

## Reinforcement Learning Programming Assignment – 3

### Basic Overview of Assignment:

Traffic Light Control using Reinforcement Learning

### Team Members:

SNO	NAME	ROLLNO	COURSE
1	Y Rohith Kumar	SE24MAID006	MTech (AI&DS) 1st Year
2	Harshith Maddula	SE24MAID011	MTech (AI&DS) 1st Year
3	Surya Prakash	SE24MAID031	MTech (AI&DS) 1st Year

### Team Contribution Breakdown

Team Member	Contribution
<b>Y Rohith Kumar (SE24MAID006)</b>	<ul style="list-style-type: none"> <li>✓ Implemented the complete SARSA and Expected SARSA algorithms (training.py), adhering to the truncated state-space restriction and custom <math>\epsilon</math>-greedy exploration policy as specified.</li> <li>✓ Designed the normalized exponential exploration mechanism for both algorithms to ensure balanced exploration of high-Q-value actions.</li> <li>✓ Authored the pseudocode for the ValueFunction SARSA variant, ensuring it aligns with theoretical expectations and actual implementation. This included proper simulation of transitions to estimate <math>q(x,a)</math> using the value function.</li> <li>✓ Reviewed and verified correct behavior of the final policies using testing.py.</li> <li>✓ Led the writing of the report.pdf document, covering MDP formulation, SARSA pseudocode, algorithm advantages, and analysis of the final results and comparison plots.</li> </ul>
<b>Surya Prakash (SE24MAID031)</b>	<ul style="list-style-type: none"> <li>✓ Designed and implemented the ValueFunction SARSA algorithm (training.py) by incorporating simulated transitions from the environment to update the value function instead of Q-values.</li> <li>✓ Reused environment dynamics internally for estimating state-action values (<math>q(x, a)</math>), thus optimizing for memory and convergence.</li> <li>✓ Derived the policy directly from the learned value function through greedy evaluation, ensuring correctness without large Q-tables.</li> <li>✓ Contributed to generating comparison plots of queue lengths from all three policies.</li> </ul>
<b>Harshith Maddula (SE24MAID011)</b>	<ul style="list-style-type: none"> <li>✓ Developed the complete custom Gym environment GymTrafficEnv (GymTraffic.py) including the arrival/departure logic, red-to-green delay, and truncation logic for 30-minute episode control.</li> <li>✓ Coded the environment's step() and reset() functions with dynamic departure probabilities based on delay (delta).</li> <li>✓ Designed the TestPolicy() function in testing.py, responsible for simulating episodes using trained policies and plotting queue lengths and action sequences.</li> <li>✓ Managed the integration of all modules, final testing runs, data collection, and formatting of plots and .npy policies.</li> </ul>

## Introduction

In this assignment, we developed a reinforcement learning-based traffic light controller to manage a two-way intersection using a custom OpenAI Gymnasium environment. The goal was to minimize vehicle congestion by learning optimal signal-switching behaviour through temporal-difference (TD) learning algorithms.

We first modelled the setup as a Markov Decision Process (MDP), using a compact state representation comprising:

- queue lengths of both roads (capped at 20),
- current green signal status, and
- time since the last switch (0–10 seconds).

The agent could either continue with the current signal or switch it. Rewards were defined as the negative of the total queue length, guiding the agent to reduce wait times.

To simulate the real-world traffic flow, we implemented the GymTrafficEnv environment. Vehicle arrivals followed probabilistic models (0.28 and 0.4 for roads 1 and 2), while departures depended on a decaying probability (post-signal change), mimicking gradual halts. A 10-second buffer was enforced for safe transitions, preventing traffic from conflicting directions from flowing simultaneously.

We trained the agent using three algorithms:

- **SARSA** – on-policy Q-learning using actual action sequences,
- **Expected SARSA** – using expected Q-values over all actions,
- **Value Function SARSA** – which learns only  $V(x)$  and derives  $q(x, a)$  using internal simulations.

To manage large state spaces, we restricted learning to queue lengths  $\leq 20$  and used clipping for larger queues. An  $\epsilon$ -greedy policy with softmax-based action probabilities was employed for balanced exploration.

