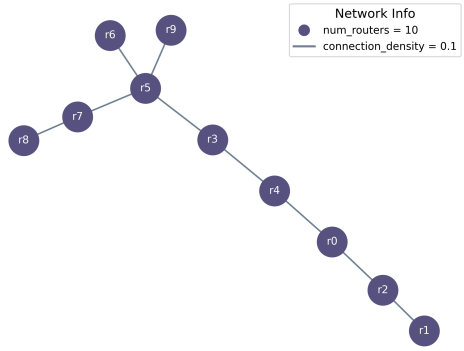
Aimilia Palaska 10453 - aimiliapm@ece.auth.gr
Academic Supervisor: Anastasios Tefas

May 30, 2025

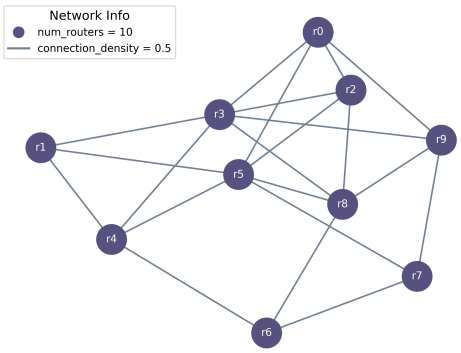
1 Introduction

This report is created as part of the Computational Intelligence & Deep Reinforcement Learning course in Electrical and Computer Engineering department of Aristotle University of Thessaloniki. The developed code in Python, along with detailed comments, can be found in the emily-palaska/CongestRL GitHub Repository.

Congest-RL presents a virtual network simulation framework designed to manage congestion and delay in packet exchange using multithreading and reinforcement learning policies. The system aims to simulate realistic network conditions while allowing agents to learn effective congestion control strategies. Section 2 outlines the core architecture of the simulation, detailing how routers, links, and packet dynamics are modeled. Section 3 describes the transformation of this simulation into a reinforcement learning environment, enabling seamless integration with learning agents. Section 4 introduces three distinct reward functions crafted to guide policy behavior based on congestion and delay signals. Section 5 presents the implemented policies, which include a random baseline, a Deep Q-Network (DQN), and a Proximal Policy Optimization (PPO) agent. Section 6 provides the results of a series of experiments designed to evaluate and compare these policies under various conditions. Finally, Section 7 discusses conclusions, current limitations of the framework and suggests potential directions for future development.



(a) Connection density 0.1



(b) Connection density 0.5

Figure 1: Graph examples

2 Network

2.1 Graph

The `CongestNetwork` is represented as a connected, undirected graph $G(R, E)$, where R denotes routers (nodes) and E the weighted connections (edges). A direct packet transfer between routers $r_i, r_j \in R$ is possible if and only if $(r_i, r_j) \in E$; otherwise, an indirect path must be computed. The graph's connectivity ensures the existence of a route between any two routers. Moreover, topology is static, but edge weights evolve to represent traffic load—higher weights imply greater congestion. Routing decisions aim to minimize congestion by selecting paths with lower cumulative weights.

The network's connectivity is governed by the `connection_density` parameter. During initialization, edges between routers are formed through a stochastic process: for each pair of routers, a value is sampled from the uniform distribution $\mathcal{U}(0, 1)$. An edge is created if this value is less than the specified `connection_density`. As a result, higher values of `connection_density` increase the likelihood of edges being added, though the exact number of edges remains uncertain due to the probabilistic nature of the process. This design ensures

that the graph topology varies across episodes, introducing non-determinism and promoting a richer diversity of network architectures for a robust learning pipeline. Two example Graphs with different densities are presented in Figure 1.

2.2 Routing

Routers are implemented via the `Router` class in `simulation.routing`. Each router serves a fixed number of users, distributed evenly and defining the packet generation capacity per timestep. Routers handle both locally generated packets and those forwarded en route to other destinations. They possess global knowledge of the network topology and edge weights, facilitating optimal routing—consistent with link-state protocols used in modern Internet architectures.

Each user may transmit one packet per timestep via its associated router. Packets are defined in `core.packets` and follow a standardized structure (Figure 2). Fields include source and destination router IDs, determined via a probabilistic function in `core.statistics` that models user activation with a default normal distribution $\mathcal{N}(0, 1)$. If activated, destination routers are selected uniformly at random. Upon creation, packets are assigned a best path using Dijkstra's algorithm (via the `core.graphs` module and `NetworkX`) [1], although alternative shortest-path heuristics are also supported. While path updates are not dynamic, the relatively short path lengths (max six hops) justify this simplification. Each packet has a weight sampled from $\mathcal{U}(1, 10)$, contributing to congestion metrics on traversed edges. Timestamps are included to enable delay analysis.

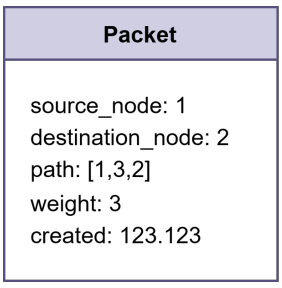


Figure 2: Packet structure

To emulate the autonomous behavior of routers, `CongestNetwork` leverages Python's `Threading` and `Queue` modules [2]. Each router runs on a dedicated thread with an incoming and buffer queue. Neighboring routers insert packets into each other's incoming queues, constrained by graph connectivity. Lastly, the routers are synchronized by shared references to `Events`, one for freezing and one for halting.

