TRAFFIC LIGHT CONTROL USING DEEP REINFORCEMENT LEARNING AT AN ISOLATED INTERSECTION WITH PEDESTRIANS

Master's Thesis

June 3, 2020

Daniel Rytter Sørensen - Nicolai Tandal Danielsen
Aalborg University
Mathematics - Economics

AALBORG UNIVERSITY
DENMARK

This page is intentionally left blank.

AALBORG UNIVERSITY
DENMARK

Faculty of Engineering and Science
Mathematics-Economics
Aalborg University
https://www.en.engineering.aau.dk/

Title:
Traffic Light Control Using Deep Reinforcement Learning At An Isolated Intersection With Pedestrians

Theme:
Deep Reinforcement Learning

Project period:
Februrary 3, 2020 - June 3, 2020

Project group:
3.222a

Group members:
Daniel Rytter Sørensen
Nicolai Tandal Danielsen

Supervisor:
Inkyung Sung
Mohamed El Yafrani

Co-supervisor:
Peter Nielsen

Number printed: 6 Pieces

Report number of pages: 30

Appendix number of pages: 2

Total number of pages: 34

Abstract:

The cost of traffic congestion is large, estimated to be roughly 110 billion euros in the EU alone. The best way to reduce this cost while utilizing the road infrastructure that is already in place, is to optimize traffic light control. An approach that has received a lot of attention is reinforcement learning based traffic light control. It has been widely researched, and many researchers have proposed methods that show promising results. However, most of the research focuses on traffic light control solely for motorized traffic and neglects the non-motorized aspect of the traffic. In this thesis we train a dueling double deep Q-network reinforcement learning algorithm with prioritized experience replay using a simulation of an isolated intersection with pedestrians. We compare its performance to a fixed-time strategy and a heuristic based on the idea of a police officer directing the traffic. We also investigate how this reinforcement learning algorithm compares to an identical algorithm trained in an environment without pedestrians. We test both of these algorithms in the same simulated test environment with pedestrians, and compare their behavior and performance.

Preface

This report has been written as a Master's thesis on the 10th semester of Mathematics-Economics at the Faculty of Engineering and Science at Aalborg University in spring 2020.

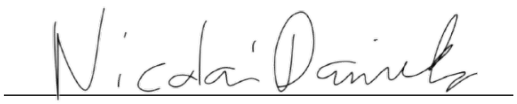
References appear throughout the thesis and the sources used are listed in the last section of the thesis. The source reference used in the thesis is the Harvard method, which has the following structure: [Surname, year, page number]. If there are several sources published by the same person in the same year, then these sources are named in the following structure: [Surname, year(letter), page number].

The references direct the reader to the bibliography, where books are specified by author, title, edition, ISBN, and publishers, while web pages are specified by author, title, institution, and date. Figures, tables, definitions, theorems, examples, equations, etc. are numbered according to chapter. For example, the first figure in Chapter 1 is numbered Figure 1.1. The descriptions of the figures and tables are to be found under the given figure or table.

All programming in this thesis has been done in Python and R. Code for the algorithm used throughout the thesis can be found here: `https://www.dropbox.com/sh/8otups4x2jexwvz/AACAzgaKNElR2k5Y_0lq6mYsa?dl=0`

Finally, we express gratitude for the guidance and constructive criticism of Inkyung Sung, Mohamed El Yafrani and Peter Nielsen.

Daniel Rytter Sørensen
<dsaren15@student.aau.dk>



Nicolai Tandal Danielsen
<ndanie15@student.aau.dk>

Table of Contents

Introduction

Improving traffic light control is one of the main ways of improving traffic conditions and reducing congestions in urban areas. Better traffic light control leads to higher traffic throughput, which means less delay and thus higher quality of life for a commuting citizen. Furthermore, the cost of traffic congestion in the EU alone is estimated to be 1% of EU's GDP [Christidis and Rivas, 2012, p. 1], which is roughly 110 billion euros annually. Because of the complex nature of the traffic system, improving traffic light control remains a difficult task. In recent years traffic light control based on reinforcement learning (RL) has received increasing attention because of its success in other fields.

Early works [Abdulhai et al., 2003] focused on a single-agent case with an isolated intersection. They used a simple state space and a sort of tabular Q-learning algorithm that requires storing the Q-values in hash-tables. Later, [LA and Bhatnagar, 2011] used reinforcement learning with function approximation in a traffic control setting, and according to the authors, they were the first paper that was able to incorporate high-dimensional full-state representation.

In [Mannion et al., 2016] they design a multi-agent experiment, using queue length, current green light phase, and elapsed time of the current phase, as the state definition. They find that their RL algorithm outperforms the commonly used fixed-time strategy when the traffic flow is random. Other papers, such as [Gao et al., 2017], use vehicular position and speed as state information and also find promising results.

With the rapid advancements within deep learning, more recent works have been using deep neural networks to approximate the value function. In [Liang et al., 2018] they combine deep neural networks with multiple state-of-the-art techniques from reinforcement learning and find that a dueling double deep Q-network (DDDQN) with prioritized experience replay (PER) performs best in their single-agent traffic control problem.

However, in none of the aforementioned papers pedestrians are included. The vast majority of the research within this field has focused on traffic light control for motorized traffic and neglected non-motorized traffic. Only a few papers include pedestrians in their simulation or otherwise focus on the non-motorized perspective rather than motorized traffic. In [Yin and Menendez, 2019] they include pedestrians, using ADP instead of deep reinforcement learning, and find that their approach outperforms a fixed-time strategy and a longest-queue-first strategy for average vehicle wait time, but fails to outperform longest-queue-first for average pedestrian wait time. [Xie et al., 2014] uses schedule-driven traffic control in an attempt to balance the wait times of pedestrians and vehicles in a weighted manner and achieves a weighted wait time significantly lower than that of a fixed-time strategy.

When including pedestrians in the reinforcement learning algorithm, the objective function becomes conflicted, as vehicles are now halted by crossing pedestrians. If we prioritize vehicles, we force the pedestrians to wait. Conversely, giving the pedestrians priority results in a longer waiting time for vehicles. Hence, including pedestrians introduces a new aspect to consider when defining how to reward the RL algorithm.

While some work has been done to include pedestrians, it is not using recent reinforcement learning techniques, such as DDDQNs. To the best of our knowledge, no paper has included both pedestrians and DDDQNs with PER. It is not unrealistic to think that the optimal policy obtained in an environment without pedestrians is not necessarily the optimal policy when pedestrians are present, since pedestrians may block cars from turning. Hence, in this thesis we examine how (if) the optimal policy changes when accounting for pedestrians, and examine if the DDDQN with PER approach still outperforms other methods such as the fixed-time method. If the policy changes significantly, it is a clear indication that pedestrians should be included in the simulation when training the agent. Otherwise, traffic light control may be suboptimal and vehicle throughput is likely lower than expected, possibly causing congestions.

We choose to focus on the impact of considering pedestrians in an isolated intersection during rush hour. While most intersections are part of a traffic network with multiple adjacent intersections, we argue that, if pedestrians impact the policy of an isolated intersection, they are also likely to impact an intersection that is part of a larger network. For this reason, and due to time limitations, we choose to focus on the isolated intersection case. Furthermore, we focus on rush hour, as it is during rush hour that the largest congestions occur, and therefore the rush hour situation is the most interesting case to study.

1.1 Problem Formulation

How does the behavior of a dueling double deep Q-network reinforcement learning algorithm with prioritized experience replay change when including pedestrians in an isolated intersection traffic light control problem during rush hour, compared to excluding pedestrians? And is it still able to outperform simple traffic light control strategies when including pedestrians in the traffic simulation?

Reinforcement Learning

Before we introduce our implementation of the DDDQN, we briefly go over the necessary theory.

The reinforcement learning approach is based on how many scientists believe humans learn: by interacting with an environment and observing how our actions affect the state of the environment, cf. [Sutton and Barto, 2015, Chapter 1]. For example, consider a young child playing with a sorting box, in which blocks with a certain shape has to go through matching holes in the box. While the child has no explicit teacher, he eventually learns what blocks go through which hole by interacting with the blocks and the sorting box. The idea in reinforcement learning is much the same; an agent interacts with the environment and learns by evaluating how good that interaction was. In order to evaluate its action, the agent receives a numerical reward based on how its action affected the environment. In the context of the sorting box scenario, the reward could be 1 when a block matches the hole and 0 otherwise. The learning process can be described as a loop:

- The agent observes state $s_t \in \mathcal{S}$ from the environment.
- Based on state s_t the agent takes an action $a_t \in \mathcal{A}$.
- The action a_t causes the environment to transition to a new state, $s_{t+1} \in \mathcal{S}$.
- The agent receives reward r_t from the environment.

$\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. The goal of the agent is then to maximize the cumulative reward it receives. The cumulative reward at step t is given by:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \end{aligned} \tag{2.1}$$

where γ is the discount factor that determines the present value of future rewards.

The described learning process and cumulative reward are used in the DDDQN. At its core, DDDQN is a Q-learning algorithm, but with extensions to either adapt to a more complex setting or improve performance.

2.1 Deep Q-learning

Q-learning, proposed by [Watkins and Dayan, 1992], is a model-free reinforcement learning algorithm. In its first version, what is referred to as tabular Q-learning, it was used to efficiently solve a Markov Decision Process (MDP).

