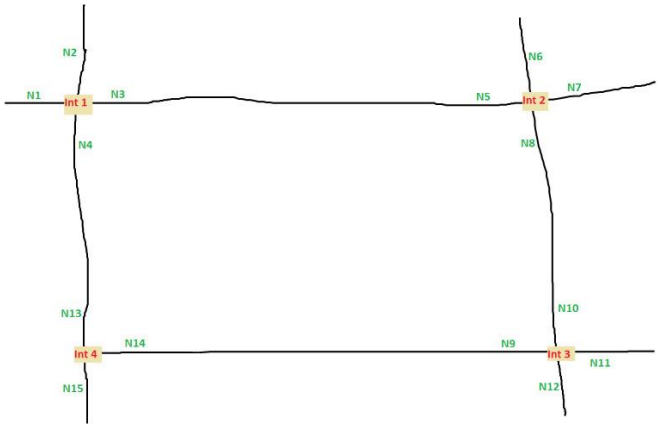
Adaptive traffic control system using Deep Reinforcement Learning

R. Anirudh
Akshay Kekuda
Mithun Krishnan

Problem Statement:

Traffic management is an area of great interest where lots of research takes place. We aim to apply Deep Reinforcement Learning to solve the traffic management problem. Our aim is to find an optimal policy to route traffic in order to reduce congestion and waiting times. For our project we have taken a real-world 4-junction road network from Texas, USA using OpenStreetmap.

We use Simulation of Urban Mobility (SUMO) software to simulate traffic scenarios. Using SUMO, traffic patterns are generated and are used to test the Deep Reinforcement Learning algorithm.



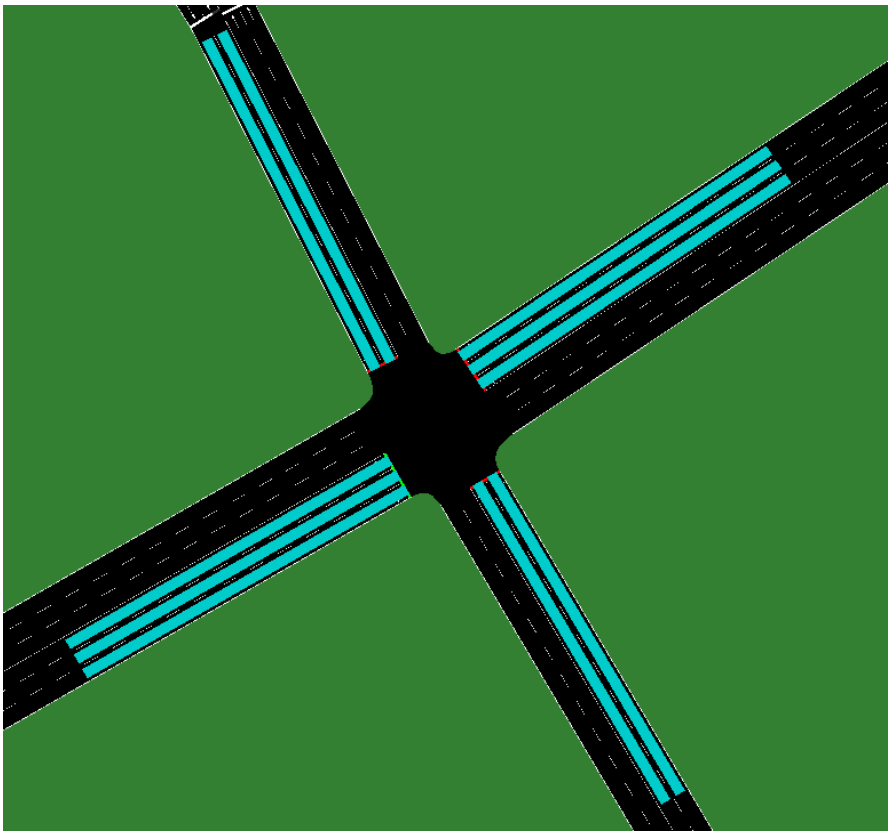
The above figure shows the traffic signals that are operating in our setting. There are a total of 15 signals across the 4 intersections. In our work, intersections are operated cyclically (INT1 ->

