Deep reinforcement learning for the optimization of traffic light control with real-time data

Monika Matyja, Jordi Morera, Sebastian Wolf *
Barcelona Graduate School of Economics
Universitat Pompeu Fabra and Universitat Autònoma de Barcelona
Barcelona, Spain
monika.matyja@barcelonagse.eu
jordi.morera@barcelonagse.eu
sebastian.wolf@barcelonagse.eu

Abstract

In this thesis we develop a traffic light control agent that can manage traffic lights with the objective to reduce traffic jams, trip time and other traffic metrics in a given network using reinforcement learning. To this end, we implement a Double Deep Q-Network algorithm and test its performance in controlling traffic lights on a 'small' and a 'large' traffic junction. We find that this algorithm beats a fixed traffic light phase program when traffic demand fluctuates, as it is capable of reacting to real-time traffic situations. The algorithm can be scaled up and holds promise to also perform well in controlling larger transport networks.

1 Introduction

Nobody likes traffic. Traffic decreases the productivity of urban spaces, and translates into lower prosperity. Controlling the amount of traffic is therefore a priority goal for urban planners. Yet, most cities face natural or financial constraints in the expansion of transport networks. Consequently, urban planners are confronted with the challenge of optimizing traffic flows within the constraints of existing infrastructure, to achieve the most efficient use of it possible. To optimize traffic flows without infrastructure modifications, urban planners can only resort to traffic control tools such as traffic lights, or traffic signs and rules.

The control sequences of traffic lights can have large impacts on traffic flow, as has been well documented for example in Pande and Wolshon [2016]. However, determining the right timing of traffic light phases can grow complex when traffic demand fluctuates and the timing of traffic light phases on one intersection affect traffic in the rest of the network [Hall, 2006]. In practice, traffic planners often simply rely on heuristics: study the typical traffic situation, and set the phase timing in advance. This is problematic when demand fluctuates. To address this, many cities have installed hardware that helps to measure traffic flow in real time, such as induction loops that detect vehicles passing over a certain point of a road, or cameras that provide a live feed of the traffic. With this information cities can optimize the traffic light signal phase times and set them dynamically based on the time of the day, or based on the length of car queues on roads detected by induction loops or traffic cameras.

Even where real-time information about the traffic situation is available, finding the traffic light phase timing that yields optimal traffic flow can be difficult. Sophisticated approaches using real-time information that adapt phase timing based on current demand have been developed. These approaches

*We thank Hrvoje Stojic and Anestis Papanikolaou for comments that helped improve this manuscript and supervision of this thesis.

Master's Thesis submission for the Degree of Master of Data Science.

require a function that maps inputs such as a count of the number of cars queued in different lanes to a certain traffic light phase timing.

At heart, the problem requires inter-temporal optimization of an objective function. Traditionally, this has been the domain of the field of DP. DP approaches to traffic control include the SCAT method proposed by Sims and Dobinson [1979], the 'SCOOT' method proposed by Hunt et al. [1981], the OPAC method proposed by Gartner [1983], RHODES proposed by Head et al. [1992], Sen and Head [1997] and the 'TUC'- strategy proposed by Diakaki et al. [1999]. However, these strategies all rely on a simplified representation of the environment in order to achieve analytic tractability, yielding suboptimal results. They are also poorly scalable, given their computational complexity. As such, DP approaches thus far have yielded in-satisfactory solutions particularly for the control of larger traffic systems.

With the recent breakthroughs in tackling complex, sequential problems with goal-oriented algorithms, new frameworks to find traffic control solutions have been put into the spotlight. In this thesis we propose to use reinforcement learning to optimize traffic light phase times for traffic control problems where real-time information on the traffic situation is available via induction loops or cameras. Reinforcement learning promises to overcome several of the challenges that dynamic programming approaches have faced in traffic control. Fundamentally, the main advantage of the reinforcement learning approach lies in the fact that it offers tools that do not require a model of traffic flow, which often provide poor approximations or can be costly to obtain. Determining traffic control without a model of traffic flow holds the promise of providing solutions for situations where dynamic programming approaches are intractable, and only sub-optimal numerical approximations could be found using dynamic programming tools.

Previous research into reinforcement learning techniques for traffic control includes El-Tantawy et al. [2014], Genders and Razavi [2016], van der Pol [2016], van Dijk [2017], Genders and Razavi [2018] and Liang et al. [2018]. Different types of reinforcement learning algorithms have been tested, assuming different levels of knowledge about the traffic situation. In this thesis we adopt a state-of-the-art Deep Double Q-Network algorithm that is well-known from Mnih et al. [2015], where it was demonstrated to beat humans in the game of Atari. One advantage of our approach over previous approaches is that it requires very little knowledge of the real-time traffic situation. Whereas other proposed solutions require top-down imagery of the traffic situation, which is not reliably available for many cities, our approach can be implemented with the hardware already available in many modern cities. We simulate our approach using two types of traffic intersections and obtain results that clearly beat a fixed traffic control policy.

In the next section, we provide an overview of the field of reinforcement learning and the main algorithms so-far proposed and used in practice. In section 3, we review previous work to employ reinforcement learning techniques to solve traffic control problems. In section 4, we introduce our approach, and then discuss the results we obtain in section 5. We conclude in section 6.

2 The field of reinforcement learning

The goal of reinforcement learning is to study how algorithms can be used to make optimal decisions in a dynamic environment. As we explain this further, we follow Silver [2015] and Sutton and Barto [2018] as our main references.

2.1 Framework

Reinforcement learning problems can be phrased in terms of:

1. an **agent** (an algorithm) that observes
2. a **state**
3. of the **environment** (perfectly or imperfectly observed)
4. and takes a an **action**
5. and finally receives a **reward** (positive or negative) for the action taken.

The agent is a decision maker that is set to live and interact with the environment. As the agent interacts with the environment, the environment reveals an observation of the state of the world to

the agent. This gives the agent information on which to base her/his next action. An agent, after having taken an action, receives a reward from the environment. The size and sign of the reward are a signal to the agent that reveals whether the state of the world that it reached is good or bad. Following a certain policy, or by trial and error, the agent finds best action responses to the states that the environment presents it with, aiming to maximize the cumulative reward it will receive.

A state s is a complete description of the state of the world. However, the state of the environment may not always be perfectly observed, therefore, reinforcement learning algorithms often decide on their actions based on the agent state, which is the state as (imperfectly) observed by the agent, rather than the environment state.

The agent is allowed to take an action a that belongs to the action space A. The action space can be discrete or continuous. The former allows only a finite number of moves and the latter defines actions as real-valued vectors in the space. This distinction has some quite-profound consequences, as some algorithms can only be directly applied in the discrete case, and others work in the continuous specification of the action space.

The definition of reward is fundamental for the performance of reinforcement learning algorithms. It is the objective function for the agent. The reward formally depends on the current state of the world s_t, the action a_t just taken, and the next state of the world s_{t+1}.

$$r_t = R\left(s_t, a_t, s_{t+1}\right) \tag{1}$$

The most common type of return across the reinforcement learning literature, and the type of reward as we will define it for our purpose, is the infinite-horizon discounted return, which is a discounted sum of rewards:

$$R = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2}$$

where $\gamma \in (0, 1)$.

