Department of Electrical, Computer, and Software Engineering

Part IV Research Project

Final Report

Project 32:

Applications of machine learning in transportation technology

Elliot Warner

Josh Allan

Avinash Malik

30/07/2020

Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

Name: Elliot Warner

Table of Contents

ABSTRACT: This report discusses an investigation in applying machine learning techniques to traffic control systems. Inefficiencies in the scheduling of traffic signal phases results in economic, social, and environmental consequences, so the use of modern algorithms, including learning algorithms, to improve this were explored and compared with the current system. SUMO, a traffic simulation software, was used for the simulation environment using data provided by Auckland Transport. An approximation of the status quo, alongside non-learning and reinforcement learning algorithms for the traffic control system were tested to determine the impact on travel time for vehicles in the network. It was found that the Earliest Deadline First algorithm was the best performing algorithm for time stopped and time loss for vehicles, across repeated trials. The reinforcement learning algorithm was capable of achieving a 5% decrease in time stopped, and a 4% decrease in time loss in the environment that it trained on, but was unable to generalize this performance across multiple trials. Further work to extend the project, and considerations for applying the results in practice, were discussed.

1. Introduction

Auckland is a large, continually growing city, and as a result those who commute suffer the financial, social and environmental consequences of excessive traffic every day.

The New Zealand Transport Agency (NZTA) has currently implemented the Sydney Coordinated Traffic Control System (SCATS) in Auckland, which was developed in the 1970s. This system lacks the flexibility to self-adapt to the rapidly changing transportation landscape, and we aim to take a data-driven approach to improve on SCATS by using machine learning to develop a more effective traffic control system (TCS).

Machine learning (ML) has shown to achieve state-of-the-art performance in many fields, including computer vision, linguistics, and classical games by leveraging compute power to build statistical models of large datasets.

One area of ML, reinforcement learning (RL) has shown significant, but narrow-focused success; DeepMind, an artificial intelligence company recently used deep RL (DRL) to develop MuZero [1], an algorithm capable of achieving superhuman performance in Go, Chess and Shogi, and state-of-the-art performance on Atari video games, without being given any knowledge of the underlying dynamics of these games. Interest has arisen in

applying DRL in other domains, however it is still in its infancy. One area in which some progress has been made is in the domain of transportation, most notably in the control of traffic control systems - but there is still room for improvement. Past research has often used more relaxed constraints, such as greater information of the traffic state than current data collection infrastructure can provide; moreover, direct comparisons have not been made between SCATS and RL algorithms.

Research will be undertaken to determine modern algorithm designs that are suitable for this application, from which we will develop one or more machine learning architectures. The simulation environment will allow us to analyse the performance of these algorithms in comparison to the current system, measured using the simulated mean vehicle travel time; the objective is to achieve an overall reduction in travel time with respect to the current system.

2. Traffic Control Systems

Many intersections in New Zealand use traffic lights that group lanes into phases to guide traffic to safely and efficiently traverse the intersection. The mechanics underlying the timings of these phases is the Traffic Control System. Research has been undertaken to analyse the status quo as well improve the efficiency of the TCS, aided in part by simulation.

2.1. Traffic Simulation Software

Traffic Simulation Software uses mathematical modelling to model the behaviour of vehicles in a network, allowing for the design and analysis of systems that might be too complicated for an exact solution to be found. Two notable traffic simulation software packages are SUMO and AIMSUN. These use car-following models, where each vehicle has its own dynamics individually programmed in the simulation [2].

2.2. SCATS

The Sydney Coordinated Adaptive Traffic System (SCATS) is an adaptive TCS that uses feedback from inductive loop detectors to manage the timings of traffic phases. The system uses multiple features to improve its efficiency: Cycle length - the time period required to serve each phase at one intersection - can be adjusted, as can the duration of each phase. Multiple adjacent intersections can be co-ordinated to improve traffic flow

by using phase timing offsets to synchronize green lights with vehicles that travel between the intersections. It is also capable of prioritizing lanes occupied by public transport. It does not use modelling of traffic demand in the management of phase timings. [3]

2.3. Proposed alternatives in Auckland

One paper examined the fairness and efficiency of intelligent transportation systems in Auckland [3], and investigated alternative solutions. The author took the approach of considering vehicles as computer instructions, where the TCS had to process each 'instruction' in a network of intersections. By framing the problem as such, it can be treated by using established computer scheduling algorithms. One algorithm was Earliest Deadline First (EDF), where priority was given to lanes that had vehicles waiting at an intersection for the longest duration. The other two algorithms were based on the queue length, prioritizing either the most occupied lanes, or the Accumulated Queue Length Weight algorithm, which used a function of the occupancy and the time spent waiting for each vehicle occupying lanes at every intersection. The algorithms also used other features such as green-waves, synchronizing phases at consecutive intersections for groups of vehicles to pass through with little delay, and pre-emption, allowing these green-waves to override the previously anticipated phase. Alongside SCATS, the algorithms were simulated on two real-world hours of traffic on a network of four intersections, and outperformed the current system when assessed on average delay. With the exception of EDF, these algorithms worked on the assumption that the queue length for each lane was known.

3. Reinforcement Learning

"Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence."

- Sutton & Barto, Reinforcement Learning: An Introduction (2nd edition)

Reinforcement Learning (RL) is a sub-field of Machine Learning (ML) concerned with the mapping of states to actions within an environment. Algorithms are designed to solve this problem by optimization of a return, and have improved to achieve greater performance in increasingly complex environments. To understand the

evolution of modern RL algorithms, it is necessary to understand the concepts that provide the foundation to reach these solutions.

3.1. Fundamentals of Reinforcement Learning

3.1.1. Markov Decision Processes

The reinforcement learning problem can be considered as a sequential, dynamic programming problem, formally defined in the form of a finite Markov Decision Process (MDP). In a single episode at any given time t, an environment is represented by a state, $S_t \in \mathcal{S}$ and a reward $R_t \in \mathscr{R}$, where $\mathcal{S}$ and $\mathscr{R}$ are finite sets. The agent selects an action given the state, $A_t \in \mathcal{A}(s)$ which acts on the environment, resulting in a new reward and state, R_{t+1} and S_{t+1} respectively [4].

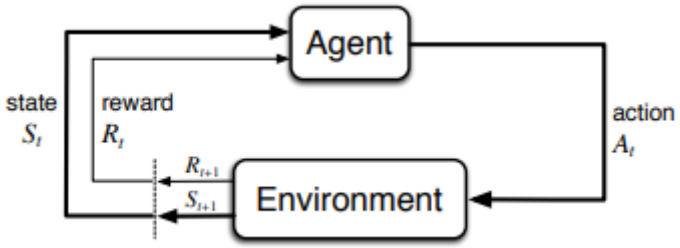

Fig. 1: A Visualization of a finite Markov Decision Process [1].

An important property of states that must be satisfied to be an MDP is the Markov property; that is, the conditional probability distribution of future states must only depend upon the current state S_t [5]. This can be formally defined as follows:

$$\mathbb{P}\,[S_{t+1} \mid S_t] = \mathbb{P}\,[S_{t+1} \mid S_1, \ldots, S_t] \tag{1}$$

The return is defined as the total sum of future discounted rewards:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2}$$

where γ is the discount rate, $0 \leq \gamma \leq 1$ [3]. In each state, an agent will select an action with accordance to a policy, π, which is the probability distribution of actions over a state:

$$\pi(a \mid s) = P\,[A_t = a \mid S_t = s] \tag{3}$$

The value of a state is given by the state-value function, where the value of the current state is determined by the expected return following a policy:

$$v_\pi(s) = E_\pi [G_t \mid S_t = s] \tag{4}$$

Similarly, the value of an action is given by the action-value function, or Q-values, which is the expected return from taking an action, and then following a policy:

$$q_\pi(s, a) = E_\pi [G_t \mid S_t = s, A_t = a] \tag{5}$$

3.1.2. Bellman Equations

As the reinforcement learning problem is a dynamic optimization problem, solving it becomes tractable with Bellman's principle of optimality [5] which states that "an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." By dividing the decision problem into smaller sub-problems, the solution can be found by iterative methods. The Bellman Equation for state values can be derived as follows, where the subsequent state after S is denoted S', and is a sub-element of $\mathcal{S}$ [4]:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big]. \end{aligned} \tag{6}$$

When an action has been taken, the next state can be one of several. To determine the optimal state-action value, the weighted sum of all possible following optimal state-action values is computed [4].