The goal is to determine the best action to take, such that the cumulative reward, given by (2.1), is maximized. To measure the value of an action, we use the Q-function. The

function takes state and action as input, and returns the expected cumulative reward of all future rewards given that the agent is in that state and takes that action:

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t].$$

For each possible action at a given state, we calculate the Q-value using the Q-function and choose the action with the highest Q-value.

However, in the case of a large or infinite state space or action space, tabular Q-learning is infeasible as it then would be impossible to learn the value of each state-action pair. To overcome this issue, deep Q-learning networks (DQN) were introduced. In DQN, a neural network is used to approximate the Q-value function. The current state is given as input, and the Q-values of all possible actions in that state are returned as the output. In DQN, we update the weights, θ, of the neural network, such that the difference between the current estimate of the Q-value and the real Q-value is reduced. However, we do not know the real Q-value, and thus need to estimate it, which we do using the following equation:

$$y_t = r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a; \theta_t).$$

That is, the reward of the chosen action in our current state plus the discounted best Q-value of the next state. We refer to this value as the Q-target, and the difference between the current estimate of the Q-value and the Q-target is referred to as the temporal difference error (TD error):

$$\delta_t = y_t - Q(s_t, a_t; \theta_t).$$

The weights of the neural network are then updated using the TD error in combination with a gradient descent method and a learning rate. Some popular gradient descent methods include stochastic gradient decent and ADAM.

2.1.1 Double Deep Q-learning

At the beginning of training, we do not have much information about the best action to take. Therefore, taking the maximum Q-value as the best action can lead to an overestimation of a non-optimal action. To handle this problem, double DQN was introduced by [van Hasselt et al., 2015]. When calculating the Q-target, we now use two networks; one network, denoted the primary network, is used to choose the best action to take for the next state, while a second network, called the target network, is used to calculate the Q-value of taking that action. Thus, the Q-target is now calculated in the following way:

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \arg\max_a Q(s_t, a; \theta_t^P); \theta_t^T)$$

where θ_t^P are the weights of the primary network and θ_t^T are the weights of the target network at step t. The weights of the target network are generally updated using one of two approaches: Either the target weights are set to be equal to the primary weights after k steps, or the target weights are updated each step using the following equation:

$$\theta_t^T = (1 - \tau)\theta_{t-1}^T + \tau\theta_t^P, \tag{2.2}$$

where τ, $0 < \tau < 1$, is a hyperparameter.

2.1.2 Dueling Deep Q-learning

Dueling DQN, [Wang et al., 2015], introduces a new architecture of the DQN that accelerates learning and results in better performance. Intuitively, the value of $Q(s, a)$ reflects how good it is to be in state s and take action a at that state. We can decompose the Q-value into the sum of:

- $V(s)$: The value of being at state s
- $A(s, a)$: The advantage of taking action a at state s, that is, how much better it is to take action a than any other possible action.

The idea in dueling DQN is to separately estimate these two values in the neural network and in the end combine them to calculate the Q-values. Thus, we now have a stream that estimates the state value, $V(s)$, and a stream that estimates the advantage of each action, $A(s, a)$. These two streams pass through an aggregation layer to get an estimate of $Q(s, a)$. By splitting the estimation of $Q(s, a)$ up into two streams we can learn which states are valuable, without having to learn the effect each action has at each state.

The Architecture of a Dueling Deep Q-Network

In Figure 2.1 we illustrate the architecture of a dueling DQN. First, the inputs are passed through 1 or more dense (fully connected) layers. The network is then split into the value and advantage streams. The streams are then combined in the aggregation layer to obtain the Q-values of each state-action pair using the following equation:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{n} \sum_{i}^{n} A(s, a_i),$$

where n is the number of actions.

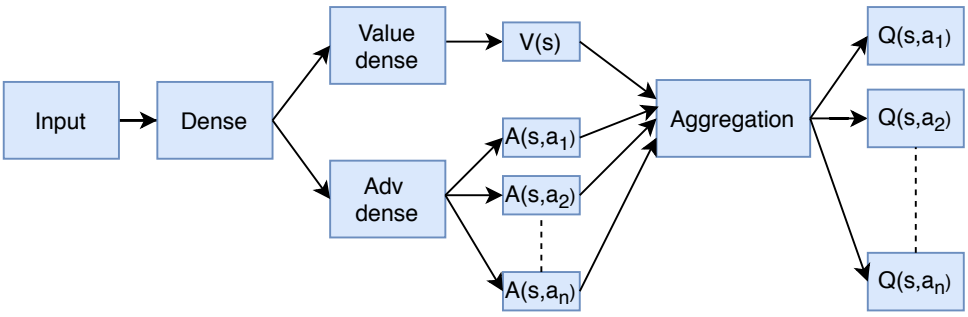


Figure 2.1: The architecture of a dueling deep Q-network.

2.2 Prioritized Experience Replay

In decision-making under uncertainty, such as reinforcement learning, consecutively sampled data points (s_t, a_t, r_t, s_{t+1}) are heavily correlated, and not independently and

identically distributed as we often assume in traditional machine learning. That is, (s_t, a_t) heavily influences (s_{t+1}, a_{t+1}). This correlation worsens the convergence and results in slower learning [Pol, 2016, p. 10]. To break this correlation and improve the learning, we use experience replay, first introduced by [Lin, 1992]. In experience replay we store past experiences in a replay memory in the form of experience tuples (s_t, a_t, r_t, s_{t+1}). Instead of using the most recent experience to update the network, the agent now samples a batch of experiences from the memory. In addition to breaking the correlation, this sampling also, in theory, lets the agent learn from the same experience multiple times. However, using experience replay means more hyperparameters must be estimated, specifically the size of the replay memory and the batch size.

Instead of sampling uniformly from the replay memory as is done in regular experience replay, prioritized experience replay (PER), proposed by [Schaul et al., 2015], samples experiences from the replay memory based on their TD error. The idea is that the experiences with a high TD error are the experiences the agent can learn the most from. By sampling the experiences with high TD errors more frequently, our agent is able to learn more efficiently. However, instead of using a greedy approach, in which we always sample the experiences with the highest TD errors, we use proportional prioritization. First, we convert the TD error into a priority:

$$p_t = (|\delta_t| + \epsilon)^\alpha \,, \tag{2.3}$$

where ϵ is a small positive number that ensures no experience has zero priority and α is a hyperparameter. This priority is then transformed into a probability. For the t-th experience, the probability is computed as:

$$P(t) = \frac{p_t}{\sum_k p_k}.$$

If $\alpha = 1$ greedy prioritization is used, while $\alpha = 0$ results in uniform sampling. In addition to the experience tuples, we must now also store the priority for each experience in the memory.

Furthermore, since we are now using prioritized sampling, we introduce bias towards experiences with high priorities, since they will be selected more often. Hence, if we update the weights in the neural network normally, we risk overfitting to the experiences with high priorities. We correct this bias by using importance sampling (IS) weights:

$$w_t = \left(\frac{1}{N} \cdot \frac{1}{P(t)}\right)^\beta.$$

If $\beta = 1$, the IS weights fully compensates for the non-uniform probabilities of being sampled. We then update the network weights by using $w_t\delta_t$ instead of δ_t. In practice, β often starts at an initial value β_0 and linearly increases to 1 as the learning progresses. This approach is used since correcting the bias in the update of the weights is most important near convergence, and a small bias in the beginning of the learning can be neglected due to the non-stationary nature of the process. Lastly, after training on a sampled batch of experiences, the priorities of the sampled experiences are updated.

System Setup and Configuration

In this chapter we describe the dynamics of the simulated intersection, the reinforcement learning model, and the implementation of the intersection and the reinforcement learning algorithm.

Note that while some commercial traffic simulation software has the option to include pedestrians, we choose to make the simulation ourselves as it lets us fully control how the simulation works.

3.1 Traffic Simulation

We choose to base our simulation on an intersection in Aalborg, Denmark; the intersection between Hadsundvej and Humlebakken. This intersection has the layout depicted in Figure 3.1.

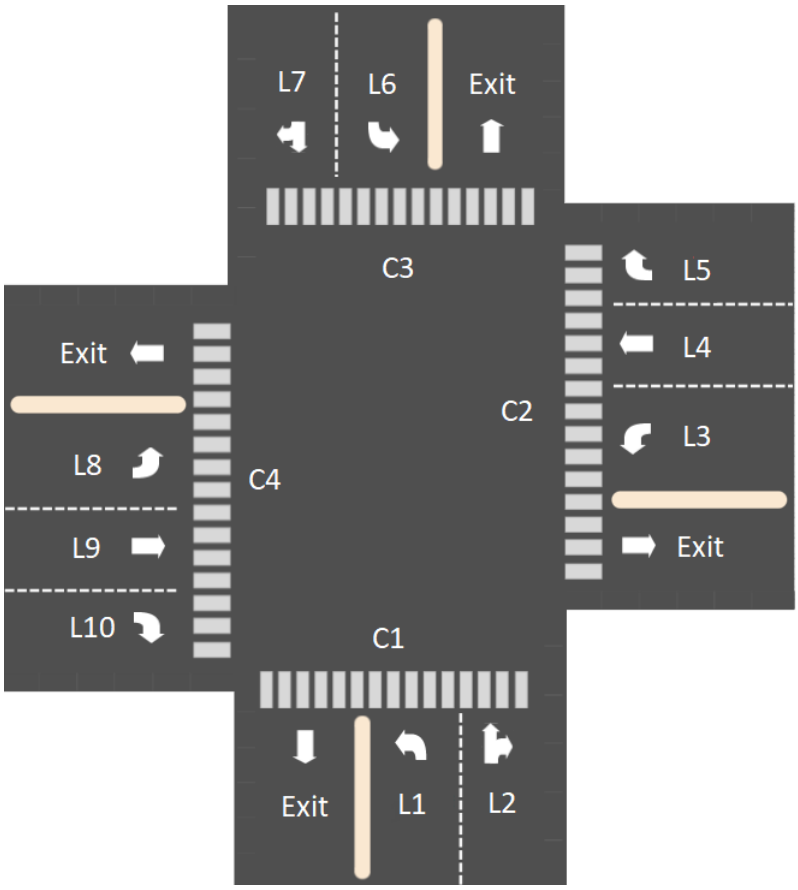


Figure 3.1: Intersection layout.