2.1.1 Markov Decision Processes

Having introduced the concept of the state and action, one can define a trajectory τ to be a sequence of states and actions in the environment.

$$\tau = (s_0, a_0, s_1, a_1, \ldots) \tag{3}$$

The state transitions are determined by the environment, and depend on only the most recent action, a_t. One refers to the state transition as everything that happens to the world between the state at time t, i.e. state s_t, and the state one period later at $t + 1$, s_{t+1}.

Similarly, as with continuous and discrete action space, the definition of the trajectory will depend on the problem at hand. One can differentiate between deterministic transition functions:

$$s_{t+1} = f\left(s_t, a_t\right) \tag{4}$$

where the function f is well-defined and determines the evolution of the trajectory. On the other hand stochastic trajectory is defined as:

$$s_{t+1} \sim P\left(\cdot | s_t, a_t\right) \tag{5}$$

In the stochastic case, the definition of the next state is following a probability distribution. A policy is a set of rules used by an agent to decide what actions to take, a mapping between states and actions. Equivalently to the aforementioned trajectories, a policy can be deterministic and stochastic:

$$\pi : s_1, a_1, s_2, a_2, \ldots, s_t \mapsto a_t \tag{6}$$

A crucial property of MDPs in reinforcement learning is the Markov property. It defines the state to be Markov if and only if:

$$\mathbb{P}\left[s_{t+1} | s_t\right] = \mathbb{P}\left[s_{t+1} | s_1, \ldots, s_t\right] \tag{7}$$

This concept revises and specifies the former definition of a trajectory. For a Markov state s_t and successor state s_{t+1} the state transition probability (transition function) can defined by:

$$\mathcal{P}\left(s_{t+1} | s_t\right) = \mathbb{P}\left[S_{t+1} = s_{t+1} | S_t = s_t\right] \tag{8}$$

This individual state transition probabilities give rise to a state transition matrix, where changes from all states to all other possible states are tracked:

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \tag{9}$$

Summing all those concepts, a Markov process is a finite sequence of states equipped with a state transition matrix defined above. In the literature is it often described as a memoryless random process because it does not use the historical information about the states. The $\mathcal{P}$ completely characterizes the environment's dynamics. The addition of rewards to the actions, gives raise to one of the most basic reinforcement learning settings, where every transition results in a numerical reward:

$$\mathcal{P}\left(s_{t+1}, r_{t+1} | s_t\right) = \mathbb{P}\left\{S_{t+1} = s_{t+1}, R_{t+1} = r_{t+1} | S_t = s_t\right\} \tag{10}$$

with the reward is defined as:

$$\mathcal{R}_{s_t} = \mathbb{E}\left[R_{t+1} | S_t = s_t\right] \tag{11}$$

A Markov process with rewards and a discounting factor for those rewards can be called a Markov Reward Process. Having the rewards incorporated in the model, one can define the total discounted reward from time-step t, which is equivalent to an accumulated reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{12}$$

where R is the reward and γ is the discount factor.

A Markov decision process (MDP) is a Markov reward process with decisions. It is an environment in which all states are Markov. In an MDP the action space is assumed to be finite. It results in slight differences in the formulas being contingent on the action that has been taken. Mathematically, the transition probability changes into:

$$\mathcal{P}\left(s_{t+1} | s_t, a_t\right) = \mathbb{P}\left[S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t\right] \tag{13}$$

and the reward function into:

$$\mathcal{R}_{s_t}^{a_t} = \mathbb{E}\left[R_{t+1} | S_t = s_t, A_t = a_t\right] \tag{14}$$

Finally, defining the environment to be a finite MDP is equivalent to assuming that its state, action, and reward sets, S, A, and R, are finite, and that its dynamics are given by a set of probabilities defined by the state transition matrix, $\mathcal{P}$, for all $s \in S, a \in A$ and $r \in R$.

2.1.2 Value and action-value functions

The value function is an estimator of how good the realized state s_t is, given the chosen policy π.

State-value function for policy π:

$$\begin{aligned} v^{\pi}(s_t) &= \mathbb{E}_{\pi}\left[R_t | S_t = s_t\right] \\ &= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s_t\right] \end{aligned} \tag{15}$$

The Q-function describes the value of choosing an action a_t according to a policy π, in the state s_t.

Action-value function for policy π (Q-function):

$$\begin{aligned} Q^{\pi}(s_t, a_t) &= \mathbb{E}_{\pi}\left[R_t | S_t = s_t, A_t = a_t\right] \\ &= \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s_t, A_t = a_t\right] \end{aligned} \tag{16}$$

2.1.3 Bellman equation

Reinforcement learning aims to maximize the cumulative return of an MDP. This is done via the Bellman equation, which provides a representation of the inter-temporal trade-off that needs to be taken into account at any decision step of an MDP. To do so, it decomposes the previously introduced value function into two parts: the immediate reward and the discounted value of successor states to directly observe that optimality amongst sub-problems will influence the optimality of the final solution.

$$\begin{aligned} v^{\pi}(s_t) &= \mathbb{E}_{\pi}\left[G_t | S_t = s_t\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s_t\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \ldots\right) | S_t = s_t\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma G_{t+1} | S_t = s_t\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma v^{\pi}\left(S_{t+1}\right) | S_t = s_t\right] \end{aligned} \tag{17}$$

The Bellman equation allows for recursive decomposition. This is important in the sequential problem solution setting because, it helps express the values of states as a function of values of other states. More precisely, once we know the value of s_{t+1}, we can compute the value at the state s_t. The Q-function can be decomposed in a similar way:

$$\begin{aligned} Q^{\pi}(s_t, a_t) &= \mathbb{E}_{\pi}\left[G_t | S_t = s_t, A_t = a_t\right] \\ &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma Q^{\pi}\left(S_{t+1}, A_{t+1}\right) | S_t = s_t, A_t = a_t\right] \end{aligned} \tag{18}$$

2.2 Algorithms

2.2.1 Taxonomy of methods

The number of new algorithms for reinforcement learning has proliferated over recent years and thus a concise up-to-date dictionary of the methods is not well-established. The list of algorithms that have been proposed is long, hence we will only introduce the algorithms that we deem to provide essential background for the final model choice. We first introduce three key concepts used to classify these algorithms.

Model free vs model based Reinforcement learning algorithms can be divided into two broad classes: model-based and model-free algorithms. Model-based algorithms rely on a "known" MDP with full information about state transitions and rewards. In contrast, model-free algorithms assume no prior knowledge of MDP transition functions. Model-free methods learn directly from episodes of experience. Model-based reinforcement learning that learn from experience also exist, but they use this experience to "construct an internal model of the transitions and immediate outcomes in the environment" [Dayan and Niv, 2008]. For traffic control problems, model-free methods have the appeal that they allow approaching problems where the environment cannot be modelled well, or where models of the traffic situation are analytically intractable.

On-policy vs off-policy based The difference between off-policy and on-policy methods lies in their updating of Q-values. Off-policy methods update the estimates of the reward for state-action pairs assuming a policy, even when that particular policy is not actually followed. On-policy methods, in turn, make updates only for the policy they follow.