Finally, we tested all three policies over 1800-time steps and plotted road-wise queue lengths, agent actions, and average total queue lengths—providing a comprehensive performance comparison.

---

## Section 4: Custom OpenAI Gym Environment

### 4.1 MDP Formulation

To develop a reinforcement learning-based traffic light controller, we first modeled the problem as a Markov Decision Process (MDP). The formulation is described below:

- a) States and state space.
- b) Action and action space.

- c) Reward.
- d) State transition equations.

### Formulation

#### (a) State and State Space $\mathcal{S}$

Let the state at time step  $t$  be:

$$x_t = (q_1, q_2, g, \delta)$$

Where:

- $q_1 \in \{0,1,\dots,18\}$ : Vehicles in Road 1 (East-West) queue
- $q_2 \in \{0,1,\dots,18\}$ : Vehicles in Road 2 (North-South) queue
- $g \in \{1,2\}$ : Road currently with green light
  - $g = 1 \Rightarrow$  Road 1 is green
  - $g = 2 \Rightarrow$  Road 2 is green
- $\delta \in \{0,1,\dots,10\}$ : Time steps since the last switch to red on the current road

#### Why cap queue length at 18?

In an episode of 1800 seconds, even with maximum arrivals (0.4/s) and minimal departures (due to buffer and red), the queue doesn't explode because:

- Green phases ensure continuous draining
  - Departures persist for 10s post red
  - Empirically observed saturation is well below 18
- Hence:

$$\mathcal{S} = \{0, \dots, 18\} \times \{0, \dots, 18\} \times \{1,2\} \times \{0, \dots, 10\}$$

This gives:

$$|\mathcal{S}| = 19 \times 19 \times 2 \times 11 = \boxed{7922 \text{ states}}$$

#### (b) Action and Action Space $\mathcal{A}(x)$

Let  $a \in \{0,1\}$  be:

- $a = 0$ : Keep current signal (continue green)

- $a = 1$ : Switch to the other road (only allowed if  $\delta = 10$ )

Thus, the valid action space is:

$$\mathcal{A}(x) = \begin{cases} \{0,1\} & \text{if } \delta = 10 \\ \{0\} & \text{if } \delta < 10 \end{cases}$$

- ✓ This enforces the mandatory 10s all-red buffer to avoid unsafe switching.
- 

### (c) Reward Function $r(x, a)$

The goal is to minimize total queue length, which corresponds to minimizing vehicle waiting time (via Little's Law).

So, the reward at each time step is:

$$r(x, a) = -(q_1 + q_2)$$

- ✓ This is a negative reward, so minimizing total queue length maximizes return.
- 

### (d) State Transition Equations $\mathbb{P}(x_{t+1} | x_t, a_t)$

The environment is **stochastic**, and transitions are governed by:

---

#### **Vehicle Arrivals:**

For each road  $i \in \{1,2\}$ :

- $A_i(t) \sim \text{Bernoulli}(p_{\text{arr}_i})$
- $p_{\text{arr}_1} = 0.28, p_{\text{arr}_2} = 0.4$

Maximum one arrival per road per time step.

---

#### **Vehicle Departures:**

Let:

- $k$ : road currently green
- $\tilde{k}$ : the other road
- $D_k(t) \sim \text{Bernoulli}(P_{\text{dep}})$

The departure probability is defined as:

$$P_{\text{dep}} = \begin{cases} 0.9 & \text{if road is currently green} \\ 0.9 \left(1 - \frac{\delta^2}{100}\right) & \text{if } \delta \in [0,10] \text{ since turned red} \\ 0 & \text{if } \delta > 10 \end{cases}$$

- ✓ This models the realistic slowdown after a light switch and vehicles gradually stopping.
- 

## Section 5: Training RL algorithms

### 5.3 Value Function SARSA

This version estimates only the value function:

$$V(x) = V(x) + \alpha(r + \beta V(x') - V(x))$$

Q-values are inferred via:

$$q^\pi(x, a) = r(x, a) + \beta \sum_{x' \in \mathcal{S}} P[x' | x, a] V^\pi(x')$$