We choose a real intersection to make the setting somewhat realistic and choose this specific intersection as it makes it convenient to gather data and establish a light cycle, as it is located close to Aalborg University main campus. Furthermore, this intersection is often

highly congested during rush hour, which makes it a fitting location to implement more efficient traffic light control.

By observing the intersection, we find that the traffic light follows the following cycle:

$$(\text{L1, L2, L6, L7, C2, C4}) \rightarrow (\text{L1, L5, L6, L10}) \rightarrow (\text{L3, L4, L5, L8, L9, L10, C1, C3}).$$

That is, a phase for the north-south lanes and crosswalks, a phase to clear some of the turn lanes, and a phase for the east-west lanes and crosswalks. Henceforth we refer to these phases as phase 1, phase 2, and phase 3, respectively.

Light Dynamics

The pedestrian lights turn red a few seconds before the traffic lights. Between each phase the light is red for approximately five seconds to clear the intersection. However, when phase 1 ends, lanes 1 and 6 remain green since they are also green in phase 2.

Furthermore, we do not include yellow light, since the purposes of yellow and red lights are much the same; to make sure no new object enters the intersection such that the intersection is clear before the next green light begins. Instead, we combine the two lights into a slightly longer red light phase for simplicity.

Pedestrian Dynamics

We assume pedestrians at crosswalks 1 and 3 take 7 seconds to cross as these crosswalks are the shortest and pedestrians at crosswalks 2 and 4 take 9 seconds. For pedestrians crossing from a side where the exit lane is the closest, we assume pedestrians block vehicles going to that exit lane for the first 3.5 seconds after they start crossing. Conversely, pedestrians starting from the other side of the crosswalk is assumed to block the vehicles for the last 3.5 seconds of their crossing. For example, a pedestrian crossing crosswalk 1 from the left side blocks the exit lane for the first 3.5 seconds of his crossing, while a pedestrian crossing from the right side blocks the exit lane for the last 3.5 seconds of his crossing.

At some intersections cyclists also have a lane in each direction, located at the right-most side, allowing cyclists to either go straight forward or turn right. Thus, cyclists is another element that can halt vehicle throughput. However we do not choose an intersection layout that includes this aspect, and instead argue that cyclists going straight will dismount the bicycle and use the crosswalk, acting as a pedestrian and cyclists turning right does not interfere with the traffic.

Vehicle Dynamics

Since vehicles in lanes 2 and 7 can choose to either drive forward or turn right, we assign a probability of turning right at these lanes. We have no data to estimate these probabilities, and simply choose 0.3 as the probability for lane 2 and 0.2 for lane 7.

A vehicle is only able to enter the intersection if it has green light and no other vehicle from the same lane is in the intersection. While in the intersection, the vehicle checks that there are no oncoming traffic or pedestrians at the moment, or within the next x seconds, blocking the exit lane. x corresponds to the required time for the vehicle in question to

reach the exit lane and get clear of the intersection and thus depends on which lane is considered and the flow of that lane.

Lane Flow

If there is flow in the considered lane, the vehicle is able to cross faster than when the traffic is stop and go. For example, consider a scenario in which the light for lane 2 turns green and the first four vehicles in the queue all wants to go straight forward, but the fifth vehicle wants to turn right. The first vehicle has to accelerate from complete stop at the stop line, whereas the second vehicle is already in motion when he reaches the stop line. Hence the second vehicle crosses the intersection slightly faster than the first vehicle. The same goes for the third and fourth vehicle. However, when the fifth vehicle reaches the intersection, he may have to brake if there are pedestrians on the crosswalk. In such a case, the flow is broken since the vehicles have come to a complete stop. A description of the flow types can be found in Table 3.1.

Flow type	When activated
No flow	For the first vehicle after green light, and when a vehicle is blocked by oncoming traffic or pedestrians.
Semi-flow	Only relevant for lanes 2 and 7. Activated only when a vehicle is going straight forward after the most recent vehicle turn right.
Flow	When the vehicle is not the first to cross after a no flow has been activated and the vehicle is not blocked by oncoming traffic or pedestrians.

Table 3.1: Description of flow types.

We choose to include an option called semi-flow, as we believe the flow of the lane is decreased when the vehicle in front is turning and the next vehicle is going straight, but not enough to classify it as no flow.

In Table 3.2, we list the time it takes for a vehicle to cross the intersection depending on which direction the vehicle is going and what the flow is. In the turning lanes, we assume vehicles drive into the intersection while waiting for either oncoming traffic or pedestrians to pass, rather than waiting at the stop line. Hence, no flow has two values in the table, depending on the situation. Furthermore, the times have slightly different interpretations. The no flow times reflect how long it takes for the vehicle to exit from its current position (at the stop line or in the intersection). However, the semi-flow and flow times reflect the time between the vehicle in front exits the intersection and the current vehicle exits. These interpretation differences are a result of the implementation choice of having max one vehicle from each lane in the intersection at all times.

	No flow (at stopping line)	No flow (in intersection)	Semi-flow	Flow
Left turn	4	2	-	2
Forward	3	-	2	1
Right turn	3	1	-	1.5

Table 3.2: Time it takes to cross the intersection (in seconds).

3.2 Reinforcement Learning Model

With the traffic simulation designed, we now only need to define the state space, the action space, and the reward function, in order to implement the RL algorithm.

State Space

In the literature, various different approaches have been used to define the traffic control probem as a reinforcement learning problem. In [Genders and Razavi, 2016] they define the state based on the position and speed of the vehicles. They segment the intersection into a grid, and use a matrix with binary values to store the information. If the grid-section is occupied by a vehicle, its value is 1, otherwise it is 0. Furthermore, they use a one-hot vector to indicate which phase is currently active.

Other papers, see [Kwong et al., 2011], use information of the queue lengths in their state definition as well as a one-hot vector to represent the current phase. While some papers, e.g. [Liang et al., 2018], argue that this is not sufficient information, we argue that this approach is more appropriate when considering pedestrians as well. Since pedestrians are present in our intersection, we would need to segment the intersection into a more detailed grid than what is usually done in the grid-approach. Normally the intersection is segmented into a grid such that each grid-section is the size of a vehicle. However, to be able to distinguish individual pedestrians we would need a much smaller grid-section size, which would result in a much larger computational cost. Therefore, we choose to define our state space using the second approach. Thus, the state space is:

$$\mathcal{S} = \{Q^{L_1}, Q^{L_2}, \ldots, Q^{L_{10}}, Q^{C_1}, \ldots, Q^{C_4}, p\},$$

where Q^{L_i} and Q^{C_j} are the set of possible queue lengths at lane i and crosswalk j, respectively, and p is the set of possible one-hot vectors that indicate the phase.

Action Space

We choose to have the agent determine the green duration of each phase. Thus, before each phase begins, a decision is made to determine how long the green period should be of that phase. To ensure stable learning of the agent, we need to ensure that the agent is able to explore the entire action space well, before it settles on a policy. This means that the action space cannot be too large. Therefore, to keep the action space small, we choose to use increments of 5 from 15 seconds to 45 seconds. Hence, the action space is:

$$\mathcal{A} = \{15, 20, 25, 30, 35, 40, 45\}$$

We choose 15 seconds as the shortest green duration as we deem that making the phases shorter, or completely skipping them, might result in driver confusion and impatience, which could lead to reckless driving and accidents.

Reward Function

Intuitively, the agent should be rewarded when the waiting time of all vehicles and pedestrians is decreased and punished when it is increased. Hence, we base our reward

function on the reward function from [Liang et al., 2018], in which they define it as the increment in cumulative waiting time between two actions:

$$r_t = W_t - W_{t+1},$$

where W_t is the cumulative waiting time of all vehicles and pedestrians in every lane/crosswalk till phase t. We calculate W_t in the following way:

$$W_t = \sum_{i=1}^{N_t} w_{i,t}$$

where N_t is the total number of vehicles and pedestrians till the t-th phase and $w_{i,t}$ denotes the waiting time of object i till the t-th phase. However, preliminary tests using this reward function did not show stable learning. Nonetheless, we found that slightly altering the initial reward function by dividing it by the number of vehicles and pedestrians in the intersection between two states results in stable learning. Thus, the reward function becomes:

$$r_t = \frac{W_t - W_{t+1}}{H_t}, \tag{3.1}$$

where H_t is the number of objects in the intersection at the start of phase t plus the number of objects arriving during phase t.

Note, the objective function of vehicles and pedestrians are conflicting which means that giving vehicles priority results in longer waiting time for pedestrians, and vice versa. One could argue that, because vehicles often carries more than one person, the agent should be rewarded more for getting a vehicle through the intersection than for getting a pedestrian through. However, in this thesis we choose to treat vehicles and pedestrians as equals, and therefore the reward function puts no priority on vehicles nor pedestrians.

