

---

# **A reinforcement learning based traffic signal control algorithm in a connected vehicle environment**

**Kaidi Yang, IVT, ETH Zurich**

**Isabelle Tan, Department of Mathematics, ETH Zurich**

**Monica Menendez, IVT, ETH Zurich**

**Conference paper STRC 2017**

**STRC**

**17<sup>th</sup> Swiss Transport Research Conference**

Monte Verità / Ascona, May 17-19 2017

## Title of paper

Kaidi Yang  
IVT, ETH Zurich  
Zurich

Isabelle Tan  
Department of Mathematics,  
ETH Zurich  
Zurich

Monica Menendez  
IVT, ETH Zurich  
Zurich

Phone: 044 633 32 46

email:

kaidi.yang@ivt.baug.ethz.ch

email:

isabelle.w.tan@gmail.com

Phone: 044 633 66 95

email:

monica.menendez@ivt.baug.ethz.ch

May 2017

## Abstract

The emerging vehicle technologies, i.e. connected vehicle technology and autonomous driving technology, can be beneficial for traffic control and operations. They not only serve as new source of information, but also enable us to control and modify the trajectory of vehicles. Reinforcement learning is widely used to design intelligent control algorithms in various disciplines. This paper provides preliminary results on how the reinforcement learning methods perform in a connected vehicle environment.

## Keywords

reinforcement learning, traffic signal control, connected vehicle technology, automated vehicles

## 1. Introduction

Traffic signal control is an essential element in urban traffic control and management systems. Signal control strategies can be classified into fixed-time, actuated and adaptive signal control strategies. Fixed time signal control strategies, such as TRANSYT (Roberson, 1969), use historical data, and provides fixed signal timings despite the real traffic scenarios. Actuate or adaptive signal control strategies generally use real-time information, such as vehicle actuation, loop detector data or video camera data. These two types are more responsive to real traffic situations. However, the traditional data source for these two types of signal control strategies are roadside detectors, which are usually installed at a fixed location and cannot provide detailed information about the movement of individual vehicles. Thanks to the recent development in connected vehicle technology (i.e. vehicles that can communicate with each other and infrastructure to provide information on speed, location, etc.), it is now possible to track and control the movement of vehicles.

The benefit of connected vehicles is two-fold. On the one hand, they provide real-time information on the speed, location, headway, etc., which can be used to develop better signal control strategies. On the other hand, wireless communication systems and automated driving can help advice drivers or control vehicles, providing a more flexible design of signal control strategies. Inspired by such benefits, connected vehicles has been attracting increasing attention in traffic signal control.

Existing literatures can be classified as two categories. The first category is optimization-based algorithms. These type of literatures build optimization models of traffic dynamics, and aim to optimize the total vehicle delay, throughput, etc. Many works formulate the traffic signal control problem into an integer programming problems (He et al., 2012; He et al., 2014; Feng et al., 2015). In two previous papers in our group, we designed a signal control algorithm for an isolated intersection. Guler et al. (2014) proposed a signal control strategy based on information provided by connected vehicles present in a traffic stream, and evaluated the benefits of this technology for different penetration rates. Yang et al. (2016) extends this work by considering automated vehicles and integrating trajectory planning of these vehicles to the proposed scheme. This extension further improves the performance indices by reducing both the delay and the number of stops. However, these category of traffic control models can be computationally expensive, and sensitive to modelling errors.

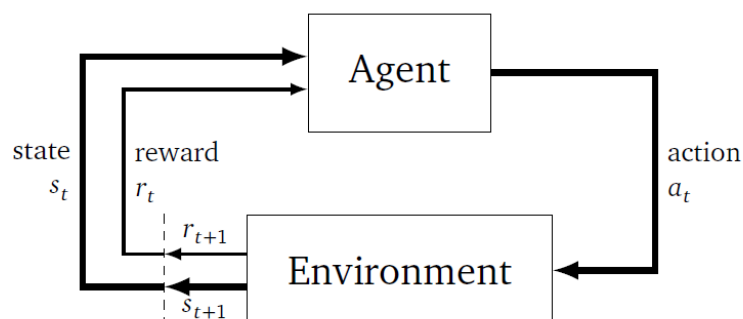
The second category of traffic signal control strategies is based on machine learning methods. The most popular method in this category is reinforcement learning, which aims to optimize the control policy through a trial and error interaction with the environment. The general idea is to approximate the traffic dynamics, i.e. the relation between the total delay and the signal