INT2 -> INT3 -> INT4 -> INT1) at a gap of 16secs. At each intersection, the decision on which signal to operate is made by the RL agent. The goal of the RL agent is to minimize Queue Lengths, Waiting Time, Time Loss and to disperse traffic through the network in the least time. We have compared our algorithm with the following existing algorithms and have obtained better results:

1. **Static Signaling:** Typical system present in cities. Signals are operated in a fixed manner (every 16s, signals change) in a round robin fashion for each intersection. In our case, order was (n1-n5-n9-n13)-(n2-n6-n10-n14)-(n3-n7-n11-n15)-(n4-n8-n12-n13) etc
2. **Longest Queue First:** At every intersection, a decision is made based on the side which has the maximum queue length and green is given for that side. Again, as per our setting, we operate on intersections cyclically.
3. **n-step differential SARSA:** Check https://github.com/Learning-something-deep/adaptive-traffic-control-nstep_sarsa for implementation details. We use these results to compare with our DRL based algo.

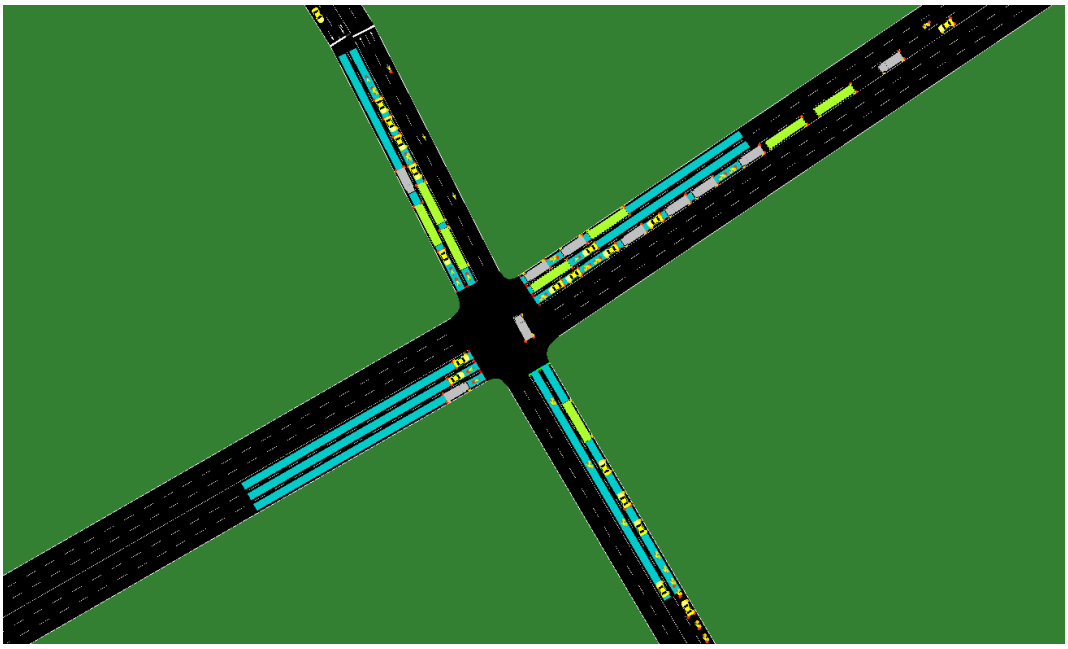
SUMO environment details:

1. **Use of induction loops:** To detect the amount of traffic waiting at the signals, we use Lane area detectors (E2 detectors) covering upto 70mts from the intersections. These detectors output the percentage of its area on which vehicles are standing.