Exploration vs exploitation Exploration allows the agent to get to know the state-action space by taking random actions. During exploitation, in turn, an agent takes actions that are predicted to yield the highest return; she/he acts greedily. The choice how much an agent should explore the environment, and how much it should exploit its current knowledge is usually hard to determine. Setting it wrongly may cause the system to be unstable or stuck in a local optimum instead of a global one. Allowing no exploration, in turn, will never expose the agent to discover the global optimum.

2.2.2 Dynamic programming

Dynamic programming is the ancestor of reinforcement learning and the archetypal example of a model-based approach. Dynamic programming usually works under the assumption of a perfect model of the environment as an MDP, where all possible state transitions are known. If the environment can

be modelled accurately, then DP offers well-developed solution approaches based on breaking the inter-temporal optimization problem into a collection of simpler sub-problems [Bertsekas, 2005].

Although DP in a sense is the foundation of reinforcement learning, it has its major limitations. In reality it is often impossible to gain a perfect model of the environment. Additionally, DP problems can be computationally expensive, especially when the state and action spaces are large. This is very often the case when modelling real-world processes, as for example in traffic control.

2.2.3 Monte Carlo

Monte Carlo methods represent the most basic model-free algorithms and are fully based on sampling sequences of states, actions, and rewards and averaging the value function over multiple sample trajectories to find the best policy. The value function is approximated by the mean returns of an episode. Due to the fact that means are computed episode-wise, one can only apply Monte Carlo methods to MDPs for which all episodes eventually terminate (episodic). Monte Carlo methods are infeasible when the state space-action space is large, as they rely on full exploration of this space.

2.2.4 Temporal-Difference learning

A natural modification of the Monte Carlo method are TD learning algorithms that combine DP and Monte Carlo ideas. Similarly to the latter, they do not require a model of the environment to find the optimal policy. However, in contrast to Monte Carlo, TD methods do not rely on exploring the entire state-action space. They regularly update their estimate of the value of a given state as they explore the environment, a feature that draws directly from the DP methodology. Additionally, TD methods do not require MDPs to be episodic. TD works in continuing (non-terminating) environments [Tsitsiklis and Van Roy, 1997]. TD suffers from less variance compared to MC methods, providing more stability; a feature that is of particular importance in a traffic control scenario. Both on and off-policy TD algorithms exist. The off-policy TD control algorithm Q-learning directly learns the optimal policy, independent of the policy being followed. This stands in contrast to on-policy algorithms like SARSA. The off-policy nature of Q-learning significantly simplifies the analysis of the algorithm for our purpose.

2.2.5 Tabular Q-learning

Tabular Q-learning is the most basic form of Q-learning algorithm [Watkins and Dayan, 1992]. The Q-learning algorithm directly estimates the value of an action a in a given state s using a lookup table. This value is usually referred to as the Q-value of the s; a-pair, Q(s; a), which is stored in the Q-table. In Q-learning actions are selected based on the largest Q-value for the current state, as per a Q-value lookup table. After an action is taken, the Q-value for the action-state pair of the actions taken is updated. The updates happen iteratively, as follows:

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha \left[R_t + \gamma \left[\max_a Q_t \left(s_{t+1}, a \right) \right] - Q_t(s_t, a_t) \right] \tag{19}$$

where a is the set of all possible actions at the new state. Hence, the estimated future value of Q at the state s_t and action a_t is the difference between the current estimate of the state-action pair, and the actual value of it. Important to note is that there is no upfront information about the true value of the pair, hence the current reward signal and the maximizing Q-value of the next state is used as an approximation of the true value. Overall, tabular Q-learning works very well in the discrete state spaces, unfortunately, lookup-table based learning cannot be applied in settings with continuous or even large discrete state-action spaces.

2.2.6 Q-learning algorithm

To apply Q-learning in problems with continuous or large discrete state-action spaces, we can attempt to estimate the Q-function directly using function approximation. When we approximate the Q-function using a neural network, we talk about Deep Q-learning (DQN).

Q-learning convergence issues Studies like Watkins [1989], Watkins and Dayan [1992] and Even-Dar and Mansour [2003] show that original convergence guarantees of Q-learning no longer hold when using Q-learning with function approximation. Reasons behind this result are that samples to fit the estimation function are correlated, that the sampling distribution is non-stationary and that we

are using the same function to predict Q-values and to evaluate them. Several remedies have been proposed. We briefly discuss two: experience replay and target freezing.

Experience replay, first proposed by Lin [1992] and then applied to Q-learning by Mnih et al. [2015], is a technique that breaks the correlation between samples. To do that it stores transitions at each step and updates the Q-function approximation with a random sample of these transitions. As we use a non-correlated sample of transitions, the method removes the dependence of successive experiences on the current weights and eliminates a source of instability.

As in other methods that bootstrap, the target for a Q-learning update depends on the current action-value function estimate. When a parameterized function approximation method is used to represent action values, the target is a function of the same parameters that are being updated. This dependence complicates the process compared to the simpler supervised-learning situation in which the targets do not depend on the parameters being updated. This situation can lead to oscillations and/or divergence. To address this problem, Mnih et al. [2015] use a technique that brought Q-learning closer to the simpler supervised-learning case while still allowing it to bootstrap. The method works as follows: whenever a certain number of updates have been done to the Q action-value network, the current Q network weights are transferred to another network, Q^{frozen}, and held frozen for the next M updates of Q. The outputs of Q^{frozen} are then used as the Q-learning targets. The resulting update rule modifies the one in equation (19) as follows:

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha \left[R_t + \gamma \left[\max_a Q_t^{freezed}(s_{t+1}, a) \right] - Q_t(s_t, a_t) \right] \tag{20}$$

Q-learning function approximations are biased. This bias stems from the use of the max operator in Q-function approximation and causes the Q-learning to suffer a large performance penalty. The bias is caused by the simple fact that $\mathbb{E}[max(x)]$ is not a good estimator for $max\ \mathbb{E}[x]$. In order to avoid this, another approximation within the approximation is suggested - this is where the Double Q-learning name originates [van Hasselt, 2010].

2.2.7 Double Q-Learning

To overcome the biasedness of Q-learning with function approximation, the double Q-learning algorithm (DDQN) stores two Q functions: Q_A and Q_B [van Hasselt, 2010]. Each Q function is updated with a value from the other Q function for the next state. It has been shown that the updated algorithm satisfies convergence and out-performs Q-learning in cases where overestimation is especially likely [Mnih et al., 2015]. The update for DDQN is stated below:

$$Q^A(s_t, a_t) \leftarrow Q^A(s_t, a_t) + \alpha(s_t, a_t) \left(r + \gamma Q^B \left(s_{t+1}, \arg\max_a Q^A(s_{t+1}, a) \right) - Q^A(s_t, a_t) \right) \tag{21}$$

Note that since Q^B was updated with experiences from the same environment, but with different samples, it can be considered an unbiased estimate for the value of this action. At each network update, one of the two networks is chosen randomly and then updated, so indices A and B are interchangeable depending on the network we chose to update.