timings using learning methods. Abdulhai et al. (2003) and Wiering (2000) proposed reinforcement learning for isolated intersections and urban network, respectively, using Q-matrix learning. The algorithm in Wiering (2000) has been extended by Kuyer et al. (2008) and El-Tantawy et al. (2013) to consider the interaction between intersections. Gilmore et al. (1993) proposed a neural network based adaptive traffic signal light controller without hidden layer. Pierre-Luc Grégoire et al. (2007) proposed a policy descent based reinforcement learning method to optimize traffic signal in a simple intersection. The reinforcement learning based methods is usually more responsive to real traffic situation, as it updates the approximation model with time. However, the aforementioned works this category did not consider the information of connected and automated vehicles. This paper aims to provide some results on how the reinforcement learning method performs using such information.

The organization of this paper is as follows. Section 2 gives a brief introduction to the reinforcement learning. Section 3 introduces the training and simulation environment. Section 4 provides preliminary results. Section 5 concludes the paper.

## 2. Background on Reinforcement Learning

Figure 1 Schematic figure for reinforcement learning



Reinforcement Learning is a Machine Learning paradigm, by which a controller's policy can be optimized through trial-and-error interactions with an environment. A schematic illustration is shown in Figure 1. By using a predefined reward function from the environment as feedback, the controller can learn which policies are effective and ideally converge to the most optimal policy. It has yielded interesting results, among which are outperforming humans in board games (Silver et al., 2016), computer games (Mnih et al., 2015) and RC helicopter control (Ng et al., 2006).

The agent-environment interaction can be more formally described using the notation from Sutton and Barto (1998) which is as follows. At every time step  $t$ , the agent obtains a (potentially partial) representation of the state  $s_t \in S$  of the environment. Given this state  $s_t$ , the agent will decide what the next action  $a_t \in A$  should be and forwards  $a_t$  to the environment. The environment then executes this action at leading to a transition to a new state  $s_{t+1} \in S$ , for which a reward  $r_{t+1}$  is given to the agent. The agent decides which action to take based on its policy  $\pi_t$ , where  $\pi_t(s, a)$  denotes the probability of taking action  $a$  when in state  $s$ . The ultimate goal of the reinforcement learning problem is to find the optimal policy  $\pi^*$  which is defined as the policy that will yield the highest expected return  $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ , which is defined as the cumulative future reward. Here,  $\gamma \in [0,1]$  is a discounting factor, which represents how myopic the agent is. If the agent is very myopic and only cares about the current reward, then  $\gamma = 0$ ; otherwise if the agent takes fully consideration of the future  $\gamma = 1$ .

The most common used reinforcement learning scheme is Q-learning. Q-learning is a value-based learning algorithm. It attempts to predict the outcome of the value function  $V^\pi(s)$ , which maps a state  $s$  and a policy  $\pi$  to a value representing how good it is to use this policy

and state. One function that is eligible as Value Function could be the expected return, such that

$$V^\pi(s) = E_\pi(R_t | s_t = s) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right)$$

We define the Q-value, also known as Action Value,  $Q^\pi(s, a)$ , as the value of taking action  $a$  in state  $s$ , while being subject to policy  $\pi$ , such that

$$Q^\pi = E_\pi(R_t | s_t = s, a_t = a) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right)$$

Q learning algorithm aims to approximate this Action Value function with some mapping from state action pair to the expected Value. Then the optimal policy greedily follow the actions that gives the best Q-Values, assuming that the obtained Action Value function converges to the Action Value function of the optimal policy.