Each thread executes the following loop:

- Initiates user packet generation, appending to the buffer queue.
- Transfers incoming packets to the buffer.
- Demultiplexes packets by: (i) consuming packets destined for the current router (computing delay); (ii) forwarding others to their next-hop based on precomputed paths; and (iii) accumulating edge weights for congestion metrics.

The number of packets processed per timestep is controlled by the router's `send_rate`, constrained by a configurable `max_send_rate`. These parameters define the RL action space, elaborated in Section 3.

2.3 Evaluation Metrics

Congestion control is assessed via two primary metrics: congestion level and delay. The former quantifies global network load and is defined as the cumulative weight of all edges at a given timestep:

$$c(t) = \sum_{e \in E} w_e$$

Here, E denotes the set of graph edges and w_e the weight of edge e, corresponding to the sum of packet weights delivered through e in the latest exchange. This value reflects transient network load and is compared against a configurable congestion limit, which constrains aggregate traffic. The threshold can be relaxed or tightened to examine system behavior under varying conditions.

To avoid trivial minimization of congestion by excessively throttling traffic, delay is introduced as a counterbalancing metric. Delay is computed per packet as the time elapsed between its creation and reception at the destination router. Specifically, when a packet reaches its destination (i.e., the router ID matches), the timestamp difference is recorded. Each router maintains a rolling buffer of 1000 delay entries, limiting memory usage and constraining analysis to recent activity. This dual-metric framework ensures policies optimize not only for stability but also for timeliness in packet delivery.

Calculating delay time presented several challenges. In a trivial case where the network runs continuously, delay can be computed simply as the difference between the packet's creation and reception timestamps. However, in a simulation setting, particularly during agent training involving forward and backward passes through neural networks, the network must be paused between steps. This introduces non-negligible computational delays in an environment where timing precision is critical.

To address this, the `CongestNetwork` class maintains a record of the last 100 active periods, defined as the intervals during which the network

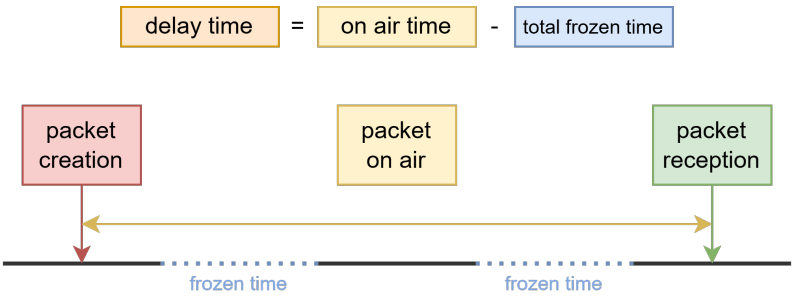


Figure 3: Delay time calculation

is actively running (i.e., between the start and end of each step). For each packet, the raw difference between reception and creation timestamps is referred to as the *on-air time*. If both events occur within the same active period, this on-air time is equivalent to the actual delay. However, if creation and reception span multiple active periods, the delay must exclude the time during which the network was frozen. This *frozen time* is computed by summing the intervals between the end of one active period and the start of the next.

A visual representation of this mechanism is provided in Figure 3. Removing frozen time from delay calculations improves network stability, as it mitigates the inflated delay values often observed in the first packets of a newly resumed active period, which would otherwise distort the agents' learning signals.

3 Environment

The `CongestionControlEnv` class implements a custom environment for training and evaluating RL agents in congestion control scenarios. Built upon OpenAI's `gym` framework [3], the environment integrates with the `CongestNetwork` simulation and exposes a modular interface for agent interaction.

Key parameters include:

- `congestion_limit`: upper bound for total edge weight, exceeding this indicates network overload.
- `reward_func`: defines the reward signal, configurable and detailed in Section 4.
- `network`: an instance of `CongestNetwork` used to execute network dynamics.
- `step_time`: simulation duration (in seconds) per environment step.
- `metadata`: dictionary encapsulating environment hyperparameters for logging and reproducibility.

Resetting the `CongestionControlEnv` initializes a new graph for the network, with the same connection density. This aspect is crucial to avoid overfitting by the agents, since the topology between training episodes is stochastic.

3.1 Observation Space

The observation vector encodes metrics relevant to the agent's decision-making. Chief among these is the congestion trace, a temporal sequence of congestion values sampled every 0.1 seconds, maintaining the last 25 samples (i.e., 2.5 seconds of data). This windowed sampling reduces the impact of transient outliers.

Complementing this is the average packet delay, computed per router from the most recent 1000 delay entries. Together, these features constitute the observation space $\mathcal{O} \subseteq \mathbb{R}^n$, where n reflects the dimensionality determined by the routers' number and the congestion samples. Both congestion and delay values are unbounded above, modeling a continuous and non-clipped observation domain.