3.3 Reinforcement Learning Implementation

In Algorithm 1 our implementation of the intersection dynamic is outlined in the form of pseudo-code. We implement it as a class with two functions: Reset and Step. The Reset function is used to reset the environment and generate the vehicles and pedestrians. The Step function is used to "take one step" in the simulation, that is, execute all the events of the current phase. The Step function takes action as an input, which determines the green light duration of the phase.

As stated in the introduction, our RL algortihm uses a DDDQN with PER. We outline this RL algorithm as a pseudo-code in Algorithm 2. Before a simulation, henceforth referred to as an episode, begins, we reset the environment. While the memory has not reached its maximum capacity, the agent takes random actions, and the Q-networks are not updated. Once the memory has been filled, the learning process begins.

When choosing an action, the agent can either take a random action, thus choosing to explore, or take the action with the highest Q-value, exploiting what it currently knows. It is common to have the agent explore at the beginning of the learning process, but gradually make it exploit more, as it becomes more certain of the Q-value estimates. In

Algorithm 1: Intersection environment

```
Class Intersection
    Initialize:
        Create the 10 lanes and 4 crosswalks of the intersection
    Procedure Reset (activates just before an episode begins)
        Create an event queue to hold events. Initially empty
        Generate arrival times for vehicles and pedestrians at each lane and crosswalk
         for the simulation duration
        Add arrivals to the event queue
        Empty all lanes of vehicles and all crosswalks of pedestrians
        As the first event, queue a green light event for phase 1
    End procedure Reset
    Procedure Step (activates whenever an action is taken):
    Input: Action
        Green light is activated for the lanes and crosswalks of the current phase,
         duration of green light depends on action taken
        while Not end of phase do
            Complete next event in event queue
            Check if end of phase
        end
        As the next event, queue a green light event for the next phase
    End Procedure Step
End Class Intersection
```

our implementation we use the ε-greedy algorithm to determine when to explore and when to exploit. Initially the exploration rate, ε, is 1 which means that it has a probability of 1 of exploring. As the learning progresses, the exploration rate linearly decays towards a predetermined minimum value.

After interacting with the environment, the networks of the agent are trained on a batch of experiences, sampled using PER. The weights of the primary network are updated using ADAM as the gradient descent method, while the target network weights are updated using (2.2). ADAM, proposed by [Kingma and Ba, 2014], is an algorithm for efficient gradient-based optimization that only requires the first-order gradients. The first-order gradients are used to compute estimates of the first and second moment of the gradients, which are then used to update the parameters. It is well suited for optimization in high-dimensional parameter spaces, such as in deep neural networks. It is one of the recommended algorithms for gradient-based optimization in the literature, according to [Ruder, 2016].

After training, the priorities of the experiences used to train on are updated in the memory, and the oldest experience in the memory is then replaced with the new experience. Lastly, the exploration rate is decayed towards its minimum and the bias correction parameter in PER is increased towards its maximum.

Algorithm 2: Learning Algorithm

```
Initialize Q-networks with random weights, θ^P and θ^T
Initialize intersection environment
Create empty prioritized experience replay memory of size M
for i = 0; i < number of episodes; i++ do
    t = 0
    Reset environment
    while Not end of episode do
        if experiences in memory < M then
            Interact with environment:
                Observe state s_t
                Take random action a_t
                Receive reward r_t
                Observe new state s_{t+1}
            Define experience priority as absolute value of reward; p_t := |r_t|
            Add the experience tuple and priority (s_t, a_t, r_t, s_{t+1}, p_t) to the memory
        end
        else
            Interact with environment:
                Observe state s_t
                With probability ε:
                    Take random action a_t
                Otherwise:
                    Take greedy action; a_t = arg max_a Q(s_t, a; θ_t^P)
                Receive reward r_t
                Observe new state s_{t+1}
            Train and update networks:
                Using PER, sample a batch of experiences from memory
                Calculate new priority for each experience in batch using (2.3)
                Update the priorities of each experience in batch in the memory
                Update θ^P using ADAM
                Update θ^T using (2.2)
            Calculate priority of the new experience using (2.3) and the newly updated
             networks
            Replace the oldest experience in the memory with the new experience
            Decay ε linearly if ε has not yet reached its minimum
            Increase β linearly if β has not yet reached its maximum
        end
        t = t + 1
        Check if end of episode
    end
end
```

4

Experiments with Pedestrians

In this chapter we put our DDDQN algorithm to use. First we make a preliminary experiment set in a simple scenario, to determine appropriate values of the algorithm's hyperparameters such that the algorithm learns well. In this simple setting we are able to evaluate whether the policy that our agent finds is good, whereas in a more realistic setting it would be more difficult to determine. After ensuring that the agent learns well, we test it in a more realistic scenario, and compares its performance to two simple strategies.

4.1 Preliminary Experiment

In this setting, we assume that only lanes 2 and 7 are open as well as crosswalks 2 and 4, and that all other lanes and crosswalks are closed. Hence, there are only vehicles arriving in lanes 2 and 7, and there are only pedestrians arriving at crosswalks 2 and 4. We use independent exponential distributions with the parameters listed in Table 4.1 to generate the interarrival times of the vehicles and pedestrians.

λ_2^L	λ_7^L	λ_2^C	λ_4^C
$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{90}$	$\frac{1}{90}$

Table 4.1: Interarrival parameters.

Since only lanes 2 and 7 and crosswalks 2 and 4 are open, the vehicles and pedestrians can only leave the intersection in phase 1. Thus, the agent should learn to pass through phases 2 and 3 as fast as possible and often choose a long green light duration for phase 1.

Through testing, we find that the parameter values listed in Table 4.2 yield fast and steady learning. Thus, we train the agent using these parameter values. Initially, the exploration rate, ε, is set to 1, which means that the algorithm always takes a random action. Once the memory is filled, the exploration rate linearly decreases, with $\varepsilon = 0.01$ as it minimum.

The DDDQN we use has the architecture that was illustrated in Figure 2.1, except our network has two initial hidden layers instead of just one. Moreover, the activation function used in each hidden layer is the rectifier (ReLU). We use the rectifier as it provides faster and better learning in deep networks than other activation functions (such as sigmoids and tanh) [Goodfellow et al., 2016, p 226]. To make sure the agent is learning well, we examine the average reward and loss per episode. The average reward of the i-th episode with n_i steps is:

$$\overline{r}_i = \frac{1}{n_i} \sum_{t=1}^{n_i} r_t,$$

where r_t is defined as in (3.1).

Parameter	Value
Memory size, M	500,000
Number of episodes	15,000
Duration of an episode	1 hour
Learning rate	0.0001
Batch size	32
Exploration rate, ε	$1 \rightarrow 0.01$
Discount factor, γ	0.95
Update parameter for target network weights , τ	0.08
Priority parameter of PER, α	0.6
Bias correction parameter of PER, β	$0.4 \rightarrow 1$
Steps until ε reaches its minimum and β reaches its maximum	350,000
Number of neurons in each hidden layer of the network	50

Table 4.2: Various parameter values of the simple experiment.

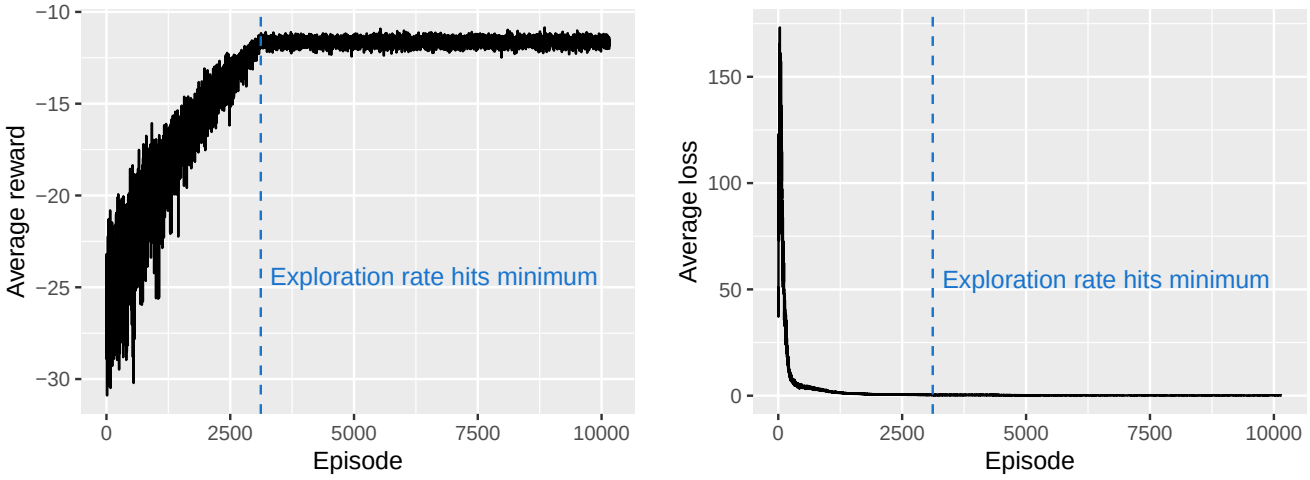
Furthermore, the average loss of the i-th episode is defined as

$$\bar{l}_i = \frac{1}{n_i} \sum_{t=1}^{n_i} \bar{\delta}_t,$$

where $\bar{\delta}_t$ is the average TD error of the sampled batch in the t-th step. Hence, the average loss per episode is a measure of how close the estimated Q-values are to the Q-target values on average during the episode.