Although freezing targets and double Q-learning can be used together, it is important not to mix up double Q-learning with target freezing explained in previous section. Note that the key idea here is how the updating is done.

The algorithm that will be used in this study is an adapted version of the Double DQN algorithm proposed in van Hasselt et al. [2016] which has as ingredients the algorithms explained in this section. In section 4 we provide the pseudo-code for our implementation.

3 Reinforcement learning in traffic control problems

As discussed in section 2, reinforcement learning crucially relies on the specification of several key components that together define the reinforcement learning problem; the environment function, the state space, the action space, the reward function and the agent. El-Tantawy et al. [2014] provide a review of previous research to use reinforcement learning for traffic control, including an overview of environment, state-action space, reward and agent specifications. As their main contribution, they provide a through comparison between a range of specification choices, and report the sensitivity of results to these design choices.

El-Tantawy et al. [2014] compare a Q-learning, SARSA and TD(λ) algorithm, and test them by running simulations on a set of intersections in Toronto. In their simulations, Q-learning and SARSA yield similarly good performance, and both beat an adaptive traffic signal control benchmark policy. In terms of state representations, they find that using the queue lengths and the number of arrivals to a green phase yields the best performance across all their experiments. Regarding the reward function, they find that the reduction in the cumulative delay yielded minimum average delay, minimum average queue length, and minimum average number of stops. In terms of action selection policies, they find that a mixture of ϵ-greedy and softmax methods results in fastest convergence and best online performance. However, in reviewing [El-Tantawy et al., 2014], it is important to note that many of the best performing reinforcement learning techniques were proposed in the years since.

Following the roots of the DQN algorithm, Genders and Razavi [2016] propose to represent the state space in a traffic coordination problem using imagery. The authors propose specifying the state as the position and speed of each vehicle combined in a large state matrix that can be derived from satellite or aerial imagery. This approach can broadly be described as the discrete traffic state encoding method. It encodes arguably the most important information about traffic participants movements in a dense state matrix that provides the agent with knowledge about the approximate current position of all vehicles in the network and the ability to infer where they will be in the immediate future, given their speed. Each position in the state matrix is a car. Traffic lights and their current phase are also included in the state matrix. The authors specify the agent itself using a deep convolutional neural network trained by Q-learning with experience replay that is rewarded according to the change in cumulative vehicle delay between actions. The agent is punished if the total delay across lanes increases or rewarded if it decreases. The authors record major improvements using this complex state specification in comparison to a one hidden layer neural network traffic signal control agent with simpler state specification. More precisely, average cumulative delay, average queue length and average travel time was reduced by 82%, 66% and 20%, respectively. While this is an interesting and innovative solution, real-time top-down imagery of the traffic situation may be lacking in many real world situations.

A similar approach has been studied by van der Pol [2016], who also uses top down imagery to derive a state representation but focused on the coordination of several agents to control a traffic network. Each single agent is defined to control a single intersection within a traffic network, trained using a Double Deep Q-Network algorithm. The author specifies the reward function as a weighted sum of the waiting time, delay, emergency stops, switches and teleports. To test this approach, the author compares the performance of a single and multi-agent setup with a linear agent as baseline, recording good results.

In a later contribution, Genders and Razavi [2018] propose to use a simplified state specification, this time with a clear emphasis on the real-world implementations of the solutions Instead of including the position and speed of every single vehicle in the network, they use aggregate measures such as lane occupancy and average speed to define the state space. They maintain the definition of the reward function and action space introduced in [Genders and Razavi, 2016]. Instead of a convolutional neural network, they use a neural network with a single hidden layer and a relu-activation function to approximate the Q-function. They find that the low-resolution state representations (e.g., occupancy and average speed) perform almost identically compared to high resolution state representations (e.g., individual vehicle position and speed). This suggests that the implementation of the reinforcement learning traffic signal controllers can yield tangible improvements without the need for major investments in the road infrastructure to collect top-down imagery, as would have been required using the approach proposed in their 2016 paper.

4 Model

4.1 States, actions and rewards in our model

Our goal in this thesis is to demonstrate that a simple reinforcement learning agent using hardware that is already available to most modern cities can be used to control traffic and beat the performance of fixed traffic light phase timing commonly in place. To achieve this, we follow recent literature applying DDQN to traffic control, but use a more realistic state-representation that can be implemented with hardware readily available in most modern cities. We design a DDQN agent that makes use of only four state variables per lane: traffic lane occupancy, average speed in a traffic lane, cumulative

time queued vehicles have spent waiting at an intersection, and the amount of time a given lane has been given right of way, if it currently has right of way. All state variables could be calculated using information from a combination of induction loops and traffic cameras, or speed detectors. The action space is defined by the decision which traffic lights to turn green. To reward the agent, we use the change in total waiting time across all lanes in the network.

Our agent is trained using the Deep Double Q-Network algorithm introduced in section 2. We make use of a number of stability tweaks discussed in section 2 and further elaborated on in what follows. We test this approach on a 'small' and a 'large' intersection, and benchmark it with a pre-set fixed time traffic control policy as is commonly used in practice. To simulate the environment, we use an open source traffic simulator 'Simulation of Urban Mobility' (SUMO) developed by the Institute of Transportation Systems at the German Aerospace Center [Krajzewicz et al., 2012]. Information like the departure time, arrival, speed and the vehicle's route through the network are easily accessible in SUMO[2].

We now introduce some terminology to further describe our state and action space and the type of traffic problem we study. A transport network can be defined as a set of links that connect a set of junctions. For our model, we consider two single junctions where all links have both inflowing and outflowing traffic. The junctions we study differ in the number of lanes, and whether turning is allowed or not. In our 'small' junction, depicted in figure 1 no turning is allowed and every link has one inflowing and one outflowing lane. This junction has only two possible actions to choose from, whether to give right of way to vehicles in the horizontal or vertical links. In the case of our 'large' intersection, depicted in figure 2, turning is allowed and every link has three inflowing and three outflowing lanes. In this situation the agent can choose from four possible actions: whether to allow the cars in the horizontal links to move straight, the cars in the vertical links to move straight, the cars in the horizontal links to turn left, or the cars in the vertical links to turn left. State variables are calculated using an induction loop that is placed 100 meters from the junction center.

In traffic control terminology, the cycle time describes how long a full cycle of light changes takes at a given junction. The number of stages divides the cycle time into a set of intervals during which a particular combination of links has right of way. As our benchmark, we use the default cycle times and stage division provided by SUMO, which are industry-standard values corresponding to the junction size and lane setup. For the 'small' junction, the benchmark policy uses a cycle time of 86 seconds, divided into a green stage of 40 seconds for the horizontal and vertical links each, and a yellow stage of three seconds for the horizontal and vertical links each. For the 'large' junction, the benchmark policy has a cycle time of 98 seconds, divided into green stages of 33 seconds for the horizontal and vertical non-turning lanes each, and 10 seconds for the horizontal and vertical turning lanes each. Again, every green stage is followed by a three second yellow phase.