3.2 Action Space

The action space $\mathcal{A}$ corresponds to router-specific adjustments of the `send_rate` parameter, which determines the number of packets dequeued and forwarded per timestep. Specifically, actions are vectors of offset values in $[-1000, 1000]$, clipped to maintain feasibility within each router's `max_send_rate`. This design enables smooth rate modulation and stabilizes training by preventing erratic policy updates.

Alternative formulations, including discrete adjustment schemes and direct `send_rate` specification, were evaluated. However, discrete actions restrict fine-grained control, and absolute rate outputs induced instability due to oscillatory behavior during early learning phases. The offset-based approach provides a balanced mechanism for convergence toward stable operating configurations.

3.3 Step Function

The `step(action)` method orchestrates environment transitions. Given an action vector, it up-

dates each router's `send_rate`, applies boundary clipping, and advances the simulation for `step_time` seconds. Upon completion, it extracts the latest observation, evaluates the reward signal, and returns a tuple comprising the observation, reward, and an auxiliary dictionary. This dictionary includes the applied action, per-router congestion and delay statistics, and the updated send rates—supporting agent training and offline analysis.

4 Reward

4.1 Linear Reward Function

The linear reward function is defined in Equation 1, where $\mathbf{c}$ is the list of congestion values, $\mathbf{d}$ is the list of delay values, L is the congestion limit, and α, β are weighting parameters. The function balances two main components: the average delay and the individual congestion levels across routers. The congestion reward component $r(c, L, \beta)$ is defined piecewise as follows:

$$r_l(c, L, \beta) = \begin{cases} \beta & \text{if } c < 0.6L \\ 4\beta - \frac{5\beta}{L}c & \text{if } 0.6L \leq c < L \\ -4\beta & \text{if } c \geq L \end{cases}$$

This function provides a simple yet effective mechanism to reward network behavior. It assigns a positive reward when the congestion level is under 60% of the limit L, linearly decreases the reward as congestion approaches the limit, and penalizes the agent heavily when congestion exceeds the limit.

The delay term acts as a regularizer, discouraging unnecessarily high delays even if congestion levels are low. However, its contribution to the total reward is modulated by the parameter α, allowing for emphasis on delay minimization to be tuned independently of congestion penalties governed by β.

This form of reward encourages cooperative behavior among agents, pushing them to remain under congestion thresholds while maintaining acceptable latency. Sweeps over different α and β values reveal how the tradeoff between delay and congestion can be modulated, as shown in Figure 5.

4.2 Quadratic Reward Function

The quadratic reward function (Equation 2) introduces a nonlinear penalty on the average delay, emphasizing stability and fairness in delay-sensitive scenarios. The first term penalizes high average delays quadratically, making extreme values more impactful in the reward signal. The congestion reward component is crafted to ensure smoothness and differentiability in the intermediate congestion regime. It is parameterized as follows:

$$\begin{aligned} k &= 10^{-2}/L \\ l &= -\frac{2\beta + 0.64kL^2}{0.4L} \\ m &= 4\beta + 0.6kL^2 \end{aligned}$$

and the piecewise definition is:

$$r_{\text{q}}(c, L, \beta) = \begin{cases} \beta & \text{if } c < 0.6L \\ kc^2 + lc + m & \text{if } 0.6L \leq c < L \\ -4\beta & \text{if } c \geq L \end{cases}$$

The quadratic term creates a convex reward landscape for delay, encouraging agents to optimize toward minimal latency with increasing sensitivity. Simultaneously, the congestion component provides a smooth transition from positive to negative rewards, aiding in stable learning. The parameters are derived to ensure continuity at the piecewise boundaries. Experimental sweeps using this function are shown in Figure 6.

4.3 Exponential Reward Function

The exponential reward function (Equation 3) introduces a saturating delay penalty, ideal for scenarios where marginal improvements in low-delay settings are more valuable than equal improvements in high-delay ones. This delay formulation yields high rewards when average delay is near zero, and quickly diminishes as the delay increases, reflecting a preference for ultra-low-latency behavior.

The congestion component remains consistent with the linear case:

$$r_{\text{exp}}(c, L, \beta) = \begin{cases} \beta & \text{if } c < 0.6L \\ 4\beta - \dfrac{5\beta}{L}c & \text{if } 0.6L \leq c < L \\ -4\beta & \text{if } c \geq L \end{cases}$$

This reward formulation is particularly useful when operating in regimes where congestion is expected to be under control and the system is optimized for responsiveness. The exponential delay term allows the agent to focus heavily on the lowest latency paths early during training. Parametric experiments using this reward structure are illustrated in Figure 7.