The average reward and average loss for each episode can be seen in Figure 4.1. While we simulate 15,000 episodes, roughly the first 5,000 episodes are spent filling the memory using random actions, and are not used to train the networks, as described in Algorithm 2. Hence, we only plot the average reward and loss after training begins. Looking at the average reward, the agent seems to learn well during the exploration phase as the reward steadily increases. When the exploration rate hits its minimum, the average reward is stable throughout the rest of the experiment. Additionally, the average loss quickly drops to a fairly low value and then very slowly keeps decreasing as the exploration rate approaches its minimum.

The fact that the reward stabilizes is a good sign, since it means that the agent has determined a policy. However, from the reward alone we cannot tell whether it is a good policy, since we are unable to determine what a "good" reward looks like. Instead, we examine the actions of the agent during the exploitation phase. In Figure 4.2, we plot all the actions of the agent in the episodes after the exploration rate has reached its minimum. From the plot we see mostly what we expected. For phases 2 and 3 the agent picks the shortest green light duration possible almost always and rarely picks any other duration. In the situations when the agents does pick a different duration, it is likely due to the fact that the minimum value of the exploration rate is 0.01, which means that 1% of the time the agent picks randomly. Looking at the data, this speculation is supported by the fact that the number of actions made during phases 2 and 3 that are not the shortest duration, correspond to 0.56% of all actions. For phase 1, the agent seems to heavily favor the longest duration possible. However, it also picks shorter durations more often than



(a) Reward function

(b) Loss function

Figure 4.1: Average reward and loss.

what is explainable by the exploration rate being 0.01. Nonetheless, if there are only a few vehicles in queue when the simulation enters phase 1, picking the longest duration possible may be unnecessary, as less time may be enough to empty the queue.

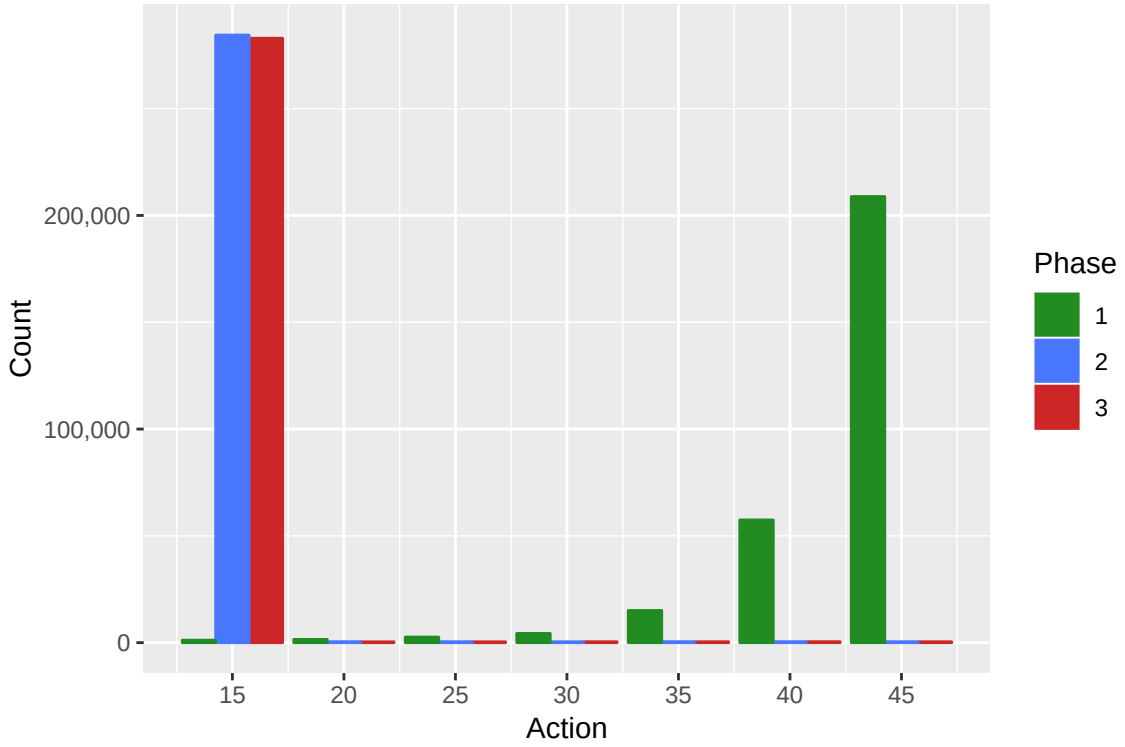


Figure 4.2: Actions taken for the three phases for all episodes after exploration rate reaches its minimum.

In conclusion, the agent seems to learn well, as it is able to determine an appropriate policy. Hence, we move on to a more realistic scenario, in which all lanes and crosswalks are open.

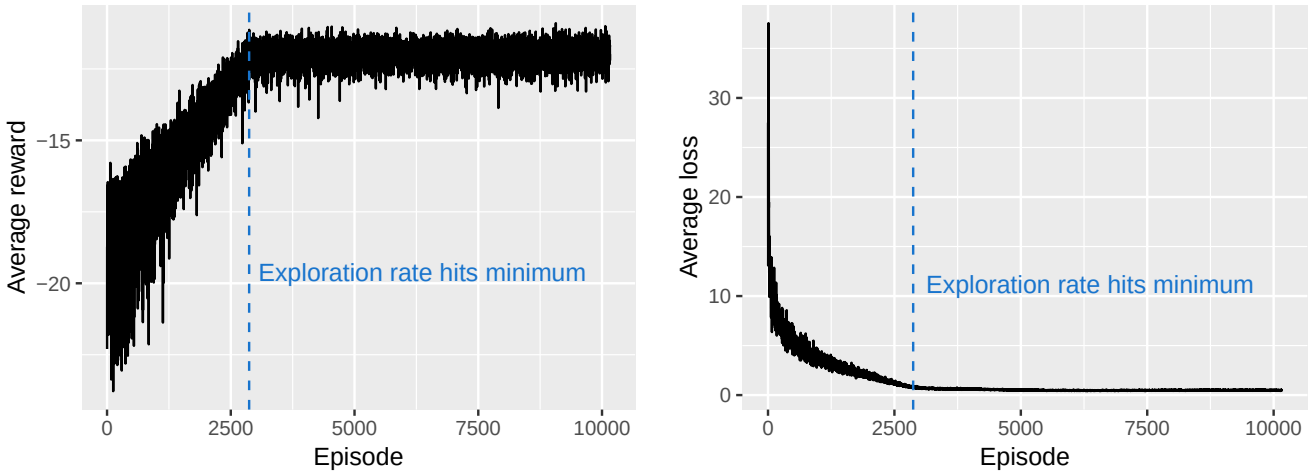
4.2 Main Experiment

In this experiment we generate the interarrival times of vehicles and pedestrians using the exponential parameters listed in Table 4.3.

λ_1^L	λ_2^L	λ_3^L	λ_4^L	λ_5^L	λ_6^L	λ_7^L	λ_8^L	λ_9^L	λ_{10}^L	λ_1^C	λ_2^C	λ_3^C	λ_4^C
$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{16}$	$\frac{1}{15}$	$\frac{1}{13}$	$\frac{1}{12}$	$\frac{1}{13}$	$\frac{1}{18}$	$\frac{1}{15}$	$\frac{1}{14}$	$\frac{1}{35}$	$\frac{1}{27}$	$\frac{1}{35}$	$\frac{1}{27}$

Table 4.3: Interarrival parameters for the vehicle lanes.

We have no data to base these parameters on, so keep in mind that these values are just our best guesses of interarrival rates during rush hour. We only change the interarrival parameters; all other parameters are the same as in the preliminary experiment, see Table 4.2. The network architecture also stays the same. In Figure 4.3 we plot the average reward and average loss of each episode. Again, the agent seems to learn well during the exploration phase, as the average reward steadily increases until the exploration rate hits its minimum. After this point, the average reward stays stable throughout the remainder of the experiment, albeit more volatile than in the preliminary experiment. This, however, is expected as the flow of the traffic is more likely to be broken by other objects since all lanes and crosswalks are open. Furthermore, the average loss decays gradually until the exploration rate hits its minimum. Hence, both the average reward and the average loss behaves as well.



(a) Reward function

(b) Loss function

Figure 4.3: Average reward and loss.