The algorithm starts with a random initialisation of the Action-Value function and improves it iteratively, by executing a batch of state-action pairs and comparing the predicted expected return against the actual received return. It then corrects the parameters of the mapping such that it fits the actual obtained result better by for example using a Gradient Descent method. The mapping used for the approximation of the Action-Value function can be anything, such as a state-action-reward table, or an artificial neural network (ANN) in the case of the Neural Fitted Q-learning (NFQ) algorithm.

We can perform the Action-Value function approximation using only short term future reward information, a principle known as Temporal Difference Learning, because the Value function satisfies the Bellman equation, shown in Equation 3. This is a common practice used in dynamic programming, also called bootstrapping.

$$V^\pi(s) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right) = E_\pi(r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s)$$

Q-learning is an off-policy Temporal Difference (TD) control algorithm, meaning that it can approximate the Action-Value function of a policy while following a different policy. This allows the method to explore while still approximating the Action-Value function for the policy considered most optimal, i.e. the greedy policy. It does so by using the most optimistic Action-Value as feedback for the parameter adjustment, instead of using the expectation of the Action-Value assuming policy  $\pi$ , represented by a sum of state-action values weighted by the probabilities of taking those actions under policy  $\pi$ . This can be seen in the following update rule,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \arg \max_a (Q(s_{t+1}, a_t)) - Q(s_t, a_t))$$

The variable  $\alpha$  represents the learning rate, so it controls the step size of the parameter adjustment that will be taken. We can see that by using the update rule as defined in Equation 4, the Action-Value function that the mapping approximates will be the Action-Value function corresponding to the optimal policy, regardless of what policy is actually followed during the training. To control the policy that is used during the training it is common to use an exploration parameter  $\epsilon$  with  $0 \leq \epsilon \leq 1$  which defines the probability with which one of the suboptimal actions is taken.

We use two types of mappings to approximate the Action-Value function, namely an artificial neural network and a state-action-reward table.

## 2.1. Q matrix

Q matrix is a matrix, where each row represents each state, and the column value represents each action. By discretizing all infinite state dimensions into a finite set of bins we obtain a finite set of  $n_s$  possible states. This is a straightforward representation, as one can read the Q value directly from this matrix. Q matrix learning is a straight forward method. However, with the number of states increases, it is hard to represent the Q matrix.

State	$a_1$	$a_2$	...	$a_n$
$s_1$	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...	$Q(s_1, a_n)$
$s_2$	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...	$Q(s_2, a_n)$
...	...	...	...	...
$s_m$	$Q(s_m, a_1)$	$Q(s_m, a_2)$	...	$Q(s_m, a_n)$