5 Policy

The `CongestRL` mechanism implements three types of policies in order to decide on actions and regulate congestion and delay. The Baseline method is used as a reference point for an unregulated network and operates on a random selection from the action space. The other two are a DQN agent and a PPO policy, analyzed further in this following subsections.

$$R(\mathbf{c}, \mathbf{d}, L, \alpha, \beta) = -\alpha \cdot \frac{1}{|\mathbf{d}|} \sum_{d \in \mathbf{d}} d + \sum_{c \in \mathbf{c}} r(c, L, \beta) \tag{1}$$

$$R_{\text{quad}}(\mathbf{c}, \mathbf{d}, L, \alpha, \beta) = -\alpha \left(\frac{1}{|\mathbf{d}|} \sum_{d \in \mathbf{d}} d \right)^2 + \sum_{c \in \mathbf{c}} r_{\text{quad}}(c, L, \beta) \tag{2}$$

$$R_{\text{exp}}(\mathbf{c}, \mathbf{d}, L, \alpha, \beta) = -\alpha + \alpha \cdot \exp\left(-\frac{1}{|\mathbf{d}|} \sum_{d \in \mathbf{d}} d \right) + \sum_{c \in \mathbf{c}} r_{\text{exp}}(c, L, \beta) \tag{3}$$

Figure 4: Reward function formulas

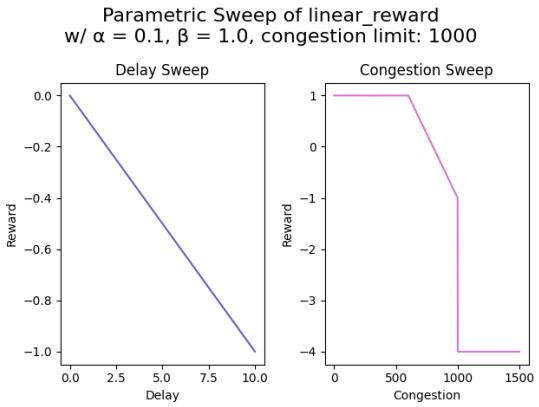


Figure 5: Linear Reward Sweep

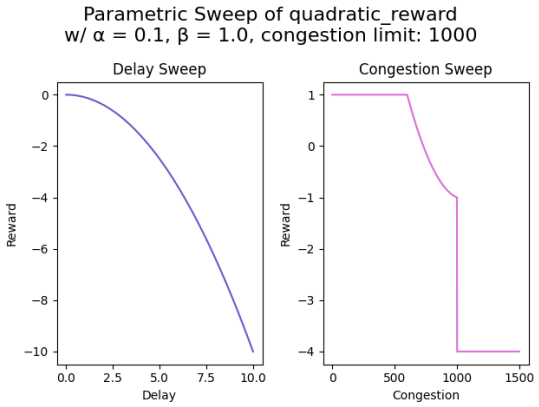


Figure 6: Quadratic Reward Sweep

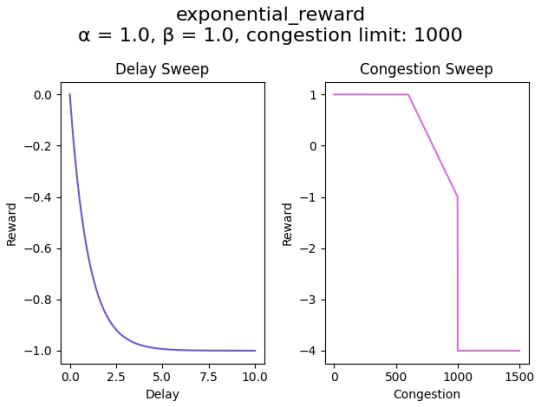


Figure 7: Exponential Reward Sweep

5.1 DQN Agent

The *Deep Q-Network (DQN)* agent is a reinforcement learning algorithm that leverages deep neural networks to approximate Q-values, which represent the expected cumulative reward for taking a particular action in a given state. In traditional Q-learning, the Q-table becomes impractical for large or continuous state spaces; DQNs address this limitation by using a neural network to estimate these values.

The DQN agent was implemented as a `torch.nn.Module` [4], with two key parameters: `state_size` and `action_size`, representing the dimensions of the input observation space and the output action space, respectively. The network consists of a single hidden layer with 128 linear neurons followed by a ReLU [5] activation function. Given the compact architecture, and to maintain simplicity, the model runs entirely on the CPU, with no need for GPU parallelization.

The output of the network is a vector of Q-values corresponding to each possible action. However, in this environment, the action space is continuous. To handle this, the Q-value for the selected action is computed as the dot product between the predicted Q-values q_v and the actual action a vector provided to the environment:

$$q_c = q_v \cdot a$$

This approach effectively weights each output dimension by its corresponding action value, allowing the agent to adapt to the continuous nature of the control problem.

The agent is trained using the Mean Squared Error (MSE) loss and optimized with the Adam optimizer [6], both standard choices for stability and efficient convergence. To encourage exploration of the environment, the agent uses an ϵ-greedy policy: at each time step, with probability ϵ, a random action is selected (exploration), and with probability $1-\epsilon$, the action with the highest predicted Q-value is chosen (exploitation). The value of ϵ decays over time, gradually shifting the agent's behavior from exploration to exploitation as training progresses.

For experience replay, the agent maintains a memory buffer with a size equal to the batch size.

At each learning step, a batch of experiences is sampled randomly from this buffer, helping to break correlations between consecutive transitions and stabilize training.

All hyperparameters used in the agent, including the learning rate, discount factor, epsilon decay rate, batch size, and others, are stored in the `metadata` dictionary for reproducibility and experiment tracking. This setup enables easy comparison between different runs and tuning of configurations for optimal performance.