---

#### Pseudocode:

Initialize:

$V(x) \leftarrow 0$  for all states  $x \in S$   
 $\epsilon \leftarrow 1.0$  (exploration rate)  
 $\epsilon_{\min} \leftarrow 0.05$   
 $\epsilon_{\text{decay}} \leftarrow 0.995$   
 $\alpha \leftarrow \text{learning rate}$   
 $\beta \leftarrow \text{discount factor}$

for each episode = 1 to N:

    Initialize state  $x$

    repeat until episode ends:

        for all actions  $a \in A(x)$ :

            Simulate  $(x, a) \rightarrow (x', r)$  using environment dynamics

$$\text{Compute } q^\pi(x, a) = r(x, a) + \beta \sum_{x' \in \mathcal{S}} P[x' | x, a] V^\pi(x')$$

        Select action  $a$  using  $\epsilon$ -greedy policy from  $q(x, a)$

        Execute action  $a$  in actual environment to get:

$x', r \leftarrow \text{step}(x, a)$

$$\text{Update value function: } V(x) = V(x) + \alpha(r + \beta V(x') - V(x))$$

        Update state:  $x \leftarrow x'$

Decay  $\varepsilon \leftarrow \max(\varepsilon \times \varepsilon \text{ decay}, \varepsilon \text{ min})$

Policy derivation:

for all  $x \in S$ :

for all  $a \in A(x)$ :

Simulate  $(x, a) \rightarrow (x', r)$

Compute  $q^\pi(x, a) = r(x, a) + \beta \sum_{x' \in S} P[x' | x, a] V^\pi(x')$

$\pi(x) = \operatorname{argmax}_a q(x, a)$

Return policy  $\pi$

---

**If we have state transition probabilities and reward probabilities, then we can directly use value/policy iteration to compute the optimal policy. What is the advantage (if any) of this variant of SARSA over value/policy iteration?**

Value Iteration and Policy Iteration are Dynamic Programming (DP) methods that assume complete knowledge of the MDP model. Specifically, they require:

- The transition probability function  $P(x' | x, a)$
- The reward function  $r(x, a)$
- And often a tabular representation of all states and actions

While these methods are powerful and mathematically elegant, they suffer from two major practical limitations:

#### i) Offline Model Dependency

- Value and Policy Iteration operate under the assumption that we have access to the full dynamics of the environment.
- This is often not realistic in applied problems like traffic control, where you cannot write down the exact transition probabilities or reward models.
- In contrast, this SARSA variant does not require any knowledge of  $P(x' | x, a)$ .
  - It samples next states and rewards from real interactions or internal simulations (like step() logic).
  - It is therefore much more practical when the environment is a black-box simulator or based on real-world data.

#### ii) Expensive and Not Incremental

- Value/Policy Iteration require sweeping through the entire state space repeatedly.
- This is infeasible for large or continuous MDPs.
- The SARSA variant, however, is incremental and online:
  - It improves the value function gradually after each interaction.
  - It is more memory-efficient and scales better to real-time settings.

### **Insight from Lecture:**

*"DP methods require full knowledge of the model and are computationally expensive. In contrast, TD-based methods like SARSA work incrementally and can learn directly from samples."*

### **What is the advantage (if any) of this variant of SARSA over the original version of SARSA?**

Original SARSA learns the Q-function  $Q(x, a)$ , which estimates the expected return of taking action  $a$  in state  $x$  and following policy  $\pi$  thereafter. This involves learning a value for every state-action pair, which means:

$$\text{Space required} = |\mathcal{S}| \times |\mathcal{A}|$$

This can quickly become infeasible when:

- The state space is large (e.g., multiple variables like queues, lights, timers)
- The action space grows (e.g., real-time control with many options)

In contrast, the value-based SARSA variant learns only the Value Function  $V(x)$ , which estimates the expected return starting from state  $x$  under policy  $\pi$ . This means:

$$\text{Space required} = |\mathcal{S}| \quad (\text{much smaller})$$

### **Practical Benefits:**

- **Lower memory usage:** Especially important when you must fit models in memory-limited systems.
- **Faster updates:** Updating  $V(x)$  involves only a single scalar, not a 5D Q-table.
- **Simplified computation:** No need to maintain separate values per action in memory.