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\ &= \sum_{s', r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big]. \end{aligned} \tag{7}$$

The Bellman Optimality equations are essentially a guarantee that taking the optimal action at each state will result in the optimal policy, and therefore only the actions for the current state need to be considered, rather than the entire state-action tree.

3.2. Control Algorithms

3.2.1. Q-Learning

Q-learning is an off-policy, model-free algorithm that iteratively updates the state-action values in an MDP; it learns from the expected optimal policy following an action, rather than using the same policy that it acts on. As it learns, the state space is explored by infrequently taking random actions according to the ε-greedy policy. The pseudocode is shown below:

Algorithm One Q-learning:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

3.2.2. Deep Q-Network

The Deep Q-Network is an enhancement to the Q-Learning algorithm, originally designed for Atari games in the Arcade Learning Environment and at the time achieved state-of-the-art performance [6].

Estimating the action-value function for a sufficiently large state-space is impractical, therefore a function approximator is used to generalise the action-values with respect to a vector of parameters. A Q-network is defined as an action-value function approximated by a neural network function approximator with weights θ.

There are two notable features of this algorithm that improve its ability to converge; these are Experience Replay and Fixed Q-Targets.

Due to the correlated relationship between sequential samples, on-policy learning can bootstrap towards the maximizing action, thus biasing the learning experience to a specific area of the state space. An Experience Replay caches previous state transitions, and is sampled at random to make the learning data more independently distributed by breaking the temporal correlations. It also has the benefit of improving the efficiency of the data captured, by reusing previous state transitions in the learning of the function approximator.

In learning scenarios other than reinforcement learning, the gradient descent step is usually completed against a fixed target. In RL, a divergent update to a function approximator from one state transition can be further amplified in subsequent updates from the following state transitions; this leads to oscillatory or divergent behaviour in its learning behaviour. Fixed Q-Targets resolves this by optimizing with respect to a set of parameters which is only updated after a fixed number of steps.

In tandem, these two features allow both the target and the updated Q-values to be more stable, thus making the neural network be more stable in its learning. The neural network architecture in question is a convolutional neural net that receives the pixel input from four consecutive, pre-processed frames and outputs the Q-value for each action.

This paper is significant in the ability to stabilize Q-Learning with a function approximator; this enables solving RL problems with large state-spaces feasible.

Algorithm Two Deep Q-Learning with Experience Replay:

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

The gradient descent step with respect to the network parameters θ, in full:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{8}$$

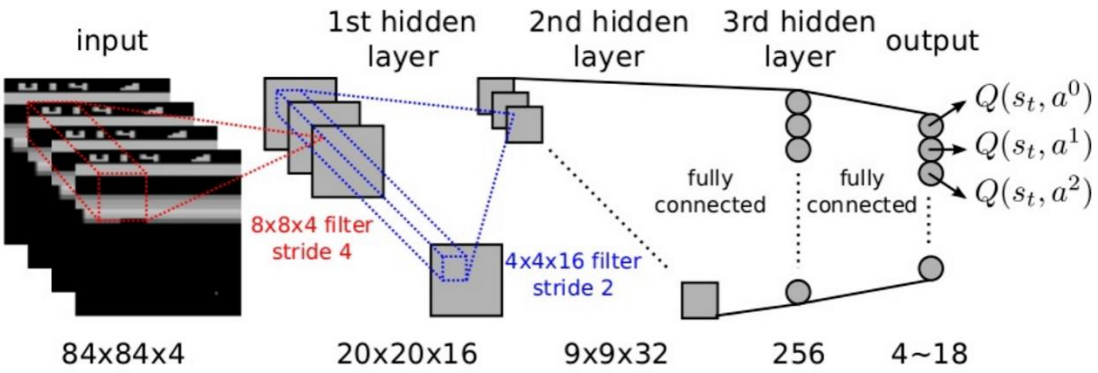

Fig. 2: The neural network architecture implemented in DQN [7]

3.3. Reinforcement Learning in Traffic Control Systems

3.3.1. Traffic Light Control with SARSA with Three State Representations

Research into controlling a TCS with Reinforcement Learning began over 20 years, with this paper [8] focusing on using SARSA to find an optimal policy. The paper used three different state representations to define the state-space: The first was vehicle count representation, achieved by summing the quantity of approaching vehicles for both phase grouping (east-west and north-south); each phase grouping was then one-hot encoded in a 10-bit vector representing ten data buckets for the quantity of vehicles, and each combination of buckets, alongside the currently activated traffic signal created a 10x10x2 tensor to represent the state space. The second representation was the fixed distance representation, dividing each of the four approaching lanes into equal partitions, and having a binary occupancy value for each. The result was an 8-bit vector, which alongside the current traffic signal, produced a 9-bit vector to completely represent the state.

The final representation, variable distance representation, is similar to the latter, with only the length of each partition modified to skew towards the intersection.

Each state representation, alongside a non-actuated TCS were compared; several metrics were used to measure the performance, including "the number of simulation steps required for all vehicles to reach their destinations; travel times for each vehicle; wait times for each vehicle; and the number of stops made by each vehicle," and found that the latter two state representations with SARSA performed the best.

Ultimately this research has some significant limitations, including minimal traffic volume and complete

knowledge of vehicle locations; however, it provided the foundations for further research.

3.3.2. An Open-Source Framework for Adaptive Traffic Signal Control

This paper investigated the use of a Deep Q-Network to control a TCS [9]. The state-space representation uses the density and queue length of incoming lanes at an intersection, as well as the most recent green phase. The Q-network receives this state observation as input, and selects the next action from a set of green phases.

The DQN TCS was compared against several other learning and non-learning algorithms, with the performance metric being the mean vehicle travel time, and at peak times performed better than every algorithm except the Max-pressure TCS, which also required the queue length as input. Interestingly, the DQN performed worse in low-demand times, which the author suggests could be due to the algorithm overfitting to periods where the magnitude of the reward is much greater. This paper shows to some degree the potential benefits of using DRL; further experimentation can be done to reduce the variance this algorithm produces.

4. Simulation

To conduct experiments on the impact of different Traffic Control System (TCS) algorithms on the throughput of traffic, the simulation environment was designed using SUMO, a traffic simulation software. Traffic simulation software uses mathematical modelling for the behaviour of vehicles in a network, allowing for the design and analysis of systems that might be too complicated for an exact solution to be found. SUMO uses car-following models, where each vehicle has its own dynamics individually programmed in the simulation [1]. To reconstruct the Ti Rakau Drive road network, the OSMWebWizard tool was used. OSMWebWizard is provided with SUMO, and uses geodata from OpenStreetMaps to generate a network file compatible with SUMO.

Fig. 3: A satellite image of the road network. Image credit: Google.