As for the preliminary experiment, we plot which actions are taken in the 3 phases, to see if any obvious pattern is present, see Figure 4.4. From the figure, it is apparent that the shortest green duration is chosen the majority of the time for phases 1 and 2. However, in phase 3, although the shortest green duration is chosen the most, the longer durations are also chosen. The behavior in phases 1 and 2 might indicate that vehicles and pedestrians do not arrive often enough to justify longer green light durations, in which case it may make sense to allow even shorter green light durations, or entirely skip phases. However,

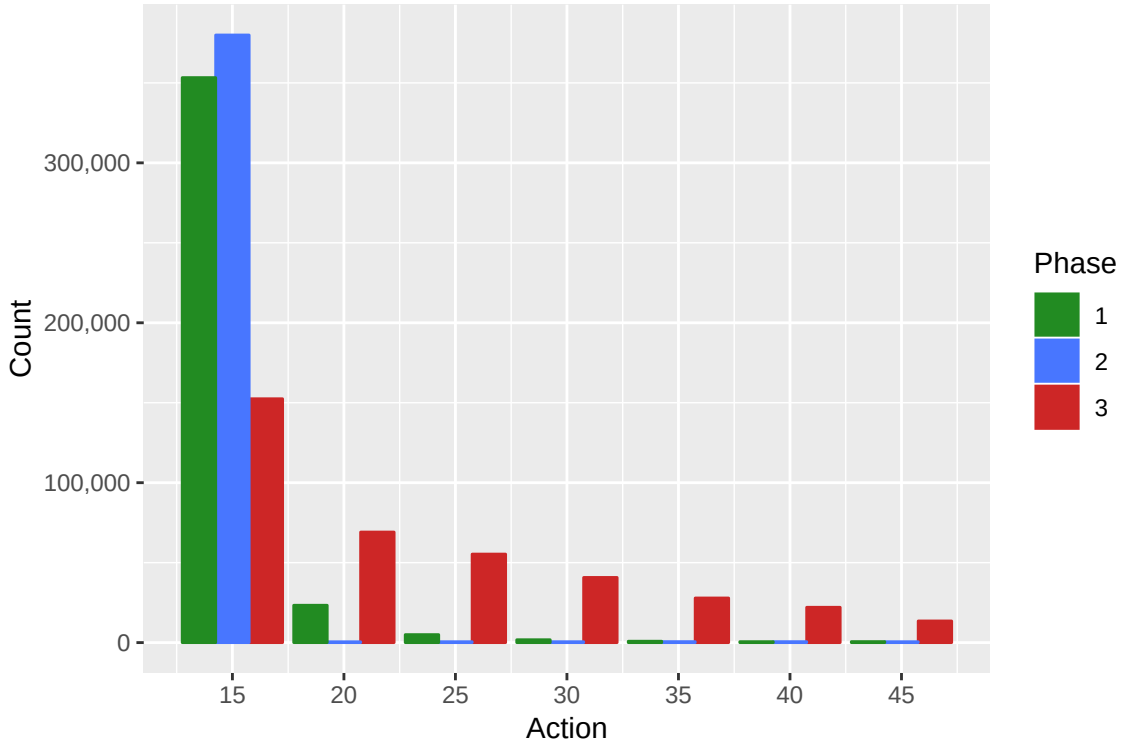


Figure 4.4: Actions taken for the three phases for all episodes after exploration rate reaches its minimum.

as mentioned earlier, more frequent light changes and no obvious light cycle may increase the chances of an accident occurring, hence we choose not to incorporate these changes.

4.3 Testing Against Benchmarks

In this section we compare the performance of the reinforcement learning algorithm to a fixed-time strategy and a heuristic strategy based on the idea of a police officer directing the traffic. We choose to compare the reinforcement learning algorithm to these strategies, as they are simple to implement and still used in practice.

Fixed-time Strategy

In the fixed-time control strategy, we fix the green duration in phases 1 and 3 to 30 seconds, and to 15 seconds in phase 2. We choose a different fixed time for phase 2 because phase 2 can be considered as a sort of "transition phase", since it consists of two lanes from phase 1 and two lanes from phase 3. Hence, the total green duration of e.g. lane 1 is the duration of phase 1 plus the duration of phase 2, and as such we deem that the duration of phase 2 does not need to be as long as phases 1 and 3.

Heuristic Strategy

In the police officer strategy, we construct a heuristic that only uses phases 1 and 3, as a police officer is unable to signal phase 2. As for the other strategies, we only allow the

heuristic to choose green durations between 15 and 45 seconds with increments of 5. The heuristic uses the following inequality to determine if it should change the phase:

$$v_{off} + p_{off} \geq v_{on} + p_{on} + c,$$

where v_{off} and p_{off} are the number of vehicles and pedestrians in the inactive phase, respectively, and v_{on} and p_{on} are the number of vehicles and pedestrians in the active phase. Furthermore c is a constant that controls how big the difference between the number of vehicles and pedestrians in the inactive and active phase should be before switching.

When a new phase begins, the heuristic waits for 15 seconds to pass, and then checks the inequality every 5 seconds. When the inequality is satisfied, or when the current phase has been active for 45 seconds, all lanes and crosswalks immediately get red light. The heuristic also changes the phase if there are no vehicles or pedestrians in the active phase while there are vehicles or pedestrians in the inactive phase waiting for green light. When the light turns red, the heuristic waits for all objects to leave the intersection, and the next phase begins.

Comparing the Strategies

To compare the three strategies, we simulate 1000 episodes using the same interarrival parameters as in Section 4.2 and run the strategies on these episodes. Note, for the DDDQN strategy, we do not train the network on these episodes. Instead, we simply use the already trained network to determine which action to take at each step. In Figure 4.5, we plot the average wait time per episode of the first 100 episodes for all three strategies. Furthermore, we also include the overall average wait time of all 1000 episodes for each strategy. We only plot 100 episodes as plotting more episodes makes it difficult to distinguish the strategies. In Appendix A, a plot of all 1000 episodes can be found.

From the figure, we observe that the DDDQN strategy obtains the lowest overall average wait time; 11.06% lower than the fixed-time strategy and 13.05% lower than the heuristic. We also observe that the DDDQN strategy appears to be more stable, as it has notably less fluctuation in the average wait time per episode than the other two strategies.

In Figure 4.6, we compare the three strategies in terms of the average vehicle wait time, average pedestrian wait time, and overall average wait time. The vehicle wait time follows the same pattern as the overall average wait time. However, the heuristic has a significantly lower average pedestrian wait time than the DDDQN and fixed-time strategies. Since pedestrians are unable to cross in phase 2, and the heuristic strategy only contains phases 1 and 3, it makes sense that the average pedestrian wait time is lower in this strategy than in strategies that also contains phase 2. Furthermore, this considerable difference in the average pedestrian wait time may indicate that it would make sense to allow other strategies to shorten phase 2, or possibly completely skip it.

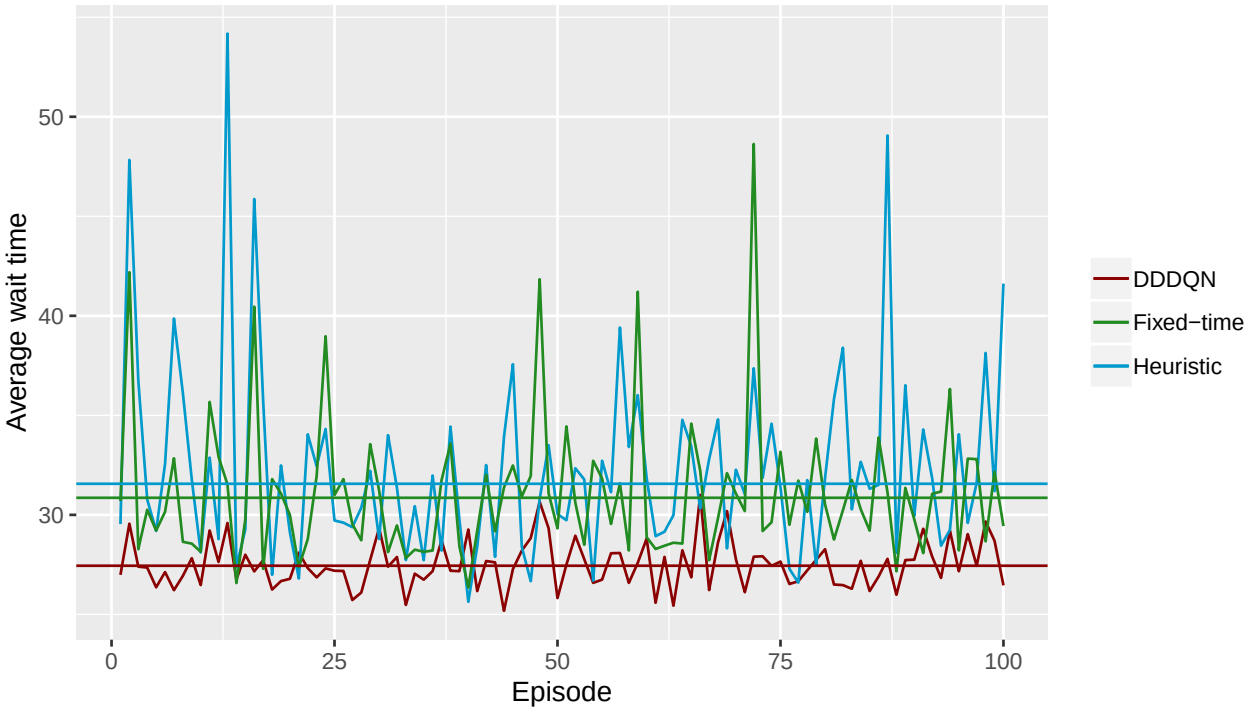


Figure 4.5: Comparison of the three strategies. The horizontal lines represent the overall average wait time of the same-colored strategy.