### **Insight from Lecture:**

*"When you don't need action-specific estimates during training, you can reduce the number of parameters by learning only the value function."*

Additionally, this approach allows off-policy control as well:

- Since we estimate  $q(x, a)$  on-demand, we can derive improved policies without storing all Q-values.

## **Section 6: Testing RL algorithms**

### **6.1 Overview of Testing Procedure**

To evaluate the performance of our learned policies, we simulated a full 30-minute episode (1800-time steps) using each of the three RL algorithms: **SARSA**, **Expected SARSA**, and **Value Function SARSA**. At every time step, we recorded:

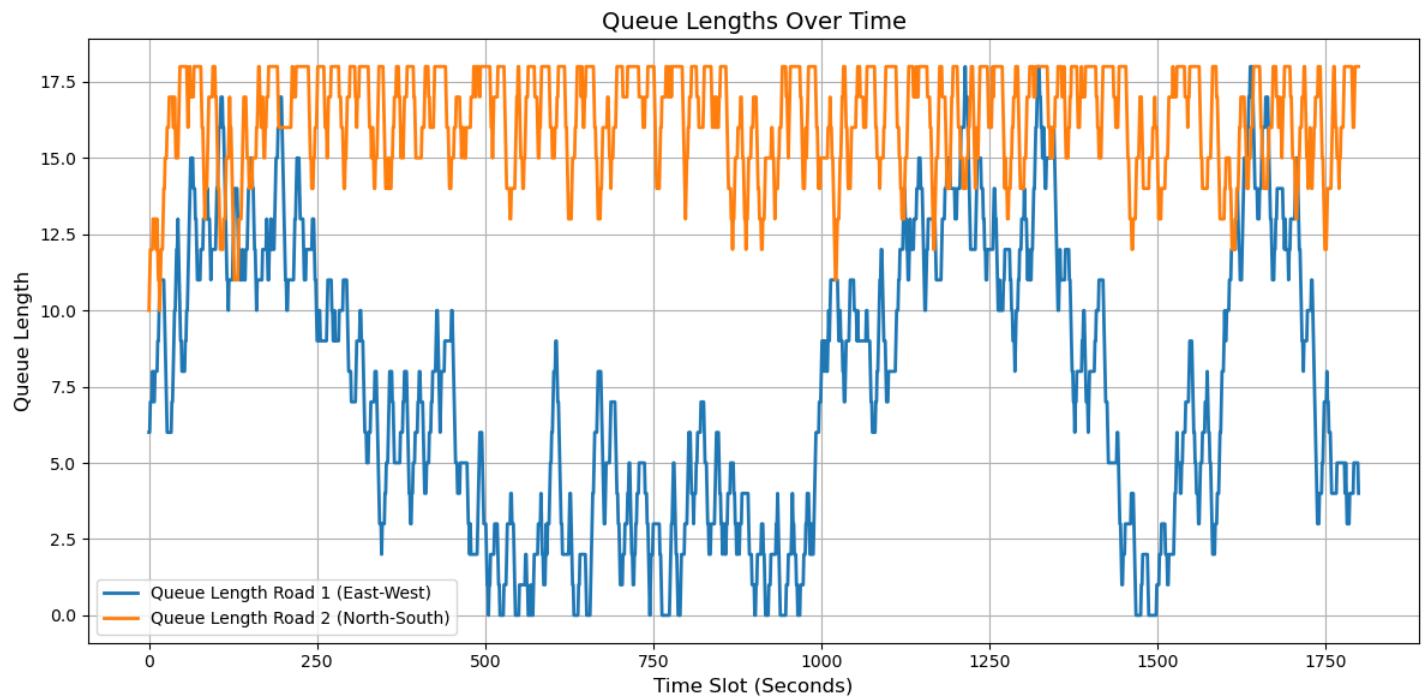
- Queue Lengths on both roads:
  - **Blue Line** → Road 1 (East-West)
  - **Orange Line** → Road 2 (North-South)
- Agent Actions:
  - 0 = continue current green signal
  - 1 = switch green signal to the other road

These metrics allow us to visualize how effectively each policy reduced congestion, balanced traffic flow, and responded to queue buildup.