Table 1. The lay-out of an Action-Value Table

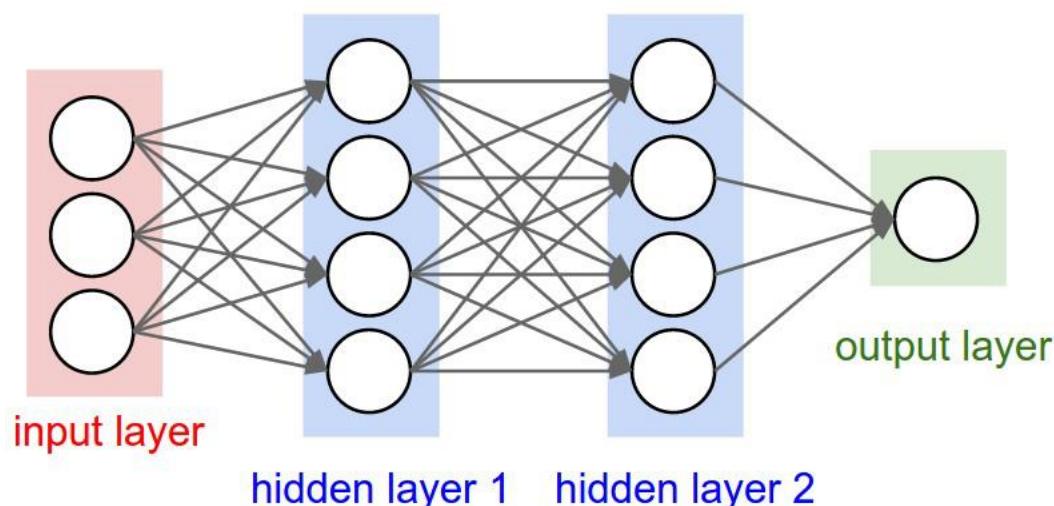
## 2.2. Q network

A more powerful way to approximate Q value is through artificial neural network (ANN). Figure 2 shows an example of the neural network. The input layer (in red shade) for the reinforcement learning consists of the actions and states; and the output layer (in green shade) represents the reward. The intermediate layers are called hidden layers. We calculate the values of a node in a layer as the output of a nonlinear activation function, whose input is the linear combination of all the nodes in the previous layer. Therefore each layer consists certain nonlinearity. With the depth of the network increases (i.e. more layers), we can represent more nonlinearity, and theoretically more complex functions. Commonly used activation functions include ReLU, sigmoid, etc. However, it is hard to see which topology and

activation function work the best. In this preliminary test, we tested the ANN with no hidden layer, one hidden layer and two hidden layers. The activation function is sigmoid, i.e.

$$f(x) = \frac{1}{1 + e^x}$$

Figure 2 A example of neural network



Source: <http://cs231n.github.io/neural-networks-1/>

### 2.3. Application to traffic signal control

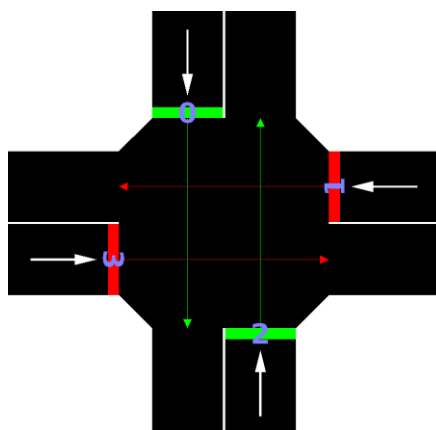
We compute the vehicle delay as the different between the actual arrival time and the virtual arrival time. In order to punish large delays, we use the negative sum of the squared delays of all cars currently approaching or waiting for the intersection, instead of total delay, as the reward value. Since it is negative, it effectively works as a punishment value instead of a reward value. We used the sum of squared delays to favor many small delays over few large delays, thereby encouraging the controller to be fair to all road-users. As the states that we pass to the controller, we use the sum of squared delays on the North and South approaches, the sum of the squared delays of the East and West approaches, the phase that is currently set on the traffic lights and the duration for how long this phase has been set. The two actions of the controller are simply defined by setting green to the North and South approach or setting green to the East and West approach.



### 3. Simulation Settings

In this paper, a simple intersection of two bi-directional streets and two phases is considered. Each direction has one lane. We call the left- and right- going direction as E and W respectively, and the upward and downward direction as S and N, respectively. This simple intersection is used as an initial building block. The traffic signal is assumed to be able to communicate with vehicles to receive information of the real-time speed and location. To ensure that the interaction is still safe we built-in 4 seconds of yellow between two different phases during which the lights cannot be switched. The red time and the green time for each phase will be decided using the reinforcement learning based algorithm. We use the Simulation of Urban Mobility (SUMO, Krajzewicz et al., 2012) as the microscopic traffic simulator, which can be called from Python scripts using the interface TraCi. We use the tools for reinforcement learning provided in the machine learning package PyBrain (Schaul et al., 2010) to train controllers, which could control the current traffic lights through the TraCi interface.