2. **Description of lanes:** Our network consists of roads of one to three lanes. All the individual lanes are 10ft wide. The main arterial roads are 1.5km to 2km in length.

3. **Types of vehicles:** The traffic consists of 4 types of vehicles. The traffic distribution is modelled close to urban traffic. Bikes (41%), Cars (37.5%), Trucks (12.5%), Buses (8.3%). These vehicles have different speeds, acceleration and braking time parameters.



4. **Traffic density and pattern:** The number of vehicles entering the network follows a Binomial distribution with n=5 & p=0.15, which closely resembles a real world scenario. The vehicles are also distributed according to the number of lanes in a road, with higher traffic in multi-lane roads.

Algorithm:

(i) Quantile Regression

Instead of learning the expected return Q, ***distributional RL*** focuses on learning the full distribution of the random variable Z directly. There are various approaches to represent a distribution in an RL setting. In QR-DQN the distribution of Z is represented by a uniform mix of N supporting quantiles:

$$Z_\theta(s,a) \doteq \frac{1}{N}\sum_{i=1}^{N}\delta_{\theta_i(s,a)}$$

where δx denotes a Dirac at x ∈ R, and each θi is an estimation of the quantile corresponding to the quantile level

$\hat{\tau}_i \doteq \frac{\tau_{i-1}+\tau_i}{2}$ with $\tau_i \doteq \frac{i}{N}$ for N for 0 ≤ i ≤ N.

The state-action value Q(s, a) is then approximated by

$$\frac{1}{N}\sum_{i=1}^{N}\theta_i(s,a)$$

Such approximation of a distribution is referred to as quantile approximation. Similar to the Bellman optimality operator in mean-centered RL, we have the distributional Bellman optimality operator for control in distributional RL

$$\mathcal{T}Z(s,a) \doteq R(s,a) + \gamma Z(s', \arg\max_{a'} \mathbb{E}_{p,R}[Z(s',a')])$$
$$s' \sim p(\cdot|s,a)$$

Based on the distributional Bellman optimality operator, proposed to train quantile estimations (i.e., $\{q_i\}$) via the ***Huber quantile regression loss***. To be more specific, at time step t the loss is

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{i'=1}^{N}\left[\rho^{\kappa}_{\hat{\tau}_i}\left(y_{t,i'}-\theta_i(s_t,a_t)\right)\right]$$

where $y_{t,i'} \doteq r_t + \gamma\theta_{i'}\left(s_{t+1}, \arg\max_{a'}\sum_{i=1}^{N}\theta_i(s_{t+1},a')\right)$ and $\rho^{\kappa}_{\hat{\tau}_i}(x) \doteq |\hat{\tau}_i - \mathbb{I}\{x<0\}|\mathcal{L}_\kappa(x)$, where $\mathbb{I}$ is the indicator function and $\mathcal{L}_\kappa$ is the Huber loss,

$$\mathcal{L}_\kappa(x) \doteq \begin{cases} \frac{1}{2}x^2 & \text{if } x \le \kappa \\ \kappa(|x| - \frac{1}{2}\kappa) & \text{otherwise} \end{cases}$$

(ii) DQN

The basic idea behind Q-Learning is to use the Bellman optimality equation as an iterative update

$$Q_{i+1}(s,a) \leftarrow \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')\right]$$

and it can be shown that this converges to the optimal Q-function, i.e. $Q_i \rightarrow Q_*$ as $i \rightarrow \infty$

For most problems, it is impractical to represent the Q-function as a table containing values for each combination of s and a. Instead, we train a function approximator, such as a neural network with parameters θ, to estimate the Q-values, i.e. $Q(s,a;\theta) \approx Q_*(s,a)$. This can done by minimizing the following loss at each step i:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'\sim\rho(.)}\left[(y_i - Q(s,a;\theta_i))^2\right] \text{ where } y_i = r + \gamma \max_{a'} Q(s',a';\theta_{i-1})$$

Here, y_i is called the TD (temporal difference) target, and $y_i - Q$ is called the TD error. ρ represents the behavior distribution, the distribution over transitions s, a, r, s′ collected from the environment.