5.2 PPO Agent

The *Proximal Policy Optimization (PPO)* algorithm is a state-of-the-art policy gradient method designed for reinforcement learning tasks with continuous or discrete action spaces. PPO improves the stability and efficiency of training by introducing a clipping mechanism that limits the deviation of the new policy from the old one during updates. This constraint helps prevent overly aggressive policy updates that can destabilize learning.

PPO belongs to the family of actor-critic methods, where two separate components are trained: the actor, which proposes actions given states, and the critic, which evaluates the expected return from a given state. The actor learns a stochastic policy by outputting a distribution over actions, while the critic approximates the value function used for computing advantages.

In this implementation, both the actor and critic share the same architecture as the one used in the DQN Agent. Specifically, they are neural networks composed of a single hidden layer with 128 linear neurons followed by a ReLU [5] activation function. The output of the actor is used to parameterize a probability distribution over actions, while the critic outputs a scalar value estimating the state value. The model is trained using the Adam optimizer [6], known for its efficiency and adaptive learning rate capabilities.

The agent selects actions by first transforming the environment state into a tensor format and passing it through the actor network to obtain a probability distribution. An action is then sampled from this distribution and clipped to remain within predefined bounds. The logarithmic likelihood of the action is also computed and stored for later use.

To compute the training targets for the critic, the agent walks backward through the stored rewards and next states, using the critic to estimate the value of the next state and applying the Bellman equation recursively. This generates the expected return from each state, which serves as the target for learning.

When updating its networks, the agent retrieves the stored experiences and computes the advantage estimates by subtracting the current critic's predictions from the target returns. It then recalculates the action probabilities using the actor and compares them to the stored probabilities from earlier. A ratio of these probabilities is calculated, and the minimum of two surrogate objectives is used to prevent large deviations from the old policy.

The final loss consists of two parts: one from the actor, which encourages high-probability actions with high advantage, and one from the critic, which minimizes the mean squared error between predicted and target values. The combined loss is backpropagated, and the optimizer updates the model parameters accordingly over multiple steps, as specified by the hyperparameters.

6 Experiments

All experiments were conducted for 10 episodes, each lasting 200 steps. To ensure comparability across methods, the parameters of the agents and the environment, unless specified otherwise, are summarized in Tables 1, 2, and 3.

Parameter	**Value**
num_routers	10
num_users	50
step_time	5
alpha	0.1
beta	1.0
reward_func	linear
congestion_limit	10,000
connection_density	0.1

Table 1: Environment Default Parameters

Parameter	**Value**
epsilon_start	1.0
epsilon_min	0.05
epsilon_decay	0.995
gamma	0.99
lr	0.0001
batch_size	8

Table 2: DQN Default Parameters

Parameter	**Value**
clip_epsilon	0.2
lr	0.0003
gamma	0.99
update_steps	10
batch_size	8