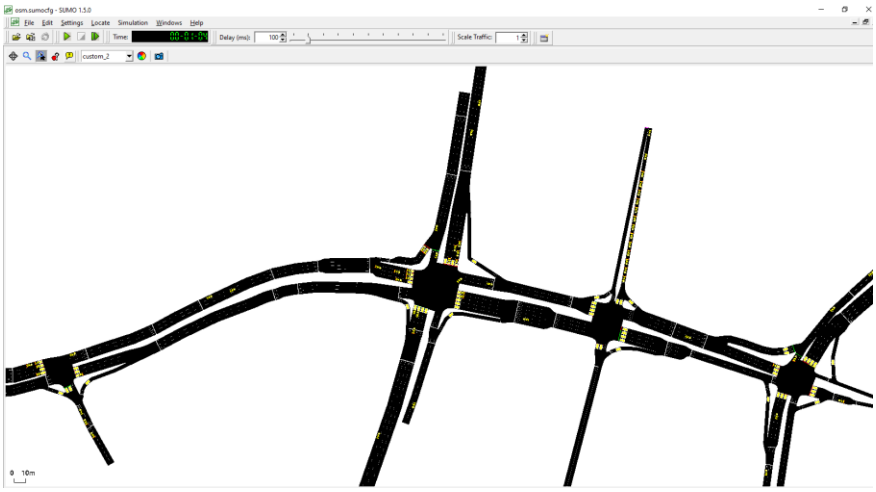

Fig. 4: The same road network generated in SUMO.

To populate the road network, real-world traffic data provided by Auckland Transport was used. This data was in the form of loop detector activations over five-minute intervals for a 24-hour period, for each loop detector, for each of the four intersections in the network. As SUMO requires specific routes for each vehicle, the traffic through each network-exiting lane was assumed to be distributed proportionately between networking-entering lanes that could reach it. Once the route distributions were calculated, individual vehicles were generated based on the traffic demand for each route. To achieve a realistic distribution of vehicles over time, the vehicles were generated by a Binomial distribution, for a discrete approximation of a Poisson process, so that the occurrence of each vehicle being generated remains independent from each other.

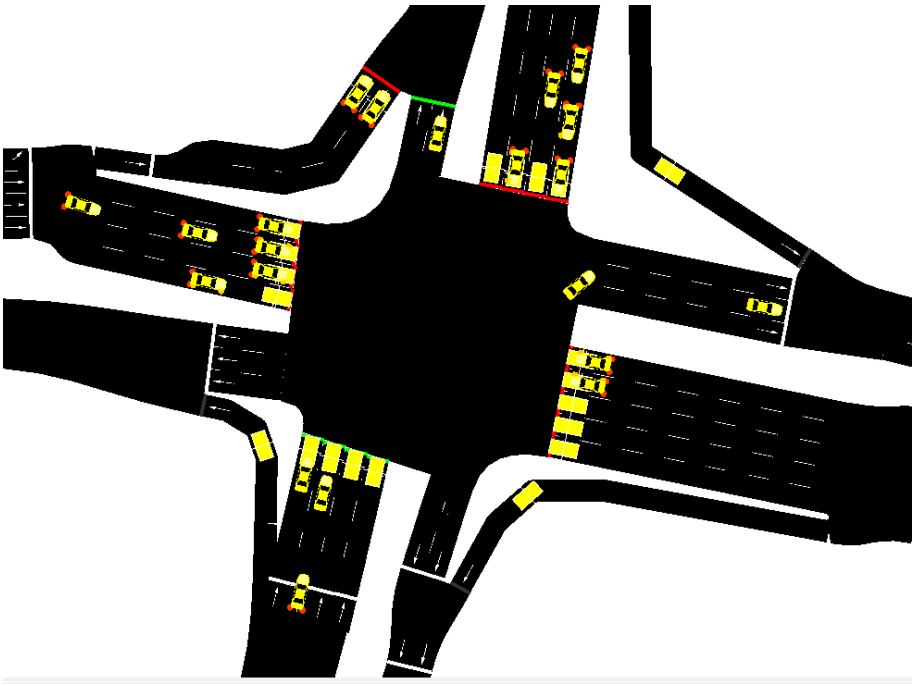

Fig. 5: Vehicles interacting with the road network during a simulation episode.

Running the simulation generates output files that are useful for understanding the performance of the traffic control system. One such file is a summary.xml file, which contains information regarding the number of vehicles 'halting' (where the speed of the vehicle is below 0.1 m/s, hereinafter referred to as vehicles 'stopped'), the mean travel time, the mean speed, and the mean speed relative to the current speed limit, for all vehicles, for each time step. Another output file is the tripinfo.xml file, which contains information on the time stopped, and the time loss (where the speed of the vehicle was sub-optimal with respect to the speed limit of the route), for each vehicle. This information is valuable for conducting analysis of each algorithms performance, and for the Reinforcement Learning process.

5. Algorithm Design

5.1. SCATS

Due to the proprietary nature of the SCATS algorithm, finding a model of SCATS for simulation proved difficult. Fortunately, alongside the traffic data, Auckland Transport also provided the exact real-time signal timings for the intersections. To recreate the SCATS algorithm, it was approached as a black box; only the

inputs of the vehicles activating the loop detectors, and the outputs of the signal phase and timings, were used. As the exact timing of the real-life loop detector activations were unknown, there is variance between the real-life vehicle timings and the probabilistic vehicle generation in the simulation, which limits the accuracy of the model. To combat this, supplementary TCS algorithms were included for a more reliable baseline with which the Reinforcement Learning solution can be compared with – this led to the implementation of the Earliest Deadline First algorithm.

5.2. Earliest Deadline First

The Earliest Deadline First (EDF) algorithm was adopted from this paper [3]. The EDF algorithm was tested on the same road network as in this project, and slightly outperformed SCATS on the test environment. The local-only variant, where each intersection acts independently, was used. This implementation of EDF has pre-emption, where the currently active phase can end early provided it does not detect traffic for three seconds. Each lane was recorded in one of multiple groups, where a group would contain all lanes that enter and exit the same roads for each intersection; groups that were non-conflicting could then be combined into phases, consistent with the current implementation. An illustration of these phases is shown below:

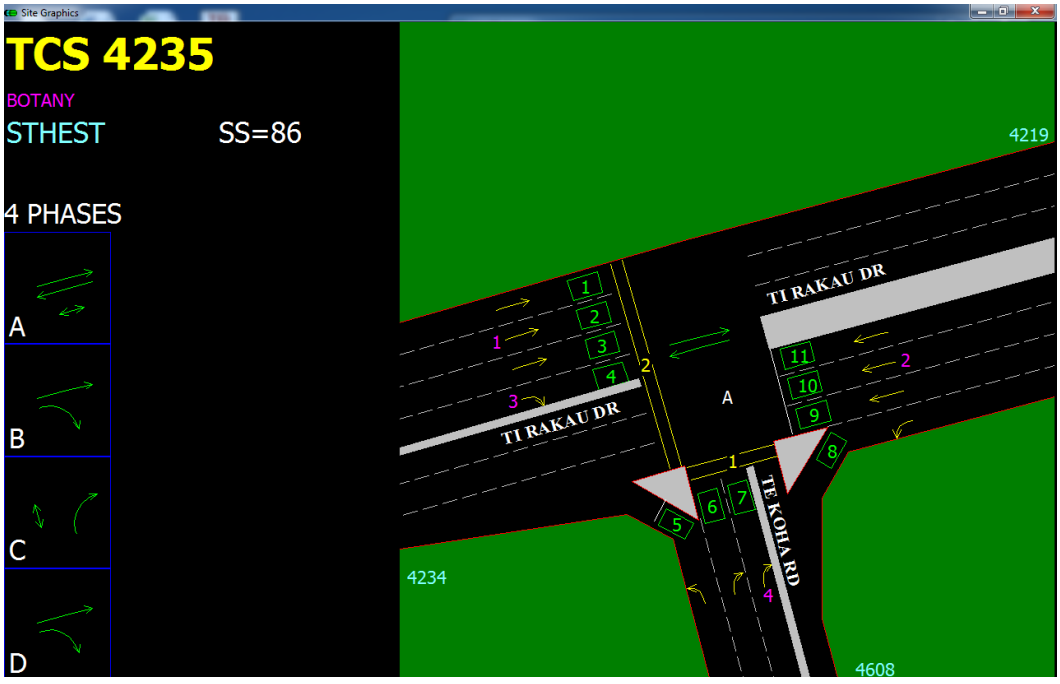

Fig. 6: Phases with the associated lanes for an intersection. Lanes 1, 2, 3 would correspond to one group. Image provided by Auckland Transport