Figure 3 Intersection layout



Source: [http://sumo.dlr.de/wiki/Simulation/Traffic\\_Lights](http://sumo.dlr.de/wiki/Simulation/Traffic_Lights)

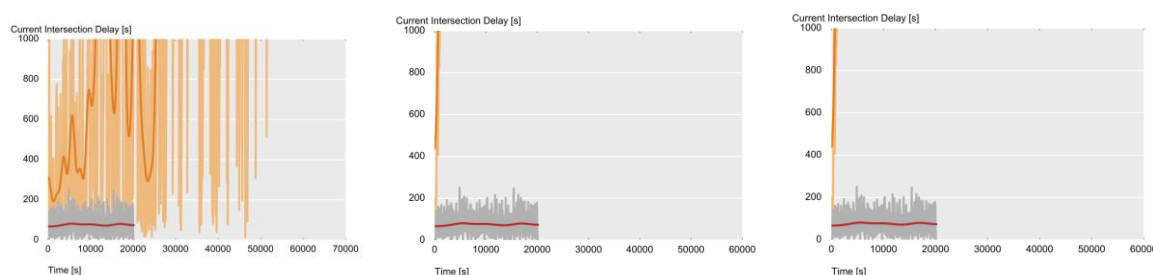
## 4. Results

### 4.1 Constant Demand

#### 4.1.1 Q-network approach

For the first simulations we have used a constant demand of 120 vehicles per hour on the E and W approaches and 60 vehicles per hour on the N and S approaches. This is relatively low, but we have to keep in mind that initially the controllers are no smarter than random controllers. In Figure 4 we can see the sum of the delay of all cars waiting for or approaching the intersection versus time for different network lay-outs. The result for our learning controller is plotted in orange and the result of the traditional naive fixed time controller is shown in grey and red. The thick orange and red lines are smoothed fits of the real data, which is shown in the background in light-orange and grey. We observe that the controller with the neural network Action-Value function approximation does not perform well when using a network with one or two hidden layers. When using a linear combination as Action-Value function approximation the results stay more reasonable at first, but then diverge as well.

Figure 4 The sum of delay of the cars currently approaching or waiting for the intersection versus time for different network lay-outs.



(a) No hidden layers.

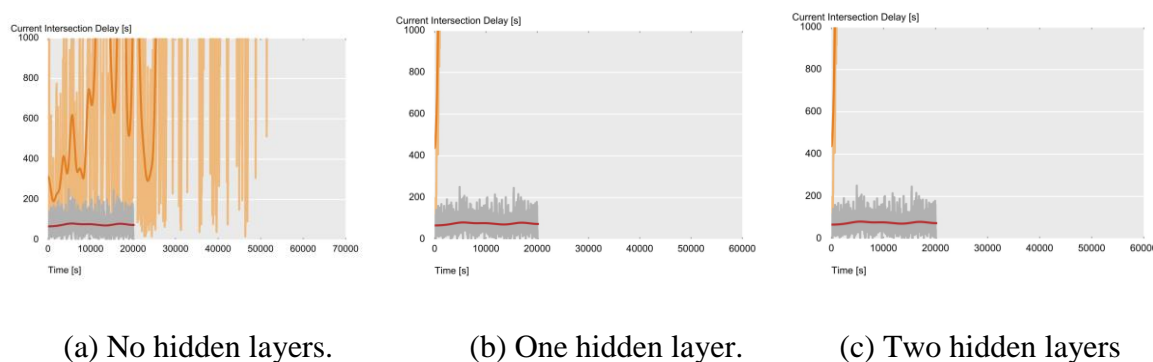
(b) One hidden layer.

(c) Two hidden layers

We also observe that in the beginning the controller was switching between the phases with a very high frequency. This is inefficient because the frequent phase transition does not encourage platooning and leads to a larger proportion of lost time. Of course we can't expect the controller to perform well right away, because it has barely had time to learn. To guide the controller to policies that we expected to be desired we introduced some constraints on the phase duration. We set the minimum phase duration to 10 seconds, and the maximum phase duration to 60 seconds. Furthermore an action could only be taken every 10 seconds, which is also more practical in real-life applications because it provides the system with plenty of time to perform

computations. The results after this constraint are shown in Figure 5. We can see that these constraints do keep the total intersection delay bounded to reasonable values, but that it still does not improve over the fixed time controller.