Table 3: PPO Default Parameters

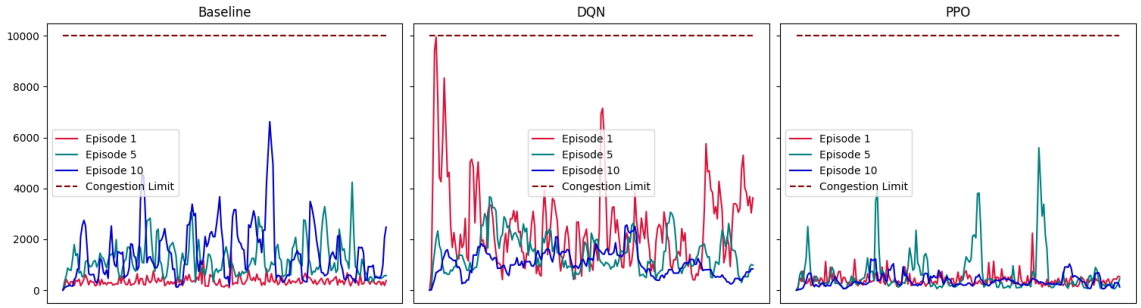


Figure 8: Congestion Mean in 3 distinct Episodes

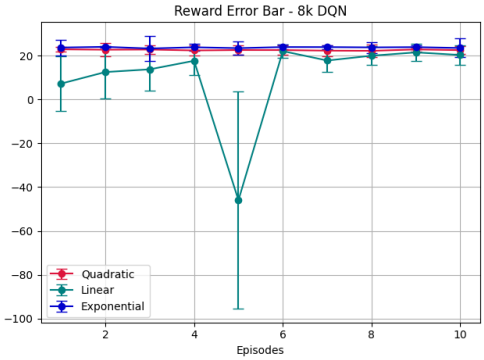


Figure 9: Reward Function Comparison

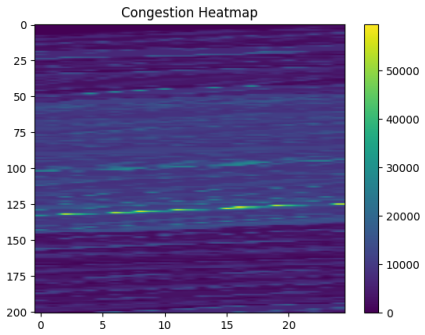


Figure 10: Congestion Heatmap DQN 8k, Episode 5

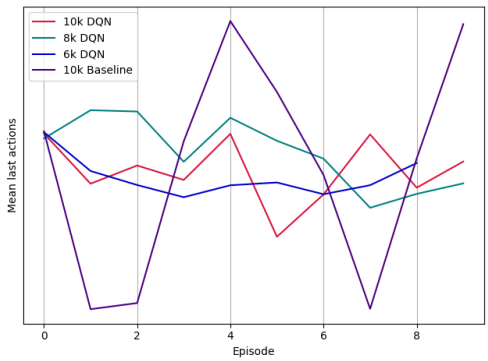


Figure 11: Mean Action at different Congestion Limits

6.1 Policy Comparison Under Fixed Congestion Limit

In the first experiment, the three policies were tested using the same environment configuration with a congestion limit of 10,000. Figure 8 presents the average congestion per step for Episodes 1, 5, and 10. The baseline policy shows unregulated behavior, starting from nearly zero congestion in the first episode and escalating to values around 4000 by Episode 10. Although the congestion limit was never exceeded, the trend indicates uncontrolled dynamics.

The DQN policy, in contrast, initially violates the limit but quickly stabilizes, showing more regular behavior by Episode 5. This suggests that the agent identifies the congestion threshold at which rewards begin to diminish. The PPO policy appears to minimize congestion more aggressively, although this could also result from favorable initializations that promote early packet exchanges.

6.2 Effect of Reward Function on DQN Performance

In the second experiment, the DQN agent was evaluated using three different reward formulations. Figure 9 illustrates the episode-wise mean and standard deviation of total rewards. Despite the randomized graph initialization in each episode, the DQN agent generalizes well. The linear reward function exhibits slower performance with an outlier in Episode 5. To better understand this anomaly, we visualized the congestion heatmap of that episode in Figure 10. The results confirm a significantly higher congestion level—ranging from 30,000 to even 50,000—compared to other episodes. Nonetheless, such anomalies can be valuable for learning robust policies.

6.3 DQN Behavior Across Varying Congestion Limits

The next experiment assessed how the DQN agent adapts to varying congestion limits (6k, 8k, and 10k). Figure 11 shows the average of

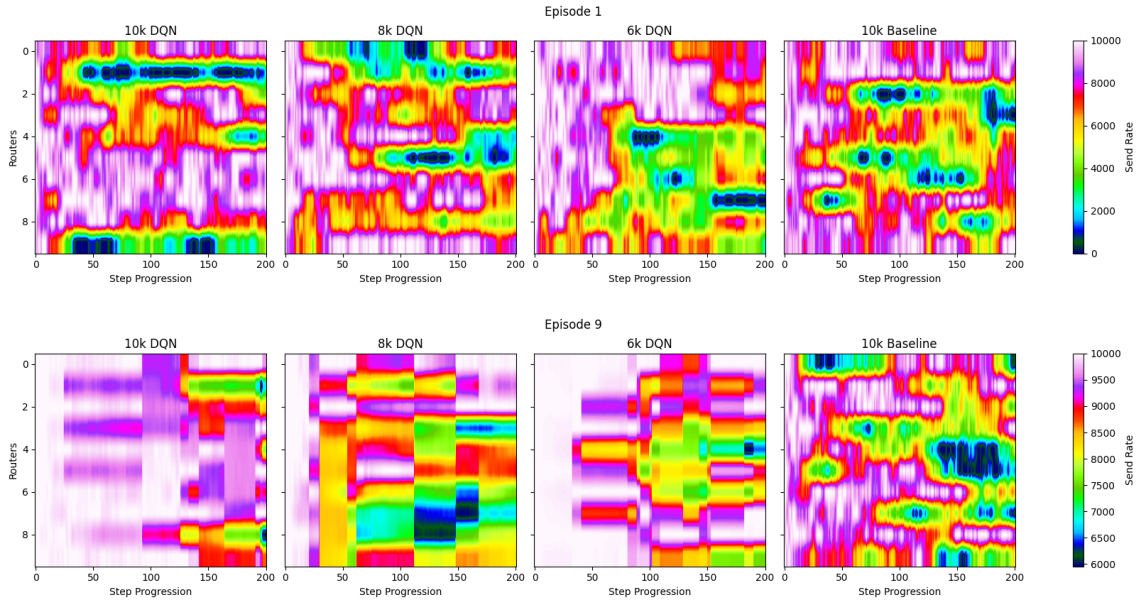


Figure 12: Send rates for DQN and Baseline

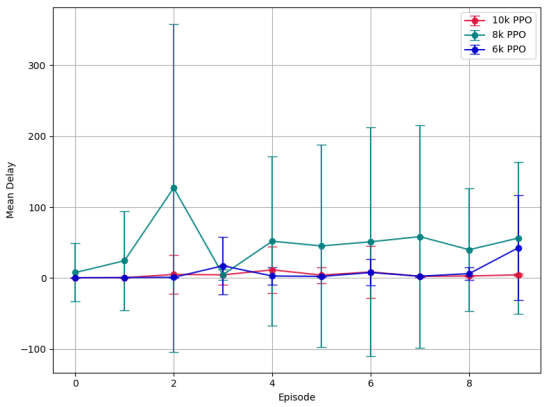


Figure 13: Delay Error Bar, Congestion Limit Sweep

the final actions taken by the agent in each setting, along with the baseline for reference. The baseline's randomness is evident through erratic decision patterns. In contrast, the DQN policy exhibits smoother trends, with oscillations increasing in frequency as the congestion limit rises.