The pseudocode for the algorithm is as follows:

Algorithm Three Earliest Deadline First:

```
phaseDuration = 0
minimum_traffic_activity = 3

For each step:
        Increment phaseDuration
        record time since loop detector activation of each active lane
                if not in transition:
                        if no active loop activations in minimum_traffic_activity or phaseDuration exceeds 180s:
                                Find the group which has waited longest
                                Find the 2nd group which has waited 2nd longest
                                Find the phase containing both groups
                                Switch to transition phase
                Else:
                        After 6s:
                                Reset phaseDuration
                                Switch to new phase
```

5.3. Queue Length Algorithm

Another algorithm that was implemented from the same paper was the Queue Length algorithm, which prioritizes the next phase based on the lane that possessed the longest queue. In practice, to implement an algorithm like this in a real-world scenario would require a more extensive sensor system to monitor the queues of each approaching lane. Testing this algorithm alongside EDF gave a more comprehensive analysis of scheduling algorithms, and provides a sanity check as they were found to achieve similar performance in this paper [3], when evaluated on the mean speed of vehicles.

5.4. Combined Queue Length and Wait Time Algorithm

A combined approach between EDF and the Queue Length algorithms was tested to see if any benefits could be gained by integrating both approaches. The wait time and queue length for each lane were multiplied by a weighting parameters, and the lane with the greatest weighted sum would be prioritized. A grid search for different values for the two weights was completed, with the optimal selection being 0.3 and 0.1 for the queue length and wait time respectively.

5.5. Q-Learning

5.5.1. Markov Decision Processes in the context of Traffic Control Systems

To implement a Markov Decision Process for the traffic environment, definitions of the states, actions, and rewards had to be determined. Selecting the state space is akin to feature engineering in supervised machine learning, so it is critical that the state variables selected provide enough information for the agent to be able to discriminate between the optimal and sub-optimal actions.

The state variables implemented include: the current active phase, the index of the longest waiting group, the time since vehicle activity was detected in the active phase (limited to nine seconds), the index of the second longest waiting group, the current active phase of the adjacent intersection, and the time spent in the current phase (limited to eight seconds).

For the actions space, an action can only be selected when the traffic control system is not transitioning to a new phase. Otherwise, for each step the available actions are to select the next phase, including the current phase to extend its duration. Initially, rewards were based on the negative average number of vehicles stopped during that action, consistent with this paper [9]; this was later changed to using rewards based on the mean speed, such as the mean speed of vehicles during an action and change in mean speed of vehicles between actions; using the mean speed as the reward gave a more holistic judgement on how effective the algorithm was, as only recording the vehicles stopped did not account for vehicles that had to decelerate as they approached a queue.

Different values for the discount factor were also tested across different experiments. The discount factor influences how near- or far-sighted the agent is. While generally this hyperparameter is set to 0.9, the rationale behind testing lower values was that prioritizing immediate rewards (i.e. serving stopped traffic) may provide a clearer reward signal for the agent; however, it was not found to significantly affect the performance or learning ability.

5.5.2. Exploration vs. Exploitation

While solving an MDP, the agent has to balance between exploring unknown actions, and exploiting the perceived optimal action. This is known as the exploration/ exploitation trade-off, and is a fundamental challenge in RL. If the agent only exploits, the policy may converge to a local maximum; if the agent only explores, the learning efficiency will be significantly diminished, and solving the MDP may become

impractical. Initially, an ε-greedy policy was used; ε-greedy ensures exploration by having an ε (where $0 \leq \varepsilon \leq 1$) chance of taking a random move, and exploits by having a 1-ε chance of taking the highest-valued action. Variants of the ε-greedy policy that were used include a fixed ε value, and an ε that decreases as training progresses; the motivation behind the latter is that the agent will be more likely to explore initially while the majority of the action-values are unknown, and then favour exploiting higher-value actions later in the training.

Another exploration policy that was investigated is the UCB1 algorithm, which can be described as 'optimism in the face of uncertainty.' Rather than selecting actions solely based on their action-value, the agent selects the action that maximizes the Upper Confidence Bound value:

$$a_t = argmax_{a \in A}\, Q_t(a) + U_t(a) \tag{9}$$

$$U_t(a) = \sqrt{(\frac{-\log p}{2\, N_t(a)})} \tag{10}$$

In doing so, actions that the agent has not tried often - and so are more likely to differ between the sampled and true means - will be preferred as the UCB value would be high; as that action is tried more, $U_t(a)$ will diminish to 0, and the action-value would be closer to its actual value due to the law of large numbers. Another benefit of using the UCB1 algorithm is that it does not require hyperparameter tuning while still maintaining sub-linear regret, as opposed to ε-based strategies.

5.5.3. Implementation

The learning process is completed by the repeated execution of several files. The RL-runner.py file executes one episode of simulation, following an ε-greedy policy using the stateActionValues.csv file. When the simulation completes, the summary.xml and stateActions.csv files are created. The stateActions file contains the state and action for each step of the simulation. The learn.py file uses the data from both output files to find the value of each state-action and reward pair, which is used to update the stateActionValues.csv file. Initially, this process was repeated 10 times with a non-zero ε value for exploration, and on the 11th episode completed the simulation following a greedy policy, where ε is equal to 0. The performance of this episode is recorded to measure improvements in the algorithm. This process was to be repeated until the algorithm

converged to the true optimal policy for the Q-learning implementation. A system diagram for the implementation is shown below. Due to the nature of the MDP, it was found that in practice the algorithm rarely converges, and tends to oscillate as minor updates to the action-value function can entirely change the trajectory the agent experiences during a simulation. Instead, the learn.py file was modified to not update the stateActionValues (and later, UCB.db) file if the previous policy performed better than a certain threshold over a set of test seeds.