Our agent does not follow a fixed cycle time and can theoretically choose to continue giving right of way to a certain link indefinitely. It is further allowed to modify the order of stages as she/he deems optimal at any given point in time. Regarding the actions the agent chooses, the only constraint we put in place is that no signal phase is allowed to last less than 10 seconds, including the yellow phase. The agent chooses every 10 seconds whether to continue with the current light phases, or switch to another phase, and if so, which phase.

[2]It is worth noting that SUMO simulations are collision-free. In other words, SUMO deals with collisions by teleporting colliding vehicles to a different place in the network - something that one needs to take into account before implementing SUMO-tested models in the real world.

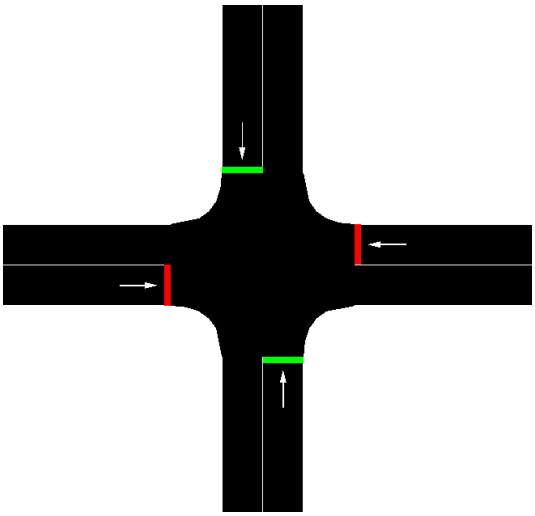

Figure 1: 'Small' intersection

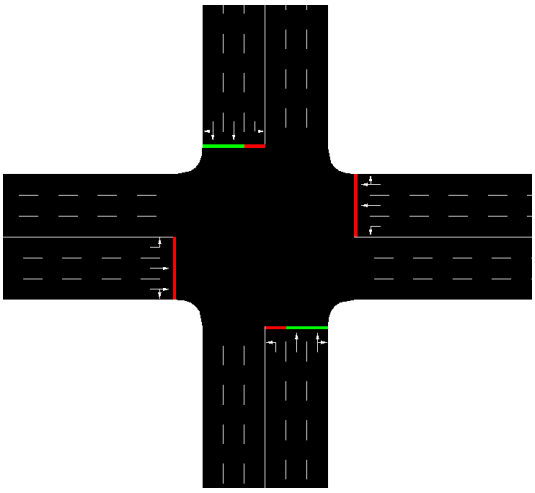

Figure 2: 'Large' intersection

4.2 DDQN in our model

Our motivation for choosing DDQN as the algorithm of choice is twofold – technical and practical. DDQN is a model-free and an off-policy algorithm, which allows for large continuous state and action spaces. In traffic control, state and action spaces are often indeed large, especially in a more realistic and complex traffic setting. DDQN is therefore easily scalable, which makes a DDQN model potentially suitable to control larger traffic networks. Further, in a traffic setting, subsequent observations are highly correlated, as (s_t, a_t) influences the probability of (s_{t+1}, a_{t+1}) [van Dijk, 2017]. This breaks the iid assumption underlying many machine learning techniques. Using a DDQN with experience replay can alleviate this by sampling the observations into mini-batches. Using mini-batches point of time does not happen based on the most recent observation of a transition, but based on a randomly sampled mini-batch of transitions drawn from the experience replay database (which we also call memory in the further text). Importantly for a model that one may test in the real world, DQN has been found to deliver reasonable stability in results [van Hasselt, 2010].

We provide pseudo-code for the DDQN algorithm we employ in algorithm 1. The main hyperparameters of the DDQN algorithm we tune over are the exploration rate ϵ, the discount rate γ, the specification of the reward function, the simulation step size δ, and the maximum allowed epsiode length. In terms of the exploration rate, we try both epsilon greedy exploration and exploration that

decays linearly over time, with different starting values. We tried two different reward functions. First, a 'negative' reward function that only provides punishment, and does so whenever the cumulative waiting time increases, and second, a 'balanced' reward function that provides positive and negative reward, depending on whether the cumulative waiting time goes up or down. We do not experiment with the simulation step size extensively, as we deem 10 seconds to be the minimum acceptable time for use in the real world between two red phases for a given lane. We do not choose a higher δ time in order to give the agent as much flexibility as possible; since keeping the status quo is always one possible action anyways.

We define our MDP to be episodic, and limit the length of a training episode to 10000 steps. However, we stop letting additional vehicles enter the system after 5000 steps, so any episode that does not finish after 10000 steps is deemed to have been unsuccessful and stopped.

Implementation To implement the algorithm and train it, we created a -Python interface, which allows our Python-based algorithm to interact with the SUMO-based environment[3]. As our q-network, we use a neural network with 4 state variables for every lane that has its own traffic phase i.e. 12 input variables for the 'small' junction and 40 input variables for the 'large' junction. We use two fully connected hidden layers with a rectified linear unit (ReLU) as activation function of the neural network. Figure 3 provides a visualization of the network for the 'simple' junction with only two actions to choose from, and hence two output nodes each providing the predicted Q-value for one action, based on which the DDQN agent takes her/his decisions. We do not do any pre-processing or feature engineering of the state variables, and feed them as they are to the input layer, recording good results with this approach.

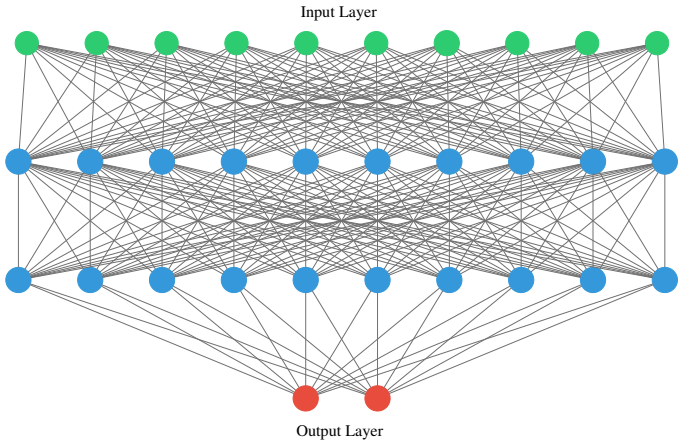

Figure 3: Q-network with 2 fully connected hidden layers

Fitting the q-network involves a large set of design decisions and hyperparameters that we now discuss briefly. To start with, we use the mean squared error as loss function to fit our q-network. We minimize this loss using backpropagation and the optimizer Adam, introduced by Kingma and Ba [2014]. Adam is a stochastic optimization algorithm that uses first-order gradients and adaptive step-sizes based on estimates of lower order moments. It is widely used in deep learning due to its good performance with neural networks. We try fitting the q-network at every step of the simulation, as well as less frequently. To fit the q-network we sample mini-batches from the experience memory, and then run a single epoch of backpropagation. The experience replay database is initially filled with transitions observed during 1000 completely random actions. As training progresses, these memories are replaced with transitions based on actions taken based on the highest predicted Q-value, and further random transitions depending on the exploration policy. The size of the mini-batch and the size of the experience memory are both hyperparameters that we tune over. At a minimum of every M steps, we copy the weights to the target q-network. Larger transfer frequencies can be expected to

[3]Please find our GitHub repository for this project under: `https://github.com/jrdmose/MS_Transports_RL`

increase stability, but decrease convergence speed. We try different values for this hyperparameter too. We further discuss the sensitivity of our model to hyperparameter changes in section 5.