Figure 5: The sum of delay of the cars currently approaching or waiting for the intersection versus time for different network lay-outs, using the phase duration constraints.



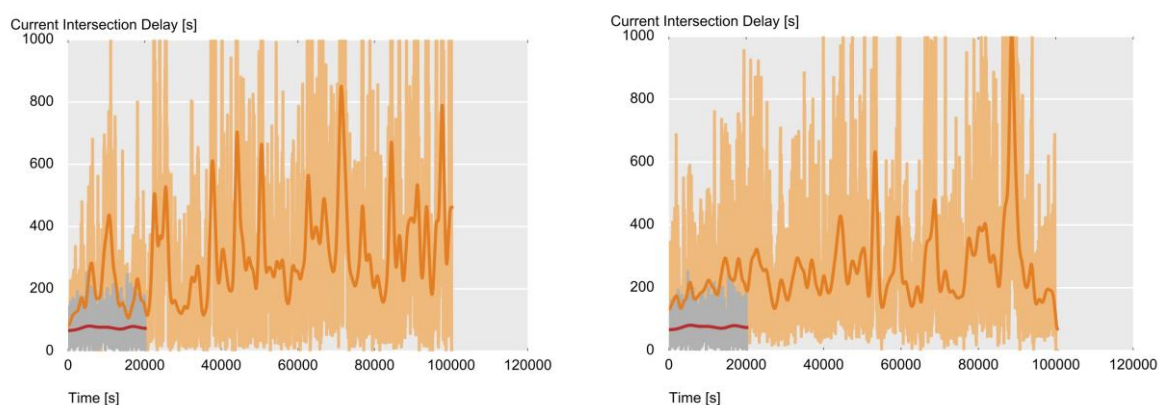
There is a general perception that the Q-learning algorithms with neural network function approximations are very hard to train requiring a lot of specialized knowledge and data (Grégoire et al., 2007). On the other hand, the highly nonlinear system may require more nonlinearity in the neural network, i.e. deeper, more input variables and with a larger variety of activation functions. This is taken as future research.

#### 4.1.2 Q matrix approach

Because three of the 4 dimensions in our state space have an infinite domain, we first need to discretize these dimensions into a finite set of bins. After this discretization we have a finite set of possible states which we can concatenate into a list. We can then combine the finite set of states and actions into a so-called Action-Value Table, as described in Section 3.1.2. For the state discretization we initially used five bins for the delays bounded by  $[0, 200, 400, 600, 800, 1000]$  and six bins for the phase duration bounded by  $[0, 12, 24, 36, 48, 60, 100]$ . The six phase duration bins were an error and should have been seven bounded by  $[0, 10, 20, 30, 40, 50, 60, 100]$ , since that is a one to one mapping for all states that it can take when using the phase duration constraint. We fixed this later when doing simulations for the non-constant demand pattern. The total intersection delay of the controller with the Action-Value Table are shown in Figure 6a and 6b, without and with the phase duration constraints respectively. The performance of the fixed time controller is again plotted for reference in grey and red. We can see this controller

also remains within reasonable bounds without the phase duration constraints, which the Action-Value Network controller did not. The performance of the controller however is still worse than the fixed time controller.

Figure 6: The sum of delay of the cars currently approaching or waiting for the intersection versus time without constraints (top) and with constraints (bottom).



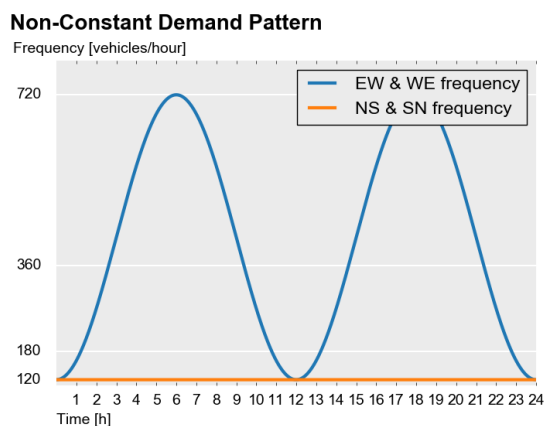
(a) Without phase duration constraints.