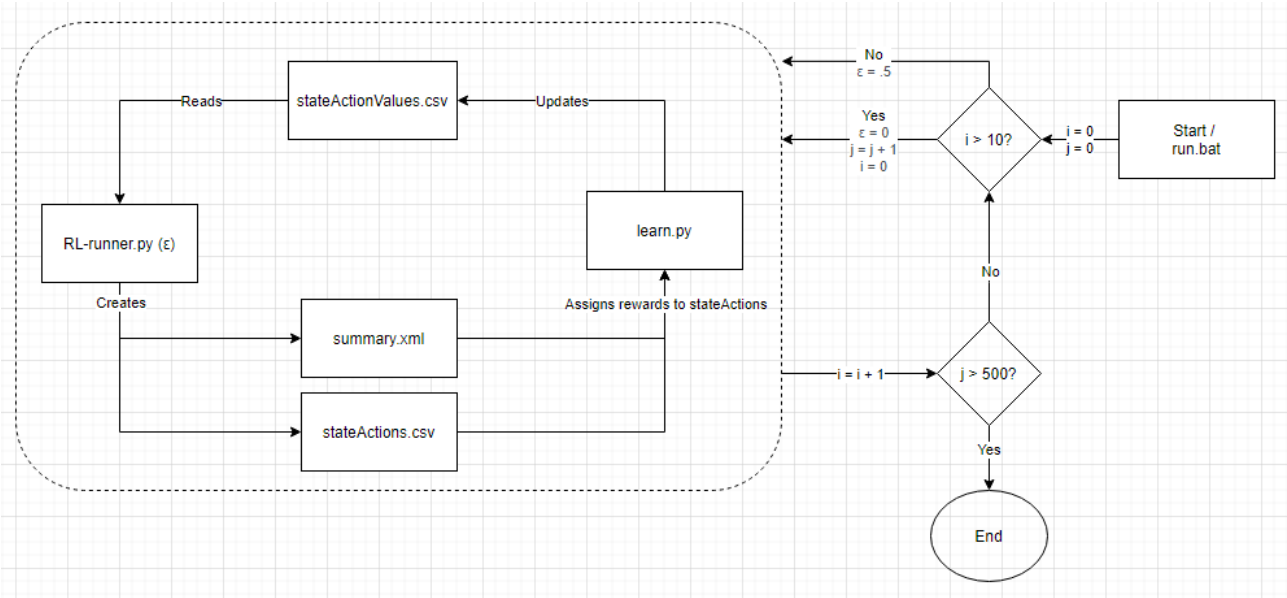

Fig. 7: The Reinforcement Learning process system diagram.

6. Results & Discussion

Testing of each algorithm was conducted using a 10-minute interval of traffic; the performance of each was evaluated by the total time loss and time stopped for vehicles for the first 10 minutes of simulation. The time loss for a vehicle is defined as the difference between the shortest duration to complete its journey under the speed limits on the path, and the actual time the vehicle took to reach its destination, in seconds. The time stopped for a vehicle is defined as the number of seconds at which the vehicle had a velocity below 0.1 m/s. These metrics are applied for all vehicles that complete their journey in a single simulation, and are automatically calculated and recorded in the tripinfo.xml file by SUMO.

6.1. SCATS

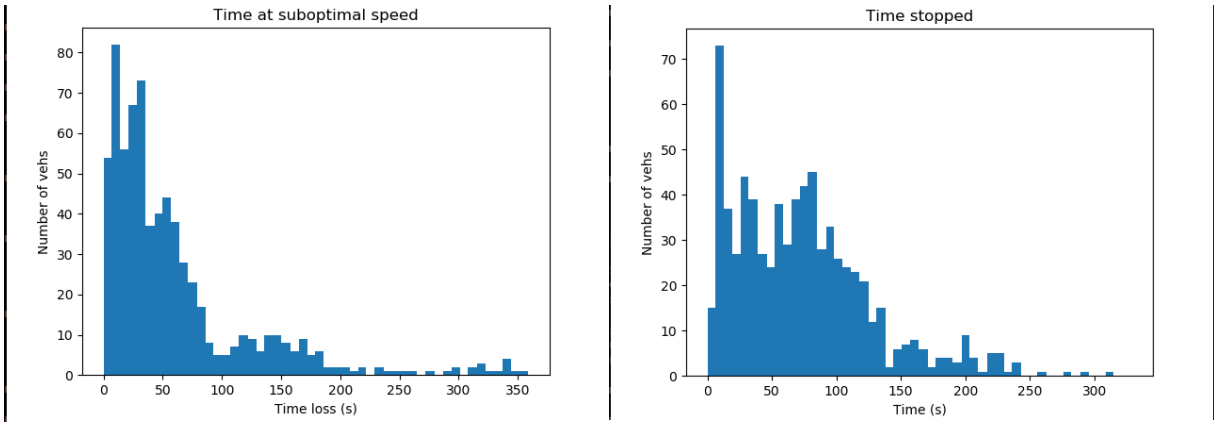

Fig. 8: Time loss and time stopped results for the SCATS algorithm.

For the time loss of the SCATS algorithm, the mean and standard deviation were found to be 73 and 55 seconds respectively. The mean and standard deviation of the time stopped were 53 and 50 seconds. The recreation of SCATS was unable to efficiently process traffic; this is likely due to the inconsistency between the real data collected and the model, and therefore was not considered as a suitable benchmark with which the Q-Learning algorithm could be compared with.

6.2. Queue Length Algorithm

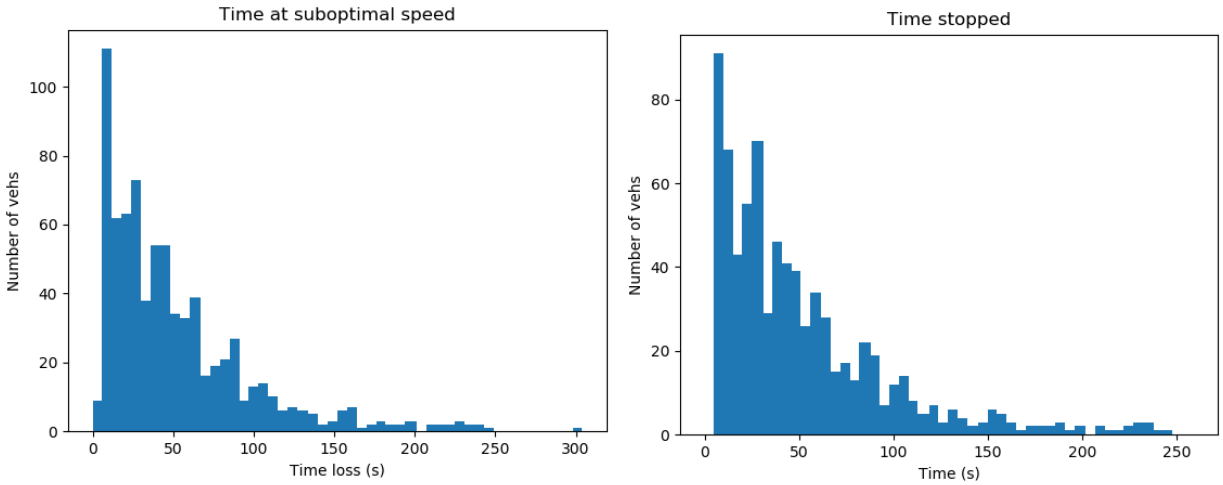

Fig. 9: Time loss and time stopped results for the Queue Length algorithm.

The Queue Length algorithm achieved a time loss mean and standard deviation of 52 and 47 seconds respectively. The mean and standard deviation were 35 and 42 seconds.

6.3. Combined Queue Length and Wait Time Algorithm

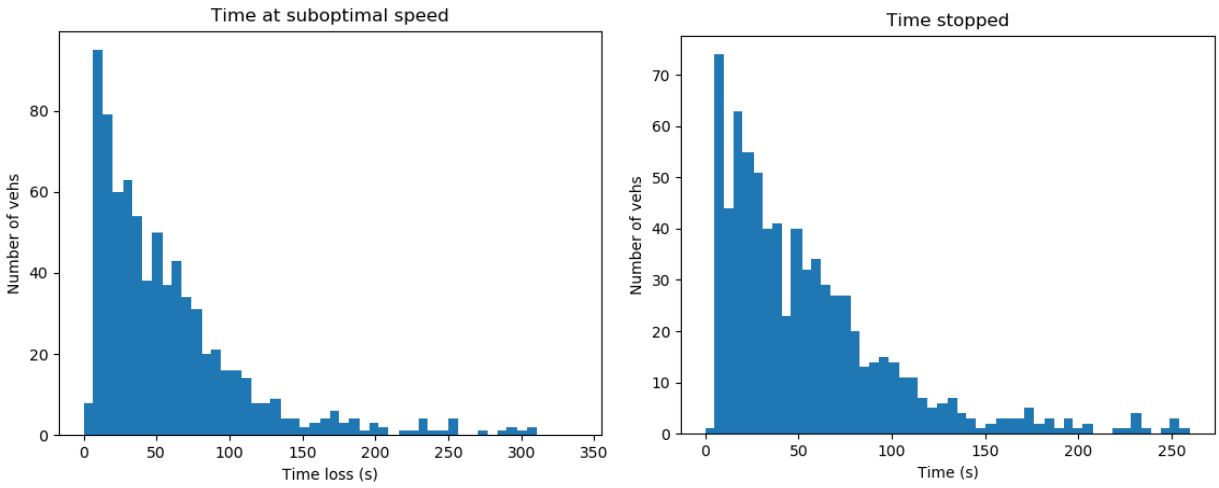

Fig. 10: Time loss and time stopped results for the Combined Queue Length and Wait Time algorithm.