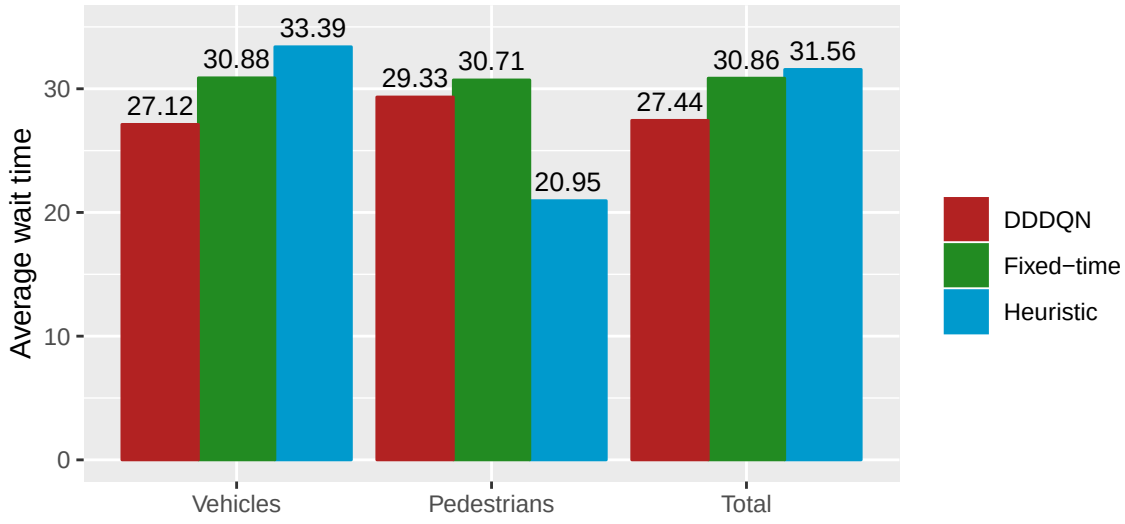


Figure 4.6: Average vehicle wait time, average pedestrian wait time, and overall average wait time of the three strategies.

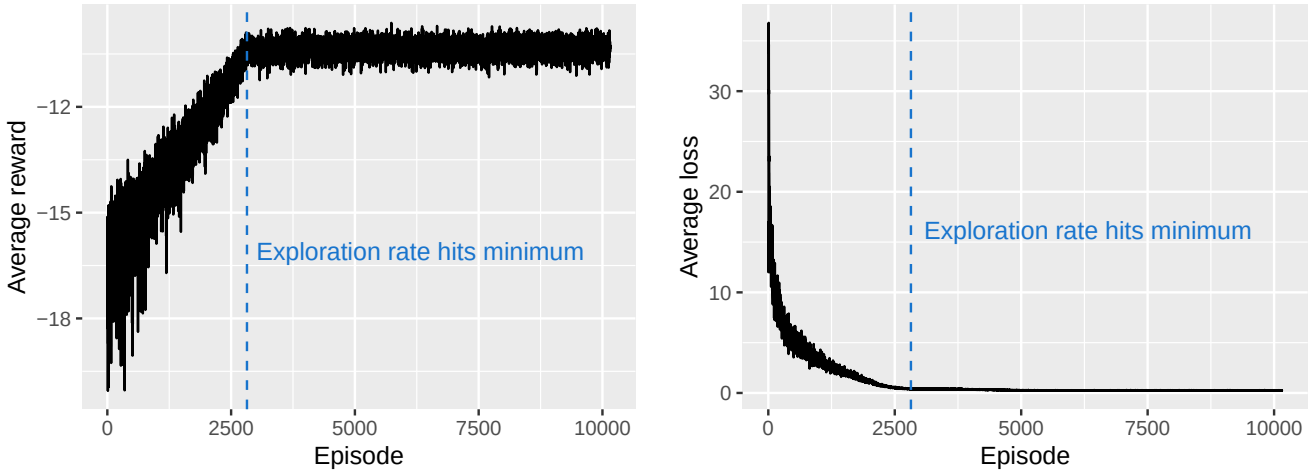
5

Impact of Excluding Pedestrians

In Chapter 4, we saw that the DDDQN algorithm learned well and was able to outperform the fixed-time strategy and the heuristic. In this chapter, we examine the impact of not including pedestrians during training. To this end, we first train a new DDDQN in a simulation environment where pedestrians do not arrive. We then test this DDDQN in an environment where pedestrians do arrive and compare it to the DDDQN from Section 4.2 that is trained with pedestrians.

5.1 Experiment Without Pedestrians

To train the DDDQN, we use the same parameters and network architecture as in Section 4.2. A plot of the average reward and loss, can be found in Figure 5.1. The average rewards seem less volatile and stabilizes at a higher value, compared to the experiment with pedestrians. The higher average reward is expected as pedestrians no longer halt the vehicles nor impact the reward function. Furthermore, without pedestrians the simulation has less randomness, which may cause the decrease in volatility. Looking at the average loss there is no notable change.



(a) Reward function

(b) Loss function

Figure 5.1: Average reward and loss.

The actions of the agent for each phase can be seen in Figure 5.2. The overall pattern is similar to that of the experiment with pedestrians in Section 4.2, however the shorter green durations are now even more favored. This change makes sense as there are no pedestrians arriving, and therefore the vehicles does not spend time waiting for the pedestrians to cross. As a result, shorter green light duration is enough to empty the queues.

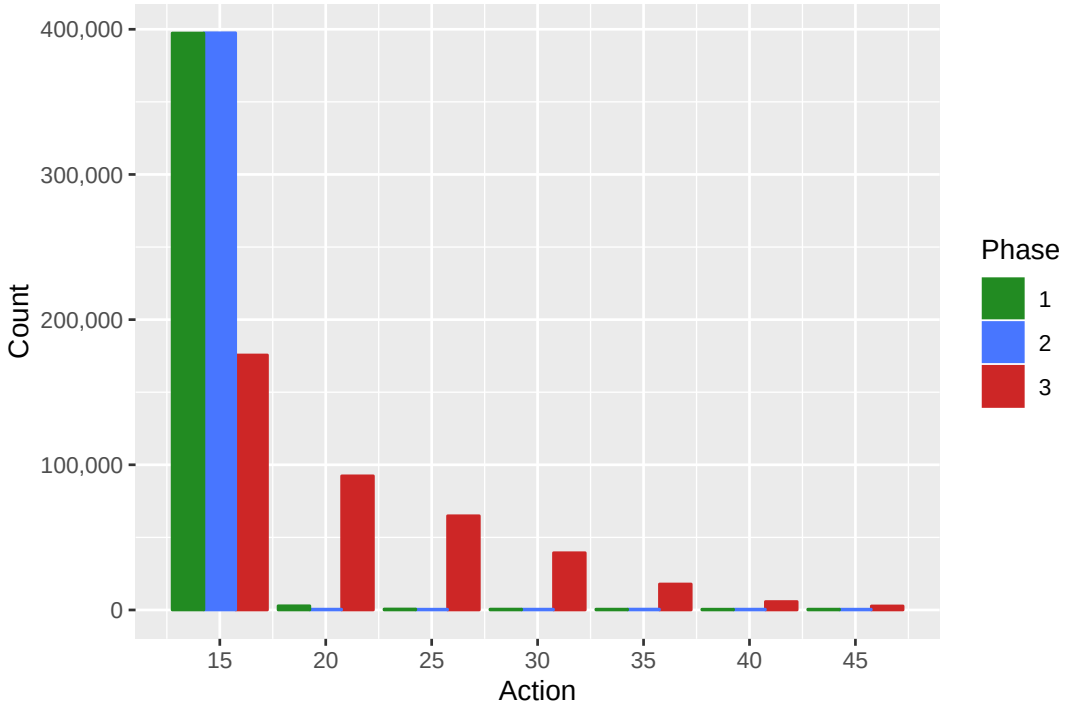


Figure 5.2: Actions taken for the three phases for all episodes after exploration rate reaches its minimum.

5.2 Comparing Dueling Double Deep Q-Networks

To get a better understanding of the importance of including pedestrians in the training, we compare the performance of the DDDQN model trained with pedestrians (the DDDQN from Section 4.2), and the DDDQN model trained without pedestrians (the DDDQN from Section 5.1). To this end, we simulate 1000 episodes with the same interarrival parameters as in Section 4.2 and compare the performance of the DDDQNs on these episodes. In Figure 5.3, we plot the average wait time per episode of both DDDQNs for the first 100 episodes, as well as the overall average wait time of all 1000 episodes for both DDDQNs. In Appendix A, a plot of all 1000 episodes can be found. From the figures we find that the DDDQN trained with pedestrians has the lowest overall average wait time; in fact it is 1.90% lower than that of the DDDQN trained without pedestrians. Thus, including pedestrians in the simulation when training the DDDQN results in a slightly lower average wait time of vehicles and pedestrians.

We also compare the actions of the two DDDQNs to examine how they behave in more details, see Figure 5.4. From the figure we see that the actions of the two DDDQNs are fairly similar for phase 1 and 2. However, for phase 3 their actions differ quite a bit.