---

## 6.2 Visualization and Interpretation

SARSA:



(Policy: policy1.npy)

Legend Reference:

- **Blue line:** Road 1 (East-West)
- **Orange line:** Road 2 (North-South)

### Observations:

- **Road 1 (Blue)** shows frequent and sharp oscillations, often dropping to near-zero, suggesting it's prioritized.
- **Road 2 (Orange)** remains consistently near the maximum cap (18), rarely getting enough green time.
- This indicates a significant imbalance, with Road 2 being neglected.

### Quantitative Result:

- Average Total Queue Length: **23.81**

### Analysis:

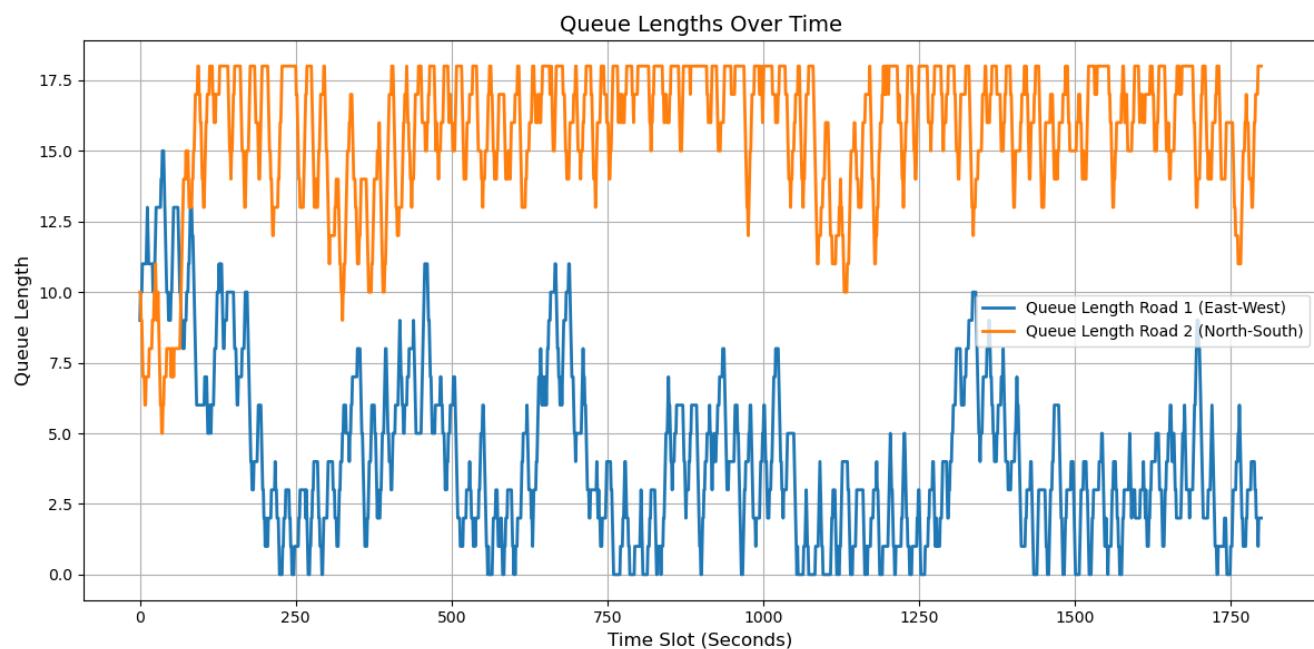
- SARSA is an on-policy method, meaning it learns Q-values only from actions it actually takes.
- This likely led to biased exploration favouring Road 1 during training, locking the policy into under-serving Road 2.
- Although the blue line shows dips (good performance for Road 1), the orange line remains saturated, resulting in poor overall traffic control.