Further analysis is provided in Figure 12, which shows send rate heatmaps for both the baseline and DQN policies in Episodes 1 and 10. Initially, all policies start from random states. However, by Episode 10, the DQN agent adapts to the congestion constraints, forming discernible patterns across routers. Lower congestion limits yield more conservative send rates, while higher limits allow for more aggressive policies.

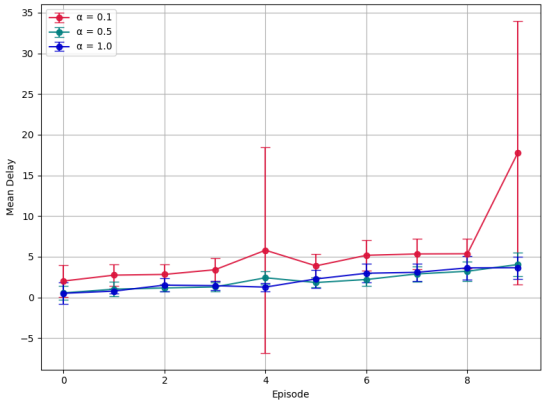


Figure 14: Delay Error Bar, DQN Linear Reward α Sweep

6.4 PPO Policy Under Different Congestion Limits

Following the DQN experiments, the PPO agent was subjected to the same congestion configurations (10k, 8k, and 6k). This time, the focus was on delay metrics. Figure 13 shows that higher congestion limits correspond to lower delays, since the agents can afford to send more packets without surpassing the threshold. The configuration with a 6k limit behaves consistently, while the 8k limit scenario shows considerable variability and frequent outliers. This suggests that the PPO agent, with the current reward function, does not directly account for delay and thus exhibits more random behavior in that metric.