Note that the parameters from the previous iteration θ_{i-1} are fixed and not updated. In practice we use a snapshot of the network parameters from a few iterations ago instead of the last iteration. This copy is called the ***target network***.

Q-Learning is an ***off-policy*** algorithm that learns about the greedy policy $a = \max_a Q(s,a;\theta)$ while using a different behavior policy for acting in the environment/collecting data. This behavior policy is usually an ϵ-greedy policy that selects the greedy action with probability $1-\epsilon$ and a random action with probability ϵ to ensure good coverage of the state-action space.

(iii) Why did we choose QR-DQN for our traffic problem?

If the environment is stochastic in nature (eg. occurrence of traffic jams) and the future rewards follow a multimodal distribution, choosing actions based on expected value may lead to suboptimal outcome.

Another benefit of modelling distribution instead of expected value is that sometimes even though the expected future rewards of 2 actions are identical, their variances might be very different. If we are risk averse, it would be preferable to choose the action with smaller variance. Knowing the distribution, rather than just the average, can improve the policy. This helps us avoid risky traffic deadlock situations.

Details of the Algorithm used:

States: State of the environment corresponds to the congestion levels in the 15 sides. We have experimented with the following state vectors:

1. Queue lengths corresponding to 15 sides (length of state vector = 15)
2. Percentage occupancy corresponding to 15 sides (length of state vector = 15) **[Used finally, as it was found to best describe the system]**

Actions: We have a total of 15 actions, which correspond to the signal that is to be made Green.

Reward: We have experimented with two different reward structures:

1. Queue lengths based reward. Reward at time t is defined as:

   $$R_t = -\sum_{j=1}^{M} q_{t,j}.$$

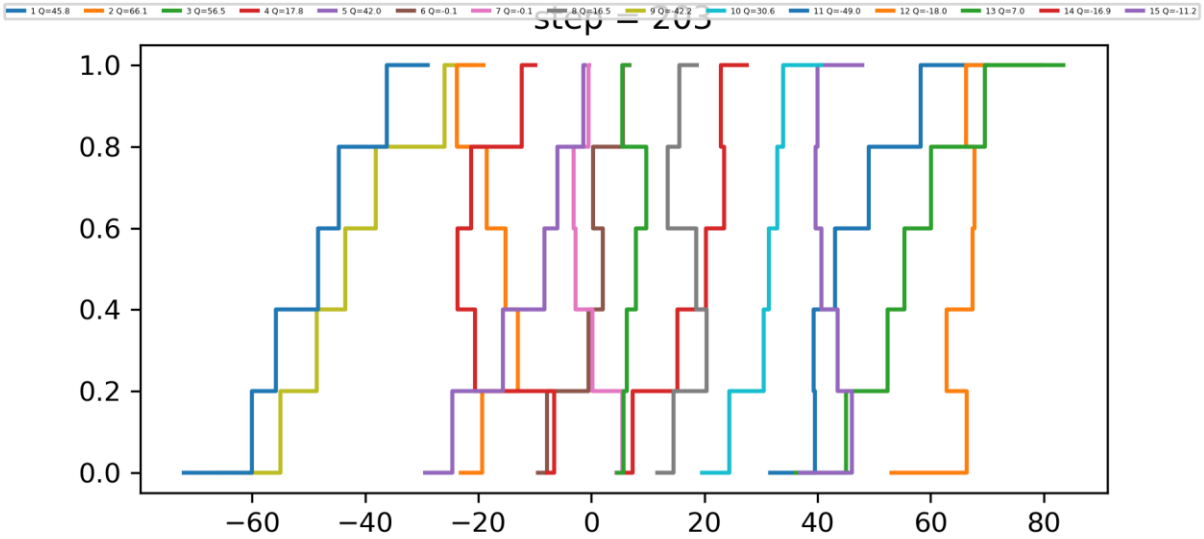
   Where $q_{t,j}$ is the queue length of side j at time t
2. Reward based on the change in percentage occupancy of a side after an action is taken for that side. If traffic from side N_i goes to side N_j, and if both sides are crowded, then N_i shouldn't be opened. We penalize such actions. **[Used finally, as it gives consistent results]**

This kind of reward structure makes our traffic system a centralised one, thus making it better than LQF and static signalling systems.