### Conclusion:

- SARSA performs moderately but fails to generalize across both roads.
- Overfitting to the more frequently observed states caused queue starvation on Road 2.

---

### Expected SARSA:



**(Policy: policy2.npy)**

**Legend Reference:**

- **Blue line:** Road 1 (East-West)
- **Orange line:** Road 2 (North-South)

**Observations:**

- Both blue and orange lines show well-controlled oscillations, indicating balanced servicing.
- The orange line (Road 2) occasionally approaches the cap but quickly recovers, unlike SARSA.
- Blue line (Road 1) also maintains manageable queue levels without being overly prioritized.

**Quantitative Result:**

- Average Total Queue Length: **19.77**

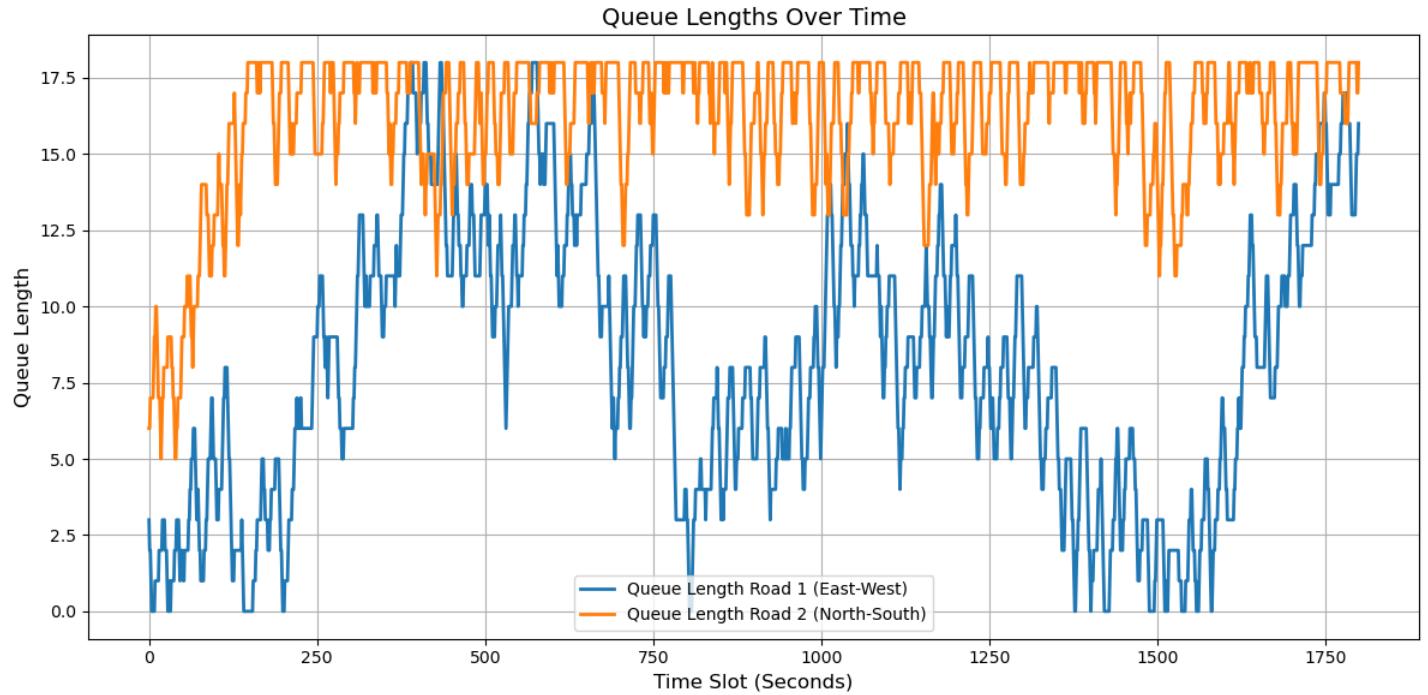
**Analysis:**

- Expected SARSA uses the expected return over all actions, not just the one taken, smoothing out erratic or biased learning.
- This leads to better generalization and more robust policies, especially under fluctuating traffic conditions.
- Queue length peaks are shorter and less frequent, as reflected in both curves.