These results imply that traffic control agents trained without considering pedestrians might perform worse than expected in a real setting where pedestrians can cross the intersection. However, the argument could be made that because we use the hyperparameters of the DDDQN with pedestrians to also train the DDDQN without pedestrians, the DDDQN without pedestrians finds a suboptimal policy. Therefore, proper tuning of the hyperparameters is highly relevant and more work to obtain better values

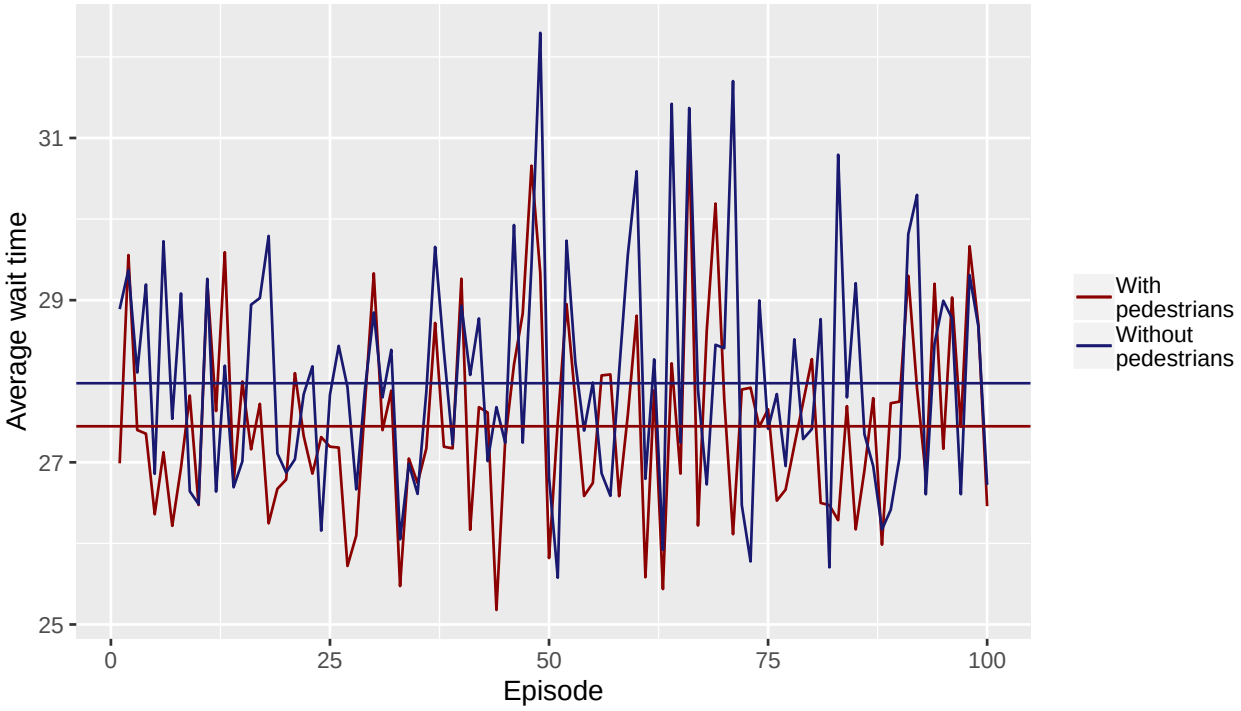


Figure 5.3: Average wait time of a DDDQN trained with pedestrians and a DDDQN trained without pedestrians.

of the hyperparameters can be done. However, due to time limitation we are unable to investigate this further.

As a passing remark, we notice that the actions of the DDDQN trained with pedestrians in Figure 5.4 differ from the actions of the same network during training in Figure 4.4. Despite our best efforts, we are unable to determine the cause of this difference, but we suspect that it is because that, even though the exploration rate has reached its minimum, the agent in Figure 4.4 still explores 1% of the time. Choosing random actions can sometimes lead the agent into situations where it has to choose suboptimal actions to recover, compared to when it is allowed to choose the best action all the time.

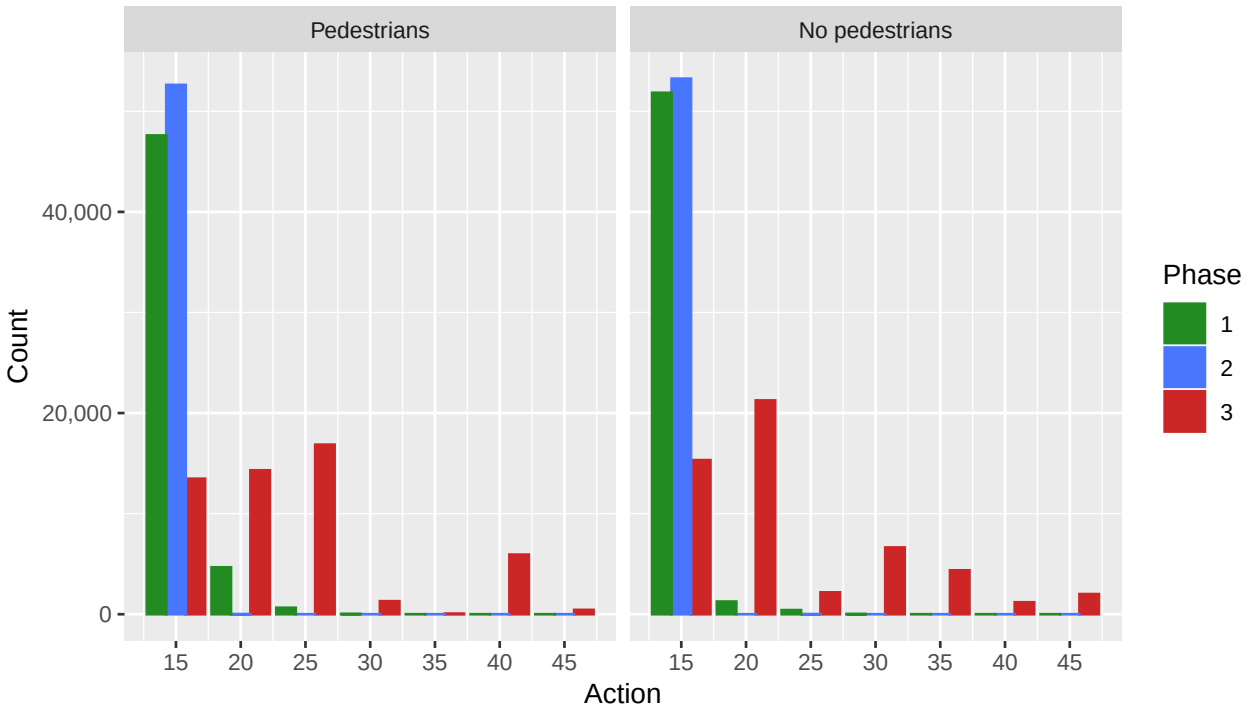


Figure 5.4: Actions of the DDDQN trained with pedestrians (left) and trained without pedestrians (right)

6

Discussion

In this thesis we have made experiments without having any data to help estimate essential values related to the simulation. These values include the various crossing times of vehicles, explained in Section 3.1, the walking speed of pedestrians, and interarrival times of vehicles and pedestrians. Under normal circumstances, we would have been able to gather data by observing the intersection, however the global pandemic made this a risk-filled endeavor. Furthermore, since most people have been encouraged to stay home, there is little traffic, making it impossible to estimate interarrival parameters fitting of rush hour. Thus, we have instead chosen these values using our best guess, and as such, the results and policies found in this thesis may be different if the proposed framework is applied on actual data. However, the main conclusions are likely to stay the same, as we believe more realistic simulation parameters will only shift the numeric results of the algorithms.

DDDQN and PER are relatively new methods to use in traffic control as well as in general. Hence, there is no formal/common approach to obtaining good values of the hyperparameters, such as the size of the memory or the number of and sizes of the hidden layers. We have relied on the parameter values of other practitioners in the field that obtain good results, and used their parameter values as starting values. We have then used trial and error to obtain parameter values that yield good results in our problem setting. As research in the field continues, more general ways of determining fitting values may be found, which might result in better performance.

As layout of the intersection, we chose a specific intersection located in Aalborg. The layout is somewhat unique and generalizing our results and policies is therefore problematic, compared to if a more generic layout had been used. However, we chose a local intersection as it would have made data gathering possible under normal circumstances.

7 Conclusion

In this thesis, we investigated how the behavior of a dueling double deep Q-network reinforcement learning algorithm with prioritized experience replay changes when including pedestrians in an isolated intersection traffic light control problem during rush hour, compared to excluding pedestrians. We also examined whether our reinforcement learning algorithm is able to outperform simpler traffic light control algorithms. To this end, we created a simulation environment that includes pedestrians, based on a local intersection in Aalborg, Denmark, and implemented and trained a dueling double deep Q-network used to determine the green light duration of each traffic phase.

In Section 4.3, we found that the reinforcement learning algorithm was able to outperform both a fixed-time algorithm and a heuristic based on the idea of a police officer signaling the traffic, using the average wait time per vehicle/pedestrian as the performance metric. The reinforcement learning algorithm outperformed the fixed-time algorithm by 11.06% and the heuristic by 13.05%

In Section 5.1, we trained another dueling double deep Q-network in a simulation environment where no pedestrians arrives but otherwise with the same parameters. We compared the actions and performance of this network with the original network that was trained in a simulation environment where pedestrians did arrive. We found that the actions of the network trained without pedestrians differs from the actions of the network trained with pedestrians. By comparing the performance of the two networks in a simulation environment with pedestrians, see Section 5.2, we found that the network trained with pedestrians outperformed the network trained without pedestrians by 1.90% when using the average wait time as performance metric. This result implies that if the reinforcement learning agent is meant to control the traffic light of an intersection in which pedestrians are able to cross, it should be trained in such an environment, otherwise suboptimal performance is likely to be obtained.

7.1 Future Research

In this thesis we use the increment in average time spent in the intersection between two successive states as the reward function. Since the policy found by a reinforcement learning agent heavily relies on the reward function, investigating other reward functions could lead to better performance of the agent.

In the reinforcement learning algorithm we use the ε-greedy algorithm to determine whether to take a random action or let the agent exploit what it has learned. The ε-greedy algorithm is fairly naive, and investigating how a more sophisticated algorithm, such as Thompson sampling, would affect