QR-DQN parameters:
These parameters were finalized after experimentation with several values. These values provided consistent good results.
n = 5 (no. of quantiles)
Target NN update frequency: 500 steps
Replay memory size: 500 experiences
Epsilon decay for e-greedy action selection: From 0.9 to 0.1 at 1/n rate
Gamma: 0.6 (discount factor)
Training batch size: 32
Training: 25 runs, where each run is around 16,000 simulation time steps. Each simulation time step equals 1sec. So each run is around 4.44hrs. And total training is equivalent to 4.62 days.

We train the Deep RL agent under different traffic patterns each run. Based on the trained NN, we run a Live algorithm to evaluate the performance, that chooses greedy actions based on the trained weight for a simulation.

Traffic Patterns: We have tested the Deep RL agent with randomly generated traffic patterns and observed that its performing better than SARSA, LQF, and Static signalling systems: (SUMO Traffic parameters: p = 0.75, e=12000, binomial=5, -L)

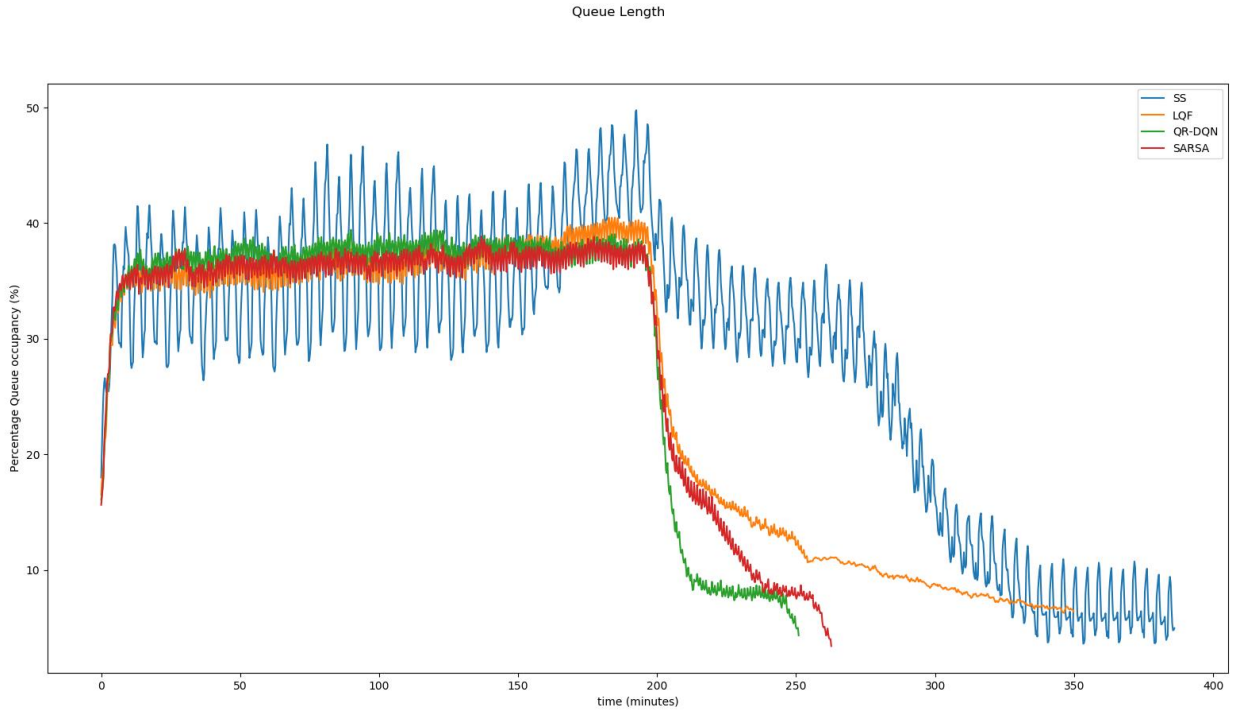


The above figure shows the estimated action value distribution Z(s, a) for all actions in a particular state. The x-axis is the expected return and y-axis are the probabilities.

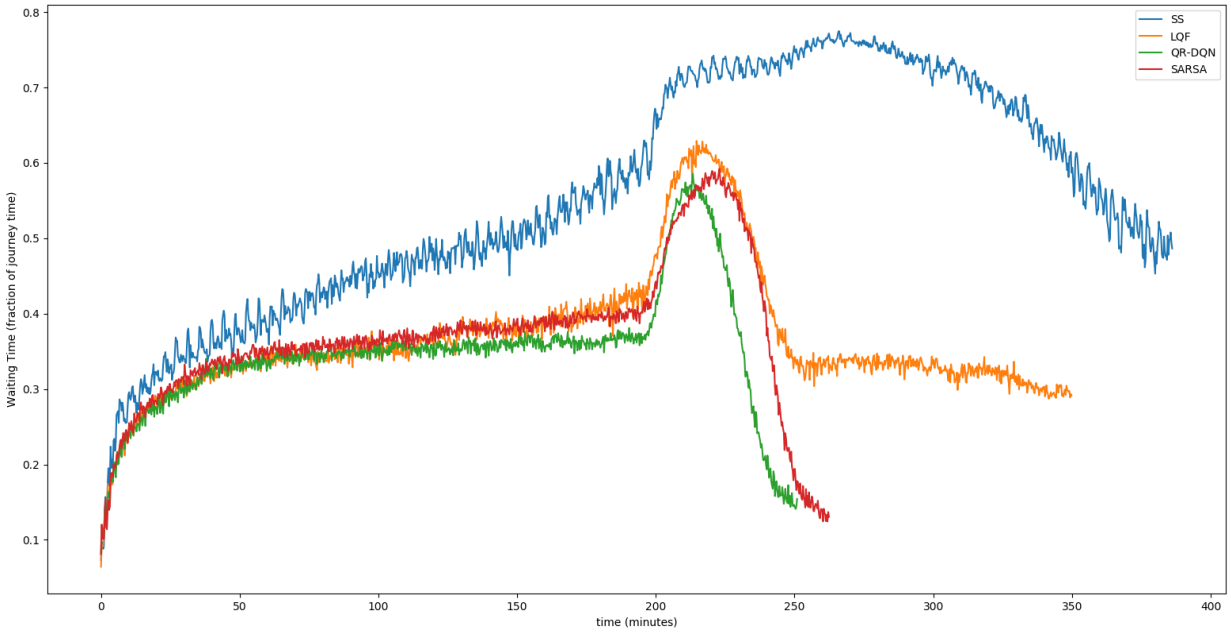
Results:

The algorithms are compared using the following metrics:

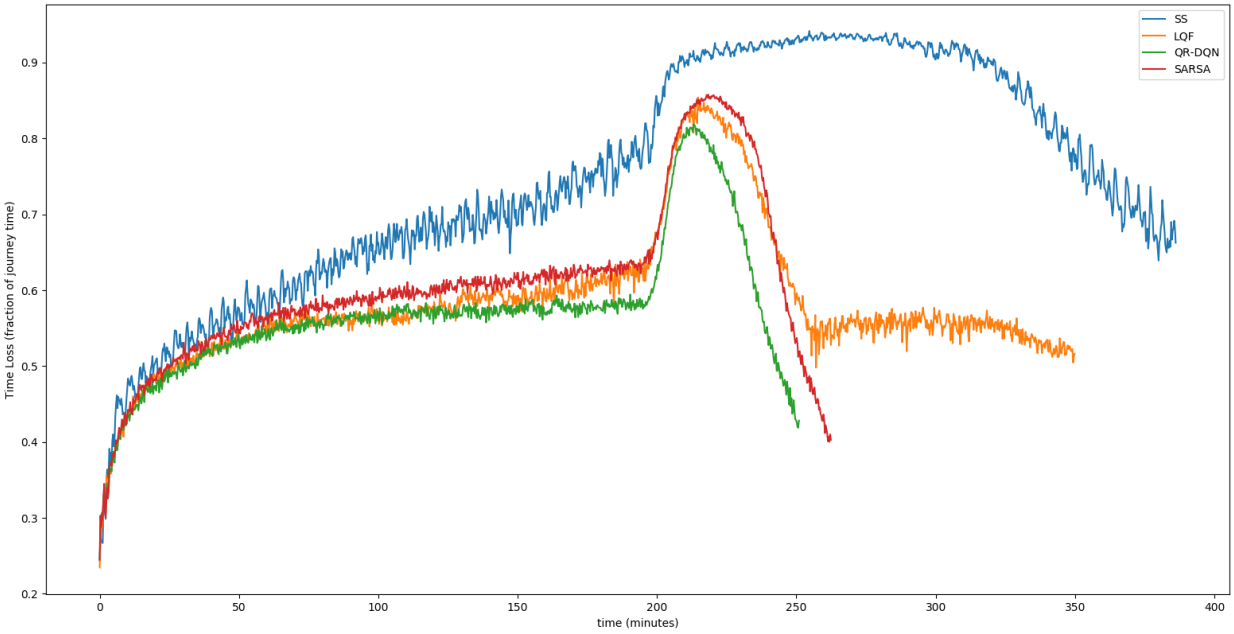
1) **Percentage queue occupancy:** Each signal has a lane area detector, which outputs the percentage of its area occupied by standing vehicles. This metric is the average occupancy of the 15 signals.
2) **Waiting Time:** This is the fraction of the journey time that a vehicle spends waiting at signals.
3) **Time Loss:** The time lost by a vehicle due to it driving below the ideal speed (due to slowdowns at signals etc.). This is taken as a fraction of the journey time.
4) **Dispersion time:** Time needed to disperse the traffic present in the road network. To measure this metric, traffic is allowed to flow for about 3.3hrs, after which, the time needed to empty the network is observed.