Algorithm 1 DDQN

Initialize Q-networks Q^A and Q^B with random weights
Initialize state $s = s_0$
Initialize action $a \sim U(\mathcal{A})$
Initialize experience replay database $\mathcal{D}$
Fill experience replay with $|\mathcal{D}|$ transitions $(< s_t, a_t, r_t, s_{t+1} >)$
for $\mathrm{i} = |\mathcal{D}|; \mathrm{i}$ `< max episode length ;` $\mathrm{i}++$ **do**

    `Interact with the environment:`
    With probability ϵ : ▷ Select actions using ϵ -greedy
        $a_t \sim U(\mathcal{A})$
    Otherwise:
        $a_t = \underset{a}{\operatorname{argmax}}\, Q^A(s_t, a)$
    Take action a_t
    Store transition $(< s_t, a_t, r_t, s_{t+1} >)$

    `Update networks:`
    $(s_{t,m}, a_{t,m}, r_{t,m}, s_{t+1,m}) \sim U(\mathcal{D})$ ▷ Sample mini-batch of transitions from $\mathcal{D}$
    $a_{t+1,m} = \underset{a}{\operatorname{argmax}} Q^A(s_{t+1,m}, a)$
    update Q^A minimising $\left(Q_t^A(s_{t,m}, a_{t,m}) - r_{t,m} + \gamma\left[Q_t^B(s_{t+1,m}, a_{t+1,m})\right]\right)^2$ on batch.
    Every M steps:
        Set $Q^B = Q^A$ ▷ Copy value network weights to target network

end for

4.3 Training and testing our model

To train and test our algorithm we program a number of different demand scenarios for both the 'simple' and the 'large' junction. SUMO facilitates probabilistic traffic demand, meaning that to define demand we choose the probabilities that cars enter a junction in a certain link at every second. We test scenarios where demand is fixed, and where it changes over the period of an episode. Compared to the fixed benchmark policy, the DDQN algorithm should have a particular advantage in situations when traffic demand fluctuates.

We summarise the probabilities we chose for our demand scenarios in table 1. As a metric to evaluate the performance of the DDQN algorithm, we use the average vehicle delay during an episode. This number increases the longer a vehicle spends waiting at an intersection, and is therefore closely related to our reward function, but more meaningful from a traffic management perspective. We also base our hyperparameter tuning on this metric. For a real world implementation, the metric could easily be adapted to take into account other variables of interest such as CO2 emissions or average speed. We note that the average vehicle delay favors the masses over the individual, which one may choose to limit to some degree in real world implementations.

Table 1: Traffic demand proportions by link

	N	E	S	W
N	0	$1/80$	$1/40$	$1/80$
E	$1/20$	0	$1/20$	$1/15$
S	$1/40$	$1/80$	0	$1/80$
W	$1/20$	$1/15$	$1/40$	0

We train the DDQN agent over a period of 100 episodes, and evaluate it over 5 episodes. Since the traffic demand is probabilistic the environment provides slightly different demand profiles in every episode. We further change the random seed in every training run to ensure reproducibility of our results. Apart from the average vehicle delay as our evaluation metric, we also study the convergence of the mean squared error of the q-network. Only when the mean squared error decreases over a training run can we expect the DDQN agent to have learned a reasonable policy. Lastly, we use the graphic user interface of SUMO to better understand the actual decisions the DDQN network takes.

5 Results

Our results[4] show that, on average, the DDQN achieves 8% lower average vehicle delays than the benchmark policy on the 'small' junction and 44% lower average vehicle delay on the 'large' junction. The DDQN also achieves 13 and 42% lower CO_2 emissions. We achieve this result consistently for all trained agents.

Table 2: Performance of DDQN agent compared to the benchmark

	Delay [s]			**CO2 Emissions [mg]**		
	RL	fixed policy	% difference	RL	fixed policy	% difference
'Small' junction	66	71	-8%	1.4e5	1.6e5	-13%
'Large' junction	202	291	-44%	4.3e5	6.1e5	-42%

To achieve these results, we ran an extensive gridsearch to find the best hyperparameters for our model. We then re-ran the best parameter combination eight times using differing random seeds to ensure reproducibility. We present the training and evaluation results for our simulations of rush hour traffic demand in figures 4 to 6. Figure 4 and 5 show the performance of the agent during training on the 'small' and 'large' junction under rush hour demand as indicated in table 1. We plot the evolution of the average vehicle delay in seconds achieved across the eight runs over the 200 training episodes. The mean performance of the DDQN agent is plotted in dark blue vs the benchmark in orange. The standard deviation over the eight runs are indicated by shades of these colors.

Figure 4 shows that due to the randomly initialized weights of the q-network, the DDQN agent at first takes completely random actions that yield very poor traffic control performance, leading to nearly twice the average vehicle delay of the benchmark policy for the 'small' junction and 20% higher vehicle delay for the 'large' junction. In the case of the 'small' junction, the DDQN first manages to beat the benchmark policy after 75 training episodes. From then on, the agent's performance oscillates around the level of the benchmark, exhibiting significantly more instability in average vehicle delay than the benchmark. In the case of the 'large' junction, the advantage of the DDQN agent becomes apparent much more quickly, almost immediately after training begins (see figure 5). This may be due to the fact that in the 'larger' junction the potential gains from an adaptive traffic control are larger under rush hour demand, as more choices are to be made that can differentiate the performance of the DDQN from a fixed-policy benchmark. The DDQN agent then outperforms the fixed benchmark consistently for all training episodes. However, we note that it exhibits relatively large variance, which points to potential instability issues.

[4]We provide two videos of the performance of our trained agent compared to the benchmark under `https://github.com/jrdmose/MS_Transports_RL/tree/master/Videos`. One video shows the performance for the 'small' junction and one video shows the performance for the 'large' junction.