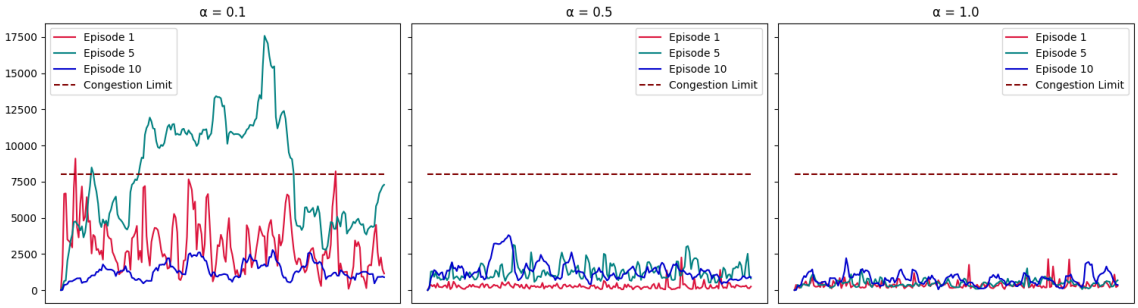


Figure 15: Congestion Mean, DQN Linear Reward α Sweep

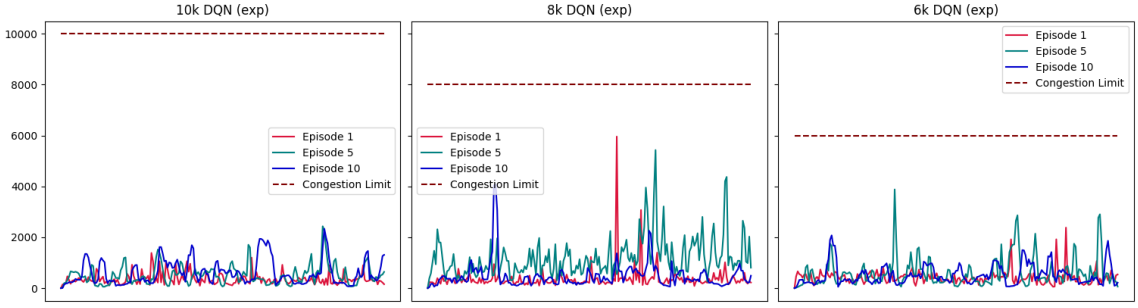


Figure 16: Congestion Mean, DQN Exponential Reward

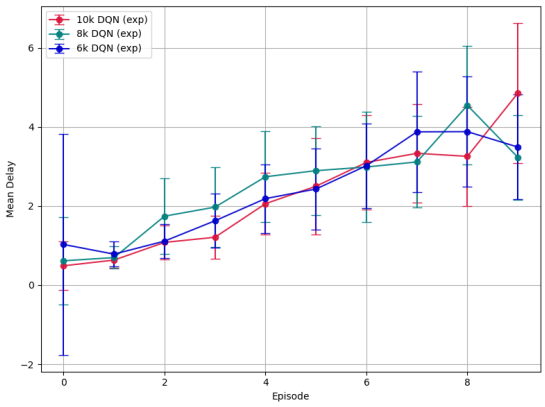


Figure 17: Delay Error Bar, Congestion Limit Sweep

6.5 Trade-Off Analysis with alpha-Weighted Reward

To introduce delay-awareness into the learning process, the DQN agent was trained with a varying parameter α, which scales the importance of delay. Values of $\alpha = 0.1$, 0.5, and 1.0 were tested. As shown in Figure 14, smaller α values lead to higher delays, whereas larger values maintain lower delays.

The trade-off is further evidenced in Figure 15, where congestion metrics are plotted for the same episodes. For instance, in the final episode with $\alpha = 0.1$, congestion is well-regulated under the limit following previous violations, but this is accompanied by a rise in average delay.

6.6 Evaluation with Exponential Reward Function

In the final set of experiments, the exponential reward formulation was used with the same three congestion limits to balance delay and congestion more smoothly. Figure 16 and Figure 17 show that this approach leads to more stable training behavior. Interestingly, Episode 10 displays fewer extreme spikes, indicating that the agent avoids abrupt changes in send rates.

This observation is further supported by the send rate heatmaps in Figure 18, where the PPO agent displays more uniform color distributions compared to the highly stochastic baseline. These findings suggest that the exponential reward fosters a more balanced and less volatile policy.

7 Conclusion

This work presented *CongestRL*, a reinforcement learning-based framework for congestion control in virtual networks. By simulating packet exchange using multi-threaded nodes and routing mechanisms, the framework enables the integration of learning-based agents to manage congestion and delay. Two reinforcement learning policies, DQN and PPO, were implemented and evaluated alongside a random baseline, offering insight into how different learning approaches interact with dynamic network conditions and reward structures. The environment design facilitated realistic feedback through tunable congestion limits and delay penalties, forming the

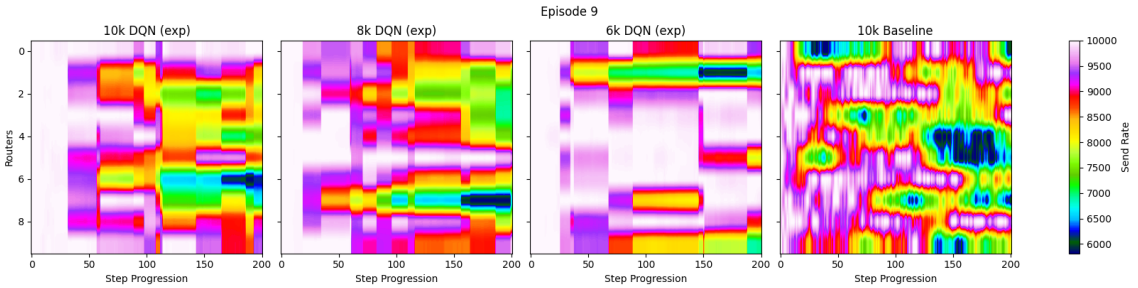


Figure 18: Send Rates Heatmap, Exponential Reward, 10th Episode

foundation for policy training and evaluation.

Experimental results demonstrated that both DQN and PPO agents were capable of learning effective behaviors, though with distinct tendencies. DQN quickly adapted to congestion constraints, gradually smoothing out violations and forming stable send-rate patterns, especially under varying congestion limits. PPO, by contrast, focused on minimizing congestion aggressively, but often at the cost of elevated or erratic delays. Through reward function variations—including linear and exponential formulations with adjustable delay weighting—it became evident that agent behavior is highly sensitive to the reward design, highlighting the trade-off between throughput and latency.

Although promising, the study faced scalability limitations, with simulations restricted to under 100 users due to computational constraints. Future directions include enabling dynamic topologies that respond to network state, rethinking static routing assumptions, and developing richer, more delay-aware reward functions. More controlled experiments using fixed graph structures may also provide better interpretability and comparative consistency. Overall, CongestRL offers a flexible testbed and solid baseline for advancing learning-based congestion control in complex, distributed networks.

Acknowledgments

Results presented in this work have been produced using the Aristotle University of Thessaloniki (AUTh) High Performance Computing Infrastructure and Resources. Additionally, the open-source language model ChatGPT [7] was utilized in parts of these experiments. It generated portions of the code which were then tested and analyzed, as well as enhanced the overall readability and clarity of this report.

References

[1] A. Hagberg and D. Conway, "Networkx: Network analysis with python," *URL: https://networkx. github. io*, pp. 1–48, 2020.

[2] P. S. Foundation, "threading — thread-based parallelism." https://docs.python.org/3/library/threading.html, 2023.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[4] A. Paszke, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[5] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, PMLR, 11–13 Apr 2011.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[7] OpenAI, "ChatGPT." https://chat.openai.com/.