For the Combined Queue Length and Wait Time algorithm, the mean and standard deviation for time loss were 58 and 53 seconds. The mean and standard deviation for time stopped were 38 and 46 seconds.

6.4. Earliest Deadline First

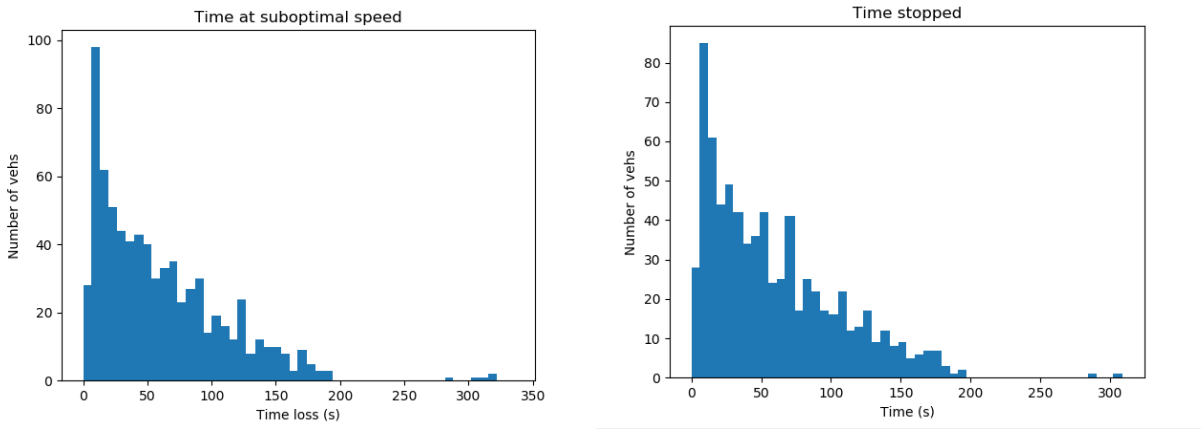

Fig. 11: Time loss and time stopped results for the EDF algorithm.

For the time loss of the EDF algorithm, the mean and standard deviation were found to be 49 and 43 seconds respectively. The mean and standard deviation of the time stopped were 31 and 41 seconds. As EDF was the most effective non-learning algorithm, the performance was measured over multiple trials so that it could be compared with the RL algorithm. The trials were completed using different seed values to generate the vehicles, and the results have been recorded in the table below:

	Seed Value	Time Loss mean	Time Stopped mean	Vehicles processed	Mean speed
	43	57	39	804	7.78
	44	51	35	768	7.57
	45	50	32	788	8.25
	46	54	36	773	7.87
	47	58	39	811	7.79
Average	-	54	36.2	788.8	7.852

Table 1: Performance of the EDF algorithm using different seed values

6.5. Q-Learning

The Q-Learning algorithm was applied to the intersection shown in Figure 5, while the other intersections used the EDF algorithm. The results of three significant experiments, each with different RL implementations have been recorded. As the reward for the three experiments were related to the mean speed, the performance and progress of each experiment were tracked. The performance was measured by the mean speed of vehicles for one episode, using the seed value it trained on. The progress of each experiment was tracked by the learning updates per episode, calculated by the sum of the absolute values of the TD-errors.

6.5.1. Experiment One

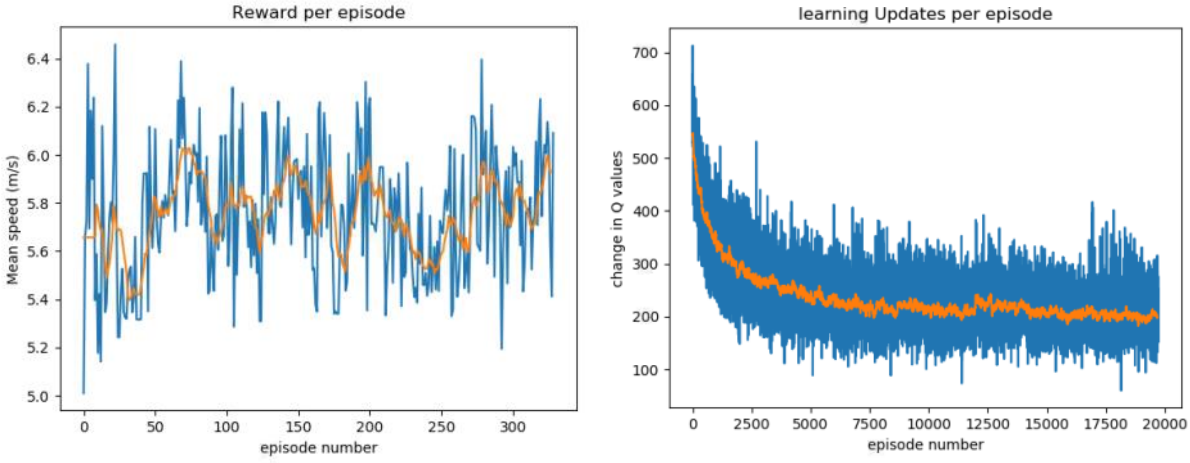

Fig. 12: Performance and progress of Experiment One.

This experiment used a state space defined by the longest waiting group, the current active phase, the second

longest waiting group, and the time since vehicle activity. The exploration policy used was ε-greedy, and the reward was the mean speed. It was determined that, due to the learning updates reaching an asymptote at such a high value, mean speed would not be a suitable reward signal. To understand why, consider a hypothetical intersection changing phase to serve one of two conflicting groups, that both have traffic occupying the respective loop detectors. The group that is not served could have one or 100 vehicles in a queue for that group – which would significantly influence the reward received for that action. This variation in the reward signal makes it significantly more difficult for the agent to find the optimal policy, as the best action could be constantly in flux for many states.

6.5.2. Experiment Two

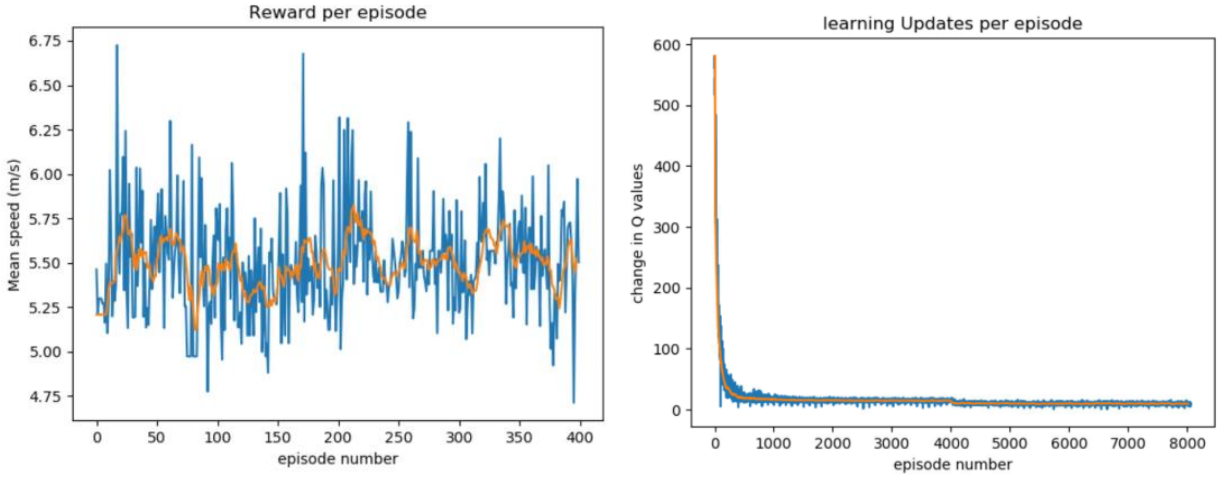

Fig. 13: Performance and progress of Experiment Two.