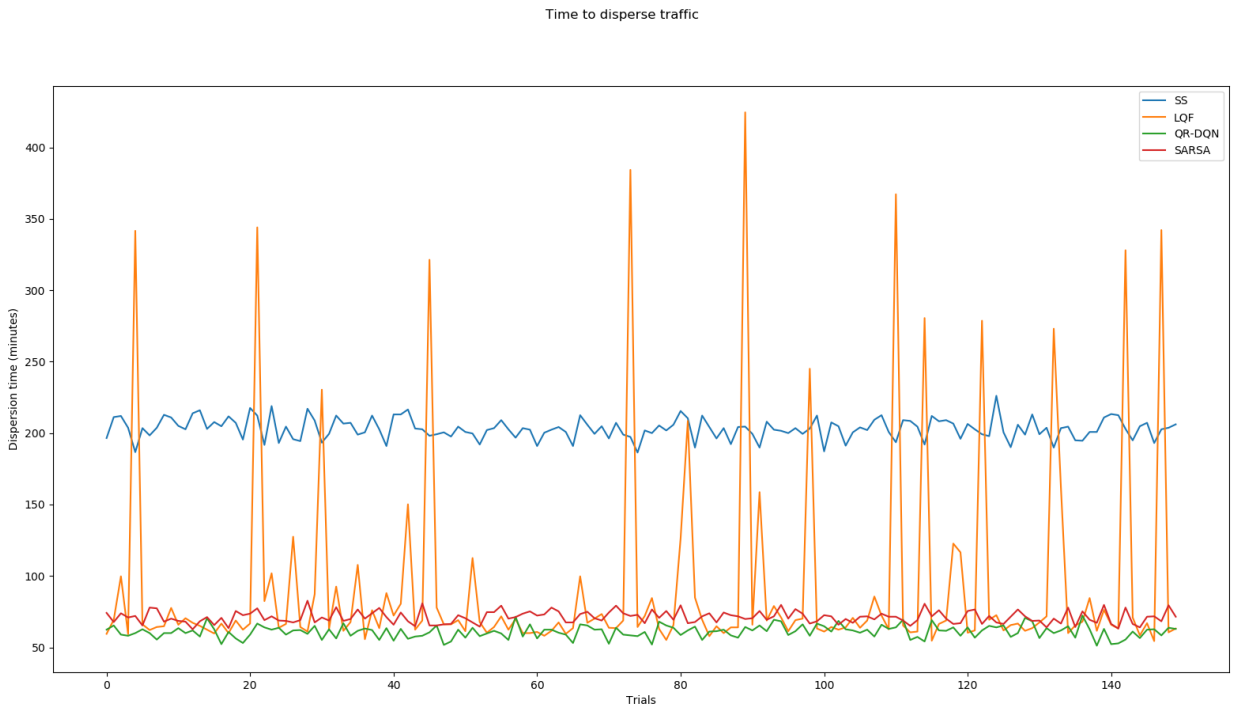


Waiting Time



Time Loss





Conclusion:

From the above graphs it can be seen clearly that QR-DQN outperforms Static Signalling [SS], Longest Queue Factor [LQF], n-step SARSA [SARSA] in all the metrics.

Static signalling, the typical traffic signalling followed in cities shows very poor performance. It has a high variance since it is not adaptive.

With respect to **Queue length**, the network takes about 150mins to reach steady state traffic flow. Once steady state is achieved, QR-DQN starts outperforming both SARSA and LQF. It shows a **7.5%** improvement as compared to LQF in steady state. During the dispersion phase (200mins onwards), QR-DQN is able to disperse the traffic **28.5%** faster than SARSA.

With respect to **Waiting Time**, the network takes about 100mins to reach steady state traffic flow. Once steady state is achieved, QR-DQN starts outperforming both SARSA and LQF. It shows a **15.7%** improvement as compared to LQF & SARSA in the steady state and dispersion phases. This shows that the vehicles reach their destinations faster in case of QR-DQN.

With respect to **Time Loss**, the network takes about 100mins to reach steady state traffic flow. Once steady state is achieved, QR-DQN starts outperforming both SARSA and LQF. It shows a **12.9%** improvement as compared to LQF & SARSA in the steady state and dispersion phases.

Finally, with respect to **Dispersion time**, we observe that LQF has multiple spikes in its dispersion times. These are called deadlock situations, wherein, the traffic comes to a complete halt due to congestion. As can be seen from the plots, both QR-DQN and SARSA never enter (**0%**) a deadlock state due to their ability to predict and avoid such situations. QR-DQN, in particular, achieves the lowest dispersion times on all trials.

References:

1) Distributional Reinforcement Learning with Quantile Regression, 2017, arXiv:1710.10044v1
2) Distributional Bellman and the C51 Algorithm: https://flyyufelix.github.io/2017/10/24/distributional-bellman.html
3) https://github.com/senya-ashukha/quantile-regression-dqn-pytorch
4) Reinforcement Learning With Function Approximation for Traffic Signal Control: http://www.cse.iitm.ac.in/~prashla/papers/2011RLforTrafficSignalControl_ITS.pdf

----------x----------