(b) With phase duration constraints

## 4.2. Varying demand

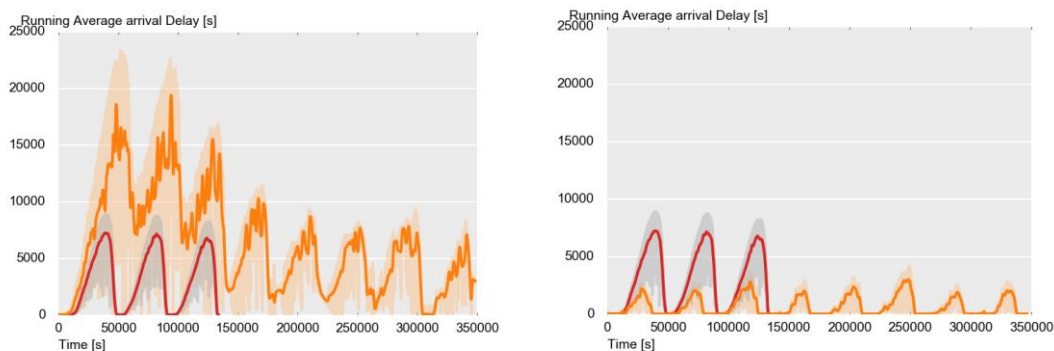
For constant demand patterns there are well-known methods to find near-optimal fixed time control cycles. Therefore the main motivation for adaptive controllers are intersections with dynamic demand patterns, since for those demand patterns the performance of a fixed time controller really breaks down. Because of this we decided to evaluate the performances on non-constant demand patterns next. We specifically chose a demand pattern in Figure 7.

Figure 7 Demand Patter



In Figure 8 we can see the running average of the arrival delay, taken over the last 10 vehicles that have arrived at their destination. We can see that in the constrained scenario the learning controller outperforms the fixed time controller, but that there does not seem to be any improvement over time. The unconstrained scenario however shows improvement over time but thus far has not improved a lot over the performance level of the fixed-time controller after four days of training.

Figure 8: The running average of the vehicles’ arrival delay, taken over the last 10 arrived vehicles for the Action-Value Table controller using the non-constant demand function. Without constraints (left) and with constraints (right).



(a) Without phase duration constraints.

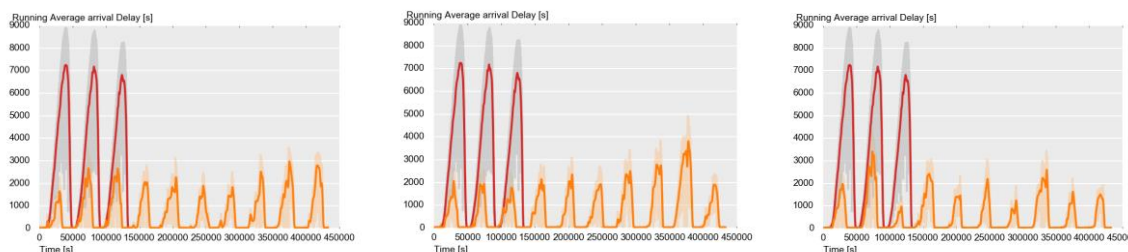
(b) With phase duration constraints

### 5.3 Varying penetration rates

When all vehicles are known to be connected vehicles, i.e. vehicles that can send and receive data, it is possible to use intersections without traffic lights and let the vehicles communicate among each other or with a central controller to determine an order for passing through the intersection. However since the transition to intelligent or connected vehicles likely will not happen overnight but gradually, it is interesting to see how smart intersections with traffic lights perform under different penetration rates of connected vehicles.

To do so we have also ran simulations with penetration rates of 80%, 60% and 40%, using the constrained discrete controller. To simulate different penetration rates we generated cars to be of type 'Simple' or 'Connected' with the corresponding probabilities. The delay information that is passed as the state to the controller is then computed only from the delays of the connected vehicles. The reward that we used as feedback however did contain the full information on the delays from all cars in the simulation. The results of this in combination with a constrained controller are shown in Figure 9.