In the second experiment, the state space was defined by the longest waiting group, the current active phase, and the time since vehicle activity, also used ε-greedy for exploration, and the reward was the percentage change in mean speed between actions. The learning updates per episode reached a much lower asymptote, and was far less noisy, which would indicate that the majority of the action-values had been updated to their expected values. To use the previous scenario again, consider the new reward signal for serving the first group; if the conflicting group has one or 100 vehicles waiting in the queue, the reward provided would still be the same. By using the change in mean speed, the variance in the signal is greatly reduced, improving the agent's ability to learn. In this case, the agent did not improve as the state space was insufficient for the agent to learn.

6.5.3. *Experiment Three*

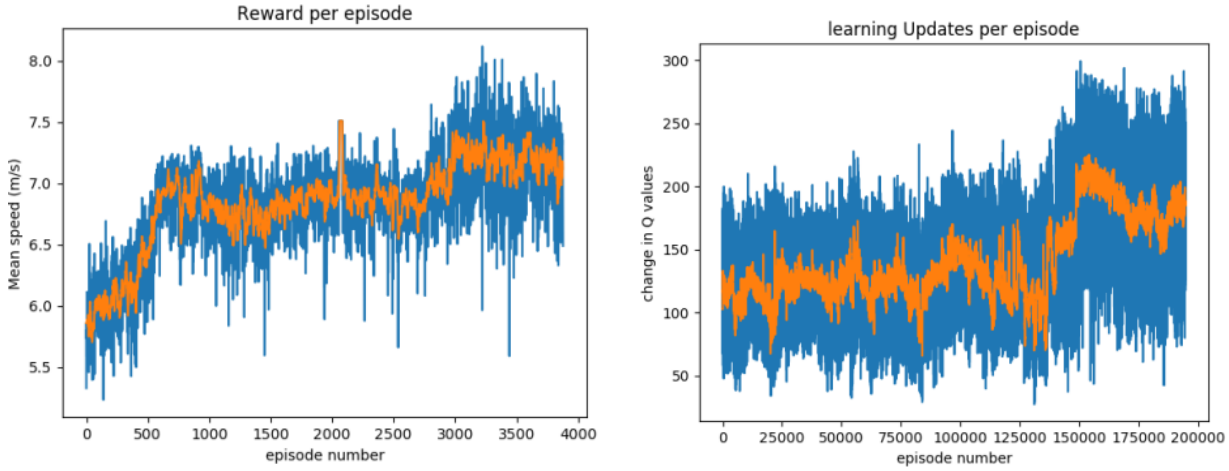

Fig. 14: Performance and progress of Experiment Three.

In the third experiment, the state space encompassed all of the state variables listed in Section 5.5.1. The reward was again the percentage change in mean speed between actions, and the exploration policy was UCB1. The learning updates for this experiment did not reach an asymptote, which would suggest that training had yet to be completed. The increased state space size, in combination with the optimistic exploration policy, resulted in an extensive training time for the agent; the learning updates progress indicates that training had yet to be completed. The agent did manage to achieve a 3% improvement in mean speed for vehicles over EDF, however it was unable to generalize this performance across the test set.

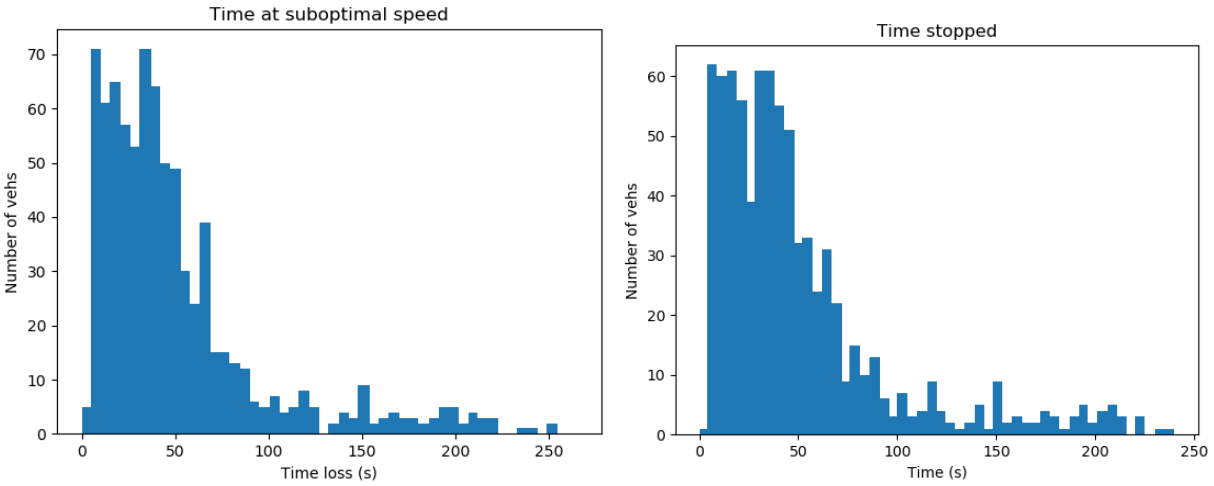

Fig. 15: Time loss and time stopped results for Experiment Three, test episode 3329.

The action-value function and UCB databases were captured mid-training, and the time loss and time stopped for the episode were evaluated on seed 42, one of the training seeds. The time loss mean and standard deviation for the agent were 51.7 and 47.8 seconds, and the time stopped mean and standard deviation were 33 and 41.7 seconds respectively; this was a 5% reduction in time stopped, and a 4% reduction in time loss. These results highlight this implementations ability to learn an effective policy, but a policy that overfits to the environment it trained in.

The cause of this inability to generalize to out-of-sample environments could be due to the observations being too simple for the agent to effectively learn. So far the RL problem has been framed as solving an MDP, though the problem could be considered as a partially observable MDP (POMDP); in a POMDP, the agent receives observations, and uses these observations to maintain a belief, or a probability distribution as to what the current state actually is. In this context, the agent only receives information from the loop detectors and the phase activations.