Figure 4: Training of agent on 'small' cross

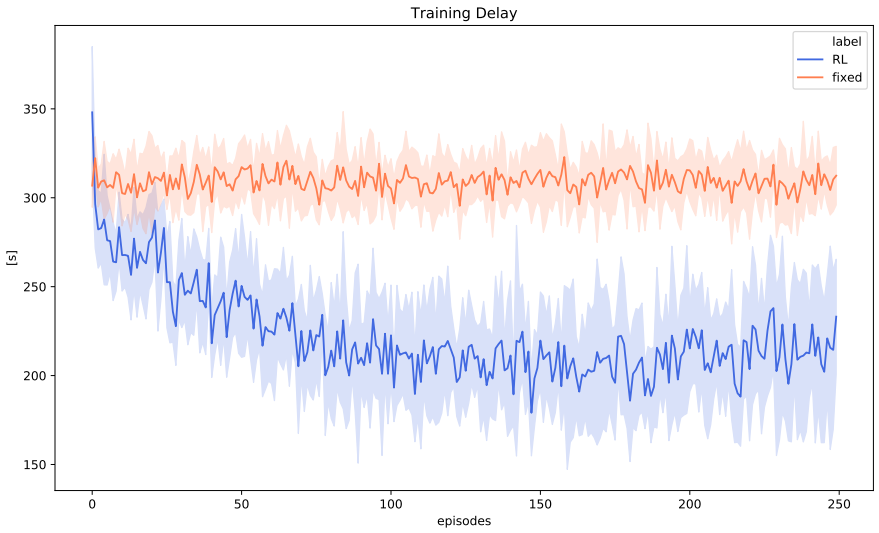

Figure 5: Training of agent on 'large' cross

We also test whether the DDQN agent can manage traffic, once trained, without exploration. For this purpose, we simulate each trained agent over five episodes and show the average vehicle delay during an episode in figures 6. On the upper panel, we plot the demand profile during for what we call the 'rush hour' scenario. In this scenario, we increase the baseline level of cars entering the junctions by a factor of 2. The demand has some variance due to the probabilisitc nature of the SUMO simulations. In the two lower panels, we show the average vehicle of the DDQN agents during the evaluation episodes. The horizontal axis displays the simulation interval in seconds. Additional cars enter the

links for nearly 1 hour (3500 seconds), after which the simulation stops. As rush hour begins, both the DDQN as well as the benchmark show increasing average vehicle delay. Soon after the maximum demand is reached, the DDQN clearly outperforms the benchmark both for the 'small' and the 'large' junctions. Again, we note that the difference to the benchmark is significantly larger for the more complex junction.

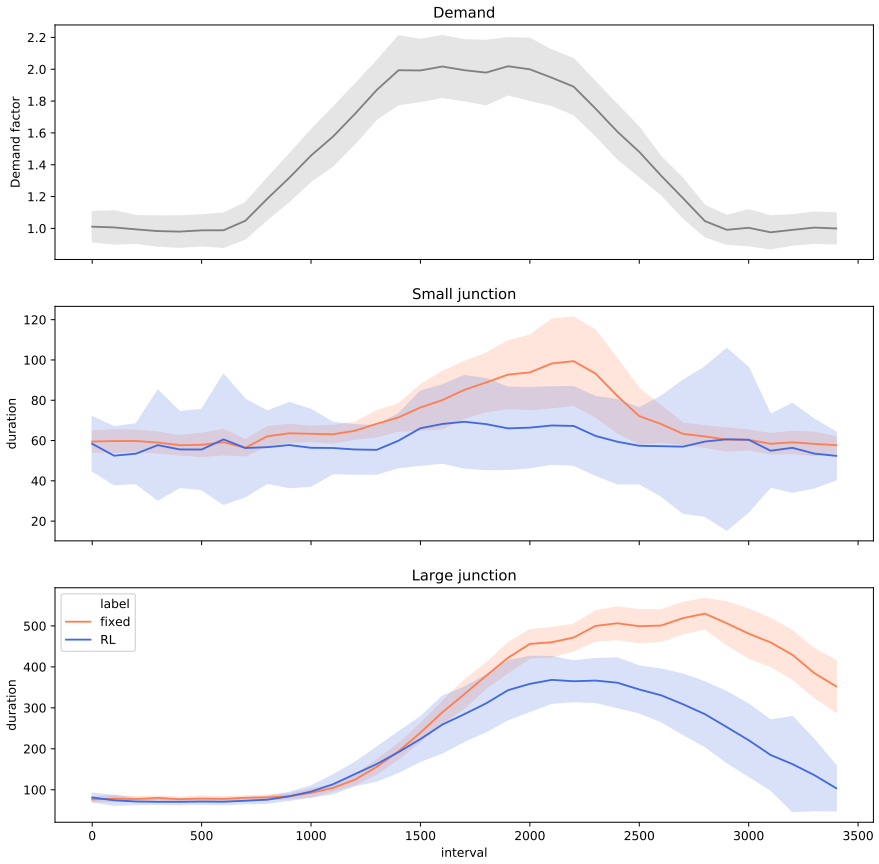

Figure 6: Average performance of trained agents in evaluation over a 1h episode

To find the hyperparameter combination that achieves these results we ran an extensive hypter-parameter search over the parameter space of the DDQN algorithm as defined in section 4. We summarised the parameters we searched over in table 3. The most crucial parameters proved to be the exploration parameter epsilon, the specification of the exploration policy, the specification of the reward function, and the train frequency. Naturally, this assumes that the remaining parameters have been put to reasonable values, as all parameters have the potential to render the algorithm non-functioning.

As would be expected, we find that generally, higher epsilon values decrease stability, whereas lower epsilon values increase stability but lead to slower convergence. When trained using stable traffic demand, epsilon greedy policies with fixed epsilon values perform well. However, when demand is set to be unstable during training such as during our rush hour scenario, a policy of linear decay starting from high exploration rates and gradually decreasing it proved to work well. Among the two

different reward functions we try, the 'balanced' reward function yielded lower average vehicle delay times than the 'negative' reward function in the 'small' and 'large' network. However, the 'balanced' reward is much more unstable in evaluation results. In line with what we expect the TD-loss decreases continuously during training. Depending on the choice of ϵ, the length of training, and whether the exploration policy decreases exploration over time, we have experienced divergence of the TD-loss. Furthermore, we find that results are not fully stable, as the range between the best performing agent and the worst performing agent is relatively large.

Table 3: DDQN parameter search grid

Parameter[5]	Description	Range tested	Best Parameter
eps	Exploration factor	10-50	30
gamma	Discount factor	0.95-0.99	0.99
train_freq	Frequency to fit q-net	1-5	1
target_update_freq	Frequency to copy θ^V to θ^T	5000-40000	5000
max_size	Max_size of replay memory	10000-100000	10000
policy	Exploration policy	Eps-greedy, Linear Decay	Linear Decay
max_ep_length	Stop episode after x steps	1000	1000
delta_time	Simulation step size	10	10
reward	Reward policy	Balanced, Negative	Balanced
num_burn_in	Size of initial memory	1000	1000
batch_size	Batch size to train q-net	1	1
optimizer	Optimization of q-net	1	1
loss_function	Objective function for q-net	MSE	MSE

6 Discussion

In this thesis we test the applicability of reinforcement learning techniques to traffic control. Traditionally, traffic control problems have been approached using DP-type solutions that optimise a value function trading-off current rewards with expected future rewards given specific policies. However, the dynamics and probability distributions describing the evolution of complex systems cannot easily be modelled deterministically. To overcome this, recent literature suggests using function approximations that learn how the environment reacts to certain policies chosen by the agent. The most promising such reinforcement learning techniques use deep neural networks for this.

The promise of reinforcement learning techniques in traffic control is that they allow finding optima for problems that are analytically intractable, where dynamic programming approaches thus far have yielded only sub-optimal results. As discussed in section 2, reinforcement learning offers several techniques that are model-free, hence do not require a full model of the environment. They can be applied to complex traffic situations without the need for simplifying assumptions. One need not understand the nature of traffic flow fully, but can still find the best way to control it. Reinforcement learning techniques also lend themselves to the use of function approximations such as neural networks and optimization methods such as stochastic gradient descent that allow mapping of very large state spaces to a single reward metric, taking into account many different variables that determine the traffic flow. Function approximators such as neural networks further facilitate variable selection, which filter out the most important variables to solve the problem at hand.