Figure 9: The running average of the vehicles' arrival delay, taken over the last 10 arrived vehicles for the constrained Action-Value Table controller, for different penetration rates.



(a) 80% Connected vehicles. (b) 60% Connected vehicles. (c) 40% Connected vehicles.

## 5. Conclusion and Discussion

### 5.1 Neural Fitted Q-learning

Like anticipated in literature, we also experienced a lot of difficulties with fitting an ANN on the Action-Value function. Employing constraints on the phase duration did help to keep the traffic jams below reasonable lengths but it is hard to say whether the trained network had any hand in the performance in those simulations. It might be that the performance here is due purely to the constraints, which we also see in the simulations with the discrete controller.

### 5.2 Matrix based Q-learning

The matrix based controller on the other hand was a lot more robust, the intersection delay never blew up like it did in the simulations with the unconstrained Action-Value Network. Furthermore in the simulations with the non-constant demand functions we saw that there was a large performance gain over the fixed time controller when we used the phase duration constraints. We do however suspect that this performance gain should be mostly attributed to the constraints that we imposed, but nevertheless it performed much better on the varying demand scenario and can thus be considered more adaptive than the fixed time controller. From traffic engineering theory on fixed time controllers we know that in heavily congested situations it is more efficient to use long green phases. Since the constrained controller is likely to use longer phases than the traditional controller, it is therefore not a surprise that during the peak hours it performed much better. However when we look at the results from the unconstrained controller which sometimes used even longer phase durations, we can also conclude that the regularization that is implied by the constraints that we used on the phase duration is also an important factor.

### 5.3 Future work

The results do not seem to be promising in this series of simulations. There are several explanations and future directions. First, the traffic states, reward function should be chosen more carefully. More definitions need to be tested. Second, as the dimension of traffic states may increase to quite high, it is necessary to handle the complexity by neural network. However, the topology of the neural network should be scrutinized.

## 6. References

- Abdulhai, B., Pringle, R., & Karakoulas, G. J. (2003). Reinforcement learning for true adaptive traffic signal control. *Journal of Transportation Engineering*, **129**(3), 278-285.
- El-Tantawy, S., Abdulhai, B., & Abdelgawad, H. (2013). Multiagent reinforcement learning for integrated network of adaptive traffic signal controllers (MARLIN-ATSC): methodology and large-scale application on downtown Toronto. *IEEE Transactions on Intelligent Transportation Systems*, **14**(3), 1140-1150.
- Gilmore, J. F., Elibiary, K. J., & Abe, N. (1993). Traffic Management Applications of Neural Networks. In *Working Notes, AAAI-93 Workshop on AI in Intelligent Vehicle Highway Systems*.
- Grégoire, P. L., Desjardins, C., Laumônier, J., & Chaib-draa, B. (2007). Urban traffic control based on learning agents. In *Intelligent Transportation Systems Conference, September 2007. ITSC 2007*. IEEE.
- Krajzewicz, D., Erdmann, J., Behrisch, M., & Bieker, L. (2012). Recent development and applications of SUMO-Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, **5**(3&4).
- Kuyer, L., Whiteson, S., Bakker, B., & Vlassis, N. (2008). Multiagent reinforcement learning for urban traffic control using coordination graphs. *Machine learning and knowledge discovery in databases*, 656-671.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G. & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, **518**(7540), 529-533.
- Ng, A., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B. & Liang, E. (2006). Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX*, 363-372.
- Robertson, D. I. (1969). TRANSYT: a traffic network study tool.
- Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., ... & Schmidhuber, J. (2010). PyBrain. *Journal of Machine Learning Research*, **11**(2), 743-746.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G. & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, **529**(7587), 484-489.
- Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction* (Vol. 1, No. 1). Cambridge: MIT press.
- Wiering, M. (2000). Multi-agent reinforcement learning for traffic light control. In *ICML June, 2000*.