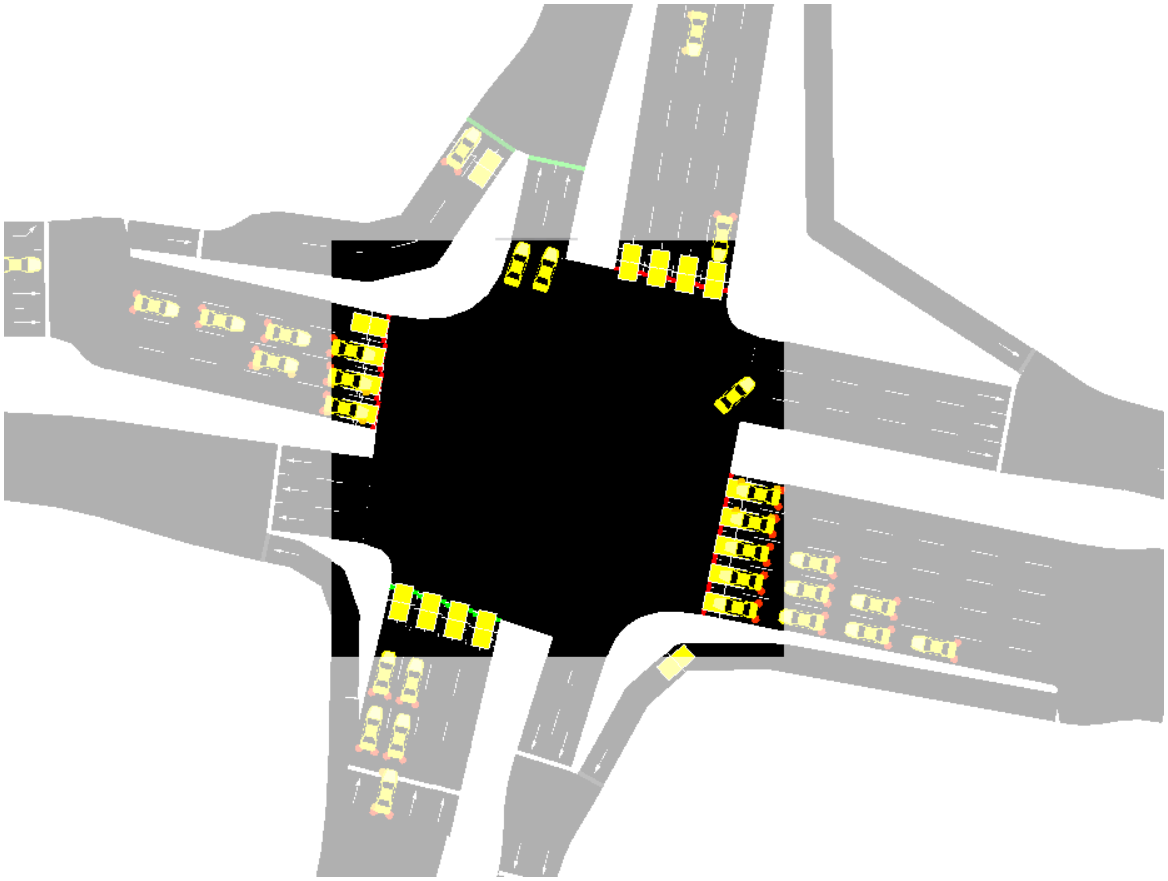

Fig. 16: An illustration of the information the agent receives.

As the incoming traffic is not considered, the agent may not have enough context to learn a policy that can generalize for a stochastic environment like traffic flow. This becomes an issue when the agent learns an expected value that is inconsistent with the test environment. Particularly for some of the states that are not commonly visited, the action-values for these states may differ between training and testing, as the traffic environments - and consequently the distributions of rewards - would vary despite being in the same state; this would lead to a sub-optimal policy when acting greedily with respect to the action-values the agent has learned through training. The difference in distributions is known as the Kullback-Leibler (KL) divergence, though in practice calculating this would require storing multiple rewards for any given action-value.

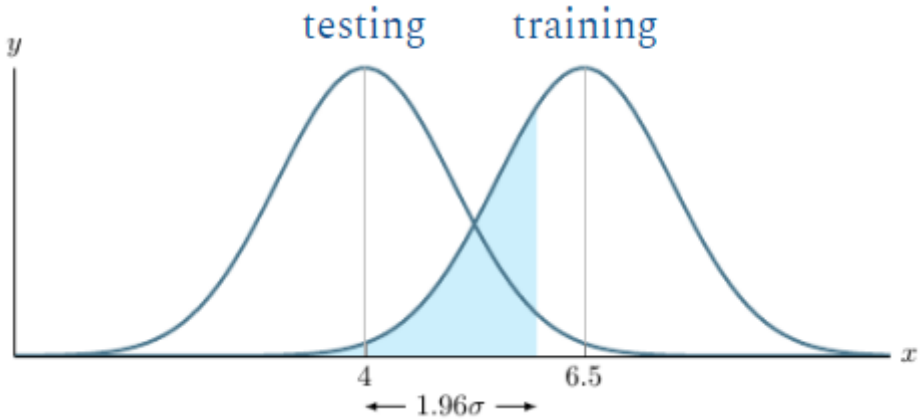

Fig. 17: Reward distributions for a single state-action. For illustrative purposes only.

7. Conclusion

From the results, it has been shown that the reinforcement learning systems implemented are not advantageous over non-learning algorithms for Traffic Control Systems. Under the constraint of only using information provided by the existing infrastructure, the RL implementations were unable to achieve a performance improvement over the non-learning algorithms across multiple test trials, despite achieving superior results on the environment it trained on. Further extensions to the RL approach can be explored to further improve its performance.

8. Future Work

For ease of implementation, comprehension and effectiveness, the EDF algorithm is the most suitable option;

in this paper [3] it was even found to improve upon the performance of the SCATS algorithm, and so may be worth investigating further by traffic infrastructure organizations such as AT. While the RL implementations were unsuccessful in out-performing the non-learning alternatives, with additional features it is possible that RL could achieve better results; however, this would likely be at the cost of ease of implementation and comprehension - this trade-off would have to be evaluated to determine if this approach is feasible. The first possible improvement to the RL implementation is to add more features, particularly related to incoming traffic. This has already been implemented to some degree, as some intersections have multiple loop detectors in separate locations along the same incoming lane, though this still can be improved upon. In practice, this would require the addition of cameras, sensors such as LiDAR, or more loop detectors around intersections, though priority could be given to areas that have the greatest vehicle density. This would likely benefit both non-learning and RL algorithms, though with greater complexity. For RL algorithms, the increased state space would likely make tabular Q-learning impractical, and may require the addition of a function approximator, such as an Artificial Neural Network. While there are more challenges associated with this, a deep reinforcement learning approach could generalize better, which could help it achieve superior out-of-sample performance. The RL infrastructure could also be scaled up to improve training; one example would be to use the IMPALA architecture which allows for distributed learning, which in turn would improve the wall-clock time, or elapsed real time, of the RL system to find an acceptable policy. Finally, with a faster RL system, the algorithm could be trained on more data, which would reduce the KL divergence between the training and testing reward distributions.

Acknowledgements

The authors of this paper would like to thank Dr. Avinash Malik for supervising the project, Dr. Minh Le Kieu, Dr. Henry Williams, Dr Jin Woo Roo, Moon Kim for their contributions towards this project, and Kipi Wallbridge-Paea for providing the traffic data.

References

[1] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. “Mastering atari, go, chess and shogi by planning with a learned model.” arXiv preprint arXiv:1911.08265, 2019.

[2] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. “Microscopic Traffic Simulation using SUMO.” IEEE Intelligent Transportation Systems Conference (ITSC), 2018.

[3] Mahmood Hikmet. “From worst-case Gini Coefficient to preemptive multitasking: consideration for fairness and efficiency in intelligent transportation systems.” 2018

[4] Sutton, Richard S.; Barto, Andrew G. (1998). Reinforcement Learning: An Introduction. MIT Press. ISBN 978-0-262-19398-6.

[5] Bellman, R.E. (2003) [1957]. Dynamic Programming. Dover. ISBN 0-486-42809-5.

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M.,

Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D.,

Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540):529–533.

[7] Yoon, Chris. “Vanilla Deep Q-Networks.” *Towardsdatascience.com*. Medium. 16 Jul. 2019.

[8] T. L. Thorpe and C. W. Anderson, “Traffic light control using sarsa with three state representations,” Citeseer, Tech. Rep., 1996.

[9] Genders, Wade & Razavi, Saiedeh. (2019). An Open-Source Framework for Adaptive Traffic Signal Control.

[10] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. “Microscopic Traffic Simulation using SUMO.” IEEE Intelligent Transportation Systems Conference

(ITSC), 2018.