To test the applicability of reinforcement learning in traffic control, we implement DDQN, one of the most successful algorithms in reinforcement learning that has recently gained popularity in traffic control research. We interface the traffic simulator SUMO with a DDQN algorithm implemented in Python, and test its performance in controlling traffic lights on a 'small' and a 'large' traffic junction. We find that the DDQN algorithm achieves significantly lower average vehicle travel times than a fixed traffic light phase program when traffic demand fluctuates, as it is capable of reacting to real-time traffic situations. Furthermore, we find that the performance difference is particularly large for the more complex junction, which suggests that the approach is particularly well suited to handle traffic control of more complex intersections.

To arrive at these results, we searched over a grid of parameters, then replicated the results of the best performing parameter set eight times, and then evaluated each of the eight trained agents

over five episodes without additional exploration. We consistently beat the benchmark policy, but do experience volatility in performance of the DDQN agent, which we attribute to the nonlinear function approximation. Furthermore, we note that during the parameter grid search large areas of the parameter space produce divergence and instability. We expect that in the extension of this model to a more complex problem, these issues would need to be carefully navigated, and thus propose to put in place stopping conditions during training that halt training once good performance has been achieved. Another tweak to consider when adapting this model to more complex problems would be to use prioritized experience replay, which we expect to be particularly relevant when traffic demand is highly seasonal or exhibits structural breaks. Lastly, we note that the set of hyperparameters that maximize performance are context-specific, and overtly extensive interpretation therefore of limited value.

Reinforcement learning is embedded in an active research community, where new algorithms and optimization methods are constantly being developed. Based on the positive results we record using a state-of-the art reinforcement learning algorithm for our experiments, we expect further advances in reinforcement learning to carry large benefits for traffic control. Nevertheless, reinforcement learning approaches also have their weaknesses compared to DP approaches. First of all, reinforcement learning algorithms yield non-linear solutions that may not be tolerated in a real world traffic control situation. They often cannot be rationalised, and thus particular care needs to be taken to put in place constraints that restrict the behaviour of the reinforcement learning agent to something that is tolerable for humans. Moreover, reinforcement learning algorithms are also known to be unstable, since their solutions in most cases involve some randomness, which in turn can cause unexpected reactions. Additionally, in many cases it is difficult to say whether a reinforcement learning algorithm has arrived at an optimal solution or not, as many local optima exist and, when in a model-free environment, the global optimum is unknown.

As of yet the field still lacks clear direction. It is still unknown what works best in terms of defining the state, specifying the reward, and training the algorithm. A further open question concerns the scaling of reinforcement learning approaches to large transport networks. Specifically, it is unclear whether the DDQN algorithm we study would perform well in the coordination of multiple intersections at once. It is possible that a decentralized approach is inferior in dealing with pass-through effects of traffic between intersections, and that an approach that creates a team of agents coordinated by a supervisor yields better performance, something that researchers in the field are already actively studying. Recent advances in applying reinforcement learning to multiplayer games such as Jaderberg et al. [2019] could provide useful guidance to approach this problem in the traffic control setting.

References

Dimitri P Bertsekas. *Dynamic programming and optimal control*. Athena scientific Belmont, MA, 2005.

Peter Dayan and Yael Niv. Reinforcement learning: the good, the bad and the ugly. *Current opinion in neurobiology*, 18(2):185–196, 2008.

Christina Diakaki, M Papageorgiou, and T McLean. Application and evaluation of the integrated traffic-responsive urban corridor control strategy in-tuc in glasgow. In *Proceedings of the 78th Annual Meeting of the Transportation Research Board*, 1999.

Samah El-Tantawy, Baher Abdulhai, and Hossam Abdelgawad. Design of reinforcement learning parameters for seamless application of adaptive traffic signal control. *Journal of Intelligent Transportation Systems*, 18(3):227–245, 2014.

Eyal Even-Dar and Yishay Mansour. Learning rates for q-learning. *Journal of Machine Learning Research*, 5(Dec):1–25, 2003.

Nathan H Gartner. *OPAC: A demand-responsive strategy for traffic signal control*. Number 906. 1983.

Wade Genders and Saiedeh Razavi. Using a deep reinforcement learning agent for traffic signal control. *arXiv preprint arXiv:1611.01142*, 2016.

Wade Genders and Saiedeh Razavi. Evaluating reinforcement learning state representations for adaptive traffic signal control. *Procedia computer science*, 130:26–33, 2018.

Randolph W Hall. *Handbook of Transportation Science*, chapter 8, pages 243–277. International Series in Operations Research & Management Science. Springer US, 2006. ISBN 9780306480584.

K Larry Head, Pitu B Mirchandani, and Dennis Sheppard. *Hierarchical framework for real-time traffic control*. Number 1360. 1992.

PB Hunt, DI Robertson, RD Bretherton, RI Winton, and A SCOOT. A traffic responsive method of coordinating signals. *Transport and Road Research Laboratory, Crowthorne, Berkshire, England, Report No. LR1014*, 1981.

Max Jaderberg, Wojciech M. Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castañeda, Charles Beattie, Neil C. Rabinowitz, Ari S. Morcos, Avraham Ruderman, Nicolas Sonnerat, Tim Green, Louise Deason, Joel Z. Leibo, David Silver, Demis Hassabis, Koray Kavukcuoglu, and Thore Graepel. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019. ISSN 0036-8075. doi: 10.1126/science.aau6249. URL `https://science.sciencemag.org/content/364/6443/859`.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012. URL `http://elib.dlr.de/80483/`.

Xiaoyuan Liang, Xunsheng Du, Guiling Wang, and Zhu Han. Deep reinforcement learning for traffic light control in vehicular networks. *arXiv preprint arXiv:1803.11115*, 2018.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Anurag Pande and Brian Wolshon. Traffic engineering handbook. *The Institute of Transportation Engineers. Wiley Online Library*, 2016.

Suvrajeet Sen and K. Larry Head. Controlled optimization of phases at an intersection. *Transportation Science*, 31(1):5–17, 1997. doi: 10.1287/trsc.31.1.5. URL https://doi.org/10.1287/trsc.31.1.5.

David Silver. UCL Course on RL: Advanced Topics (COMPM050/COMPGI13), 2015. URL http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

AG Sims and KW Dobinson. Scat the sydney coordinated adaptive traffic system: philosophy and benefits. In *International Symposium on Traffic Control Systems, 1979, Berkeley, California, USA*, volume 2, 1979.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

Elise van der Pol. Deep reinforcement learning for coordination in traffic light control. Master's thesis, University of Amsterdam, 2016.

Jaimy van Dijk. Recurrent neural networks for reinforcement learning: an investigation of relevant design choices. Master's thesis, University of Amsterdam, 2017. URL https://esc.fnwi.uva.nl/thesis/centraal/files/f499544468.pdf.

Hado van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Christopher John Cornish Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. *King's College, Cambridge*, 1989.