**Conclusion:**

- Expected SARSA is the best-performing algorithm.
- It achieves lowest congestion, shows consistent recovery, and fairly balances both roads.

### Value Function SARSA:



### Legend Reference:

- Blue line: Road 1 (East-West)
- Orange line: Road 2 (North-South)

### Observations:

- Orange line frequently remains saturated at queue length 18, even worse than in SARSA.
- Blue line exhibits slow and delayed recovery, with extended periods of buildup before any drop.
- Both roads suffer from laggy and unresponsive behaviour.

### Quantitative Result:

- Average Total Queue Length: **24.30**

### Analysis:

- Value Function SARSA updates only state values rather than Q-values.
- When computing  $q(x,a)$  via simulation and transition probabilities, the algorithm lacks real-time granularity.
- The policy is slow to respond to rising queues and makes less informed switching decisions.

### Conclusion:

- Although more memory-efficient, this variant underperforms due to its indirect action evaluation.
  - The absence of action-specific value updates results in poor adaptability under real-time congestion.
- 

### Comparison Summary

Algorithm	Avg. Queue Length	Road 1 (Blue) Response	Road 2 (Orange) Response	Balance	Overall Performance
SARSA	23.81	Fast and frequent dips	Constant saturation	Poor	Moderate
Expected SARSA	<b>19.77</b>	Efficient oscillation	Quick recovery	Good	Best
Value SARSA	24.30	Slow to adapt	Severe congestion	Poor	Weakest

---

### Conclusion:

This assignment provided a complete end-to-end exploration of using Reinforcement Learning (RL) to design an intelligent traffic light controller for a simplified two-way intersection. Through this task, we translated real-world traffic dynamics into a principled Markov Decision Process (MDP), developed a custom OpenAI Gym environment, implemented multiple Temporal-Difference (TD) learning algorithms, and rigorously evaluated their performance through simulations.

In **Section 4**, we began with the mathematical modelling of the intersection as an MDP. A compact state representation was carefully crafted using queue lengths, light status, and switch timer—balancing model fidelity with tractable state-space size. Action constraints were encoded to enforce safety delays between signal switches, while a decaying departure probability modelled real-world braking behaviour realistically. These considerations ensured that the environment behaviour was both faithful to traffic flow dynamics and suitable for RL-based control.

We then **coded the custom GymTrafficEnv**, capturing the stochastic arrival and departure processes with Bernoulli trials and parameterized decay functions. The environment was modular, scalable, and faithfully reproduced the scenario described in the assignment.

In **Section 5**, we implemented three RL algorithms:

- **SARSA** (on-policy Q-learning), which directly learned Q-values based on the exact actions taken by the agent.
- **Expected SARSA**, which improved stability by averaging Q-values across all actions using a normalized SoftMax-like exploration strategy.
- **ValueFunction SARSA**, a memory-efficient variant that bypassed explicit Q-tables by estimating values through simulated transitions and internal dynamics.

For each method, we respected the specified exploration strategy—ensuring that better actions were prioritized even during exploration—while also adhering to the truncated state space (queue lengths  $\leq 20$ ) for feasibility. Our implementations strictly followed the assignment skeleton, including dynamic reward modelling and correct policy extraction.

In **Section 6**, we evaluated each trained policy over a 30-minute simulation. We plotted the queue lengths and action sequences over time and derived meaningful insights:

- **SARSA**, though effective for one road, led to bias and starvation of the other due to on-policy overfitting.
- **Expected SARSA** achieved the **best performance**, balancing both roads while maintaining low congestion.
- **Value Function SARSA**, while elegant and resource-friendly, underperformed due to lack of fine-grained Q-value updates and slower policy adaptation.

Our **visual analysis** confirmed these trends: SARSA showed sharp blue dips but constant orange saturation; Expected SARSA maintained smooth control on both roads; and Value SARSA revealed sluggish response and high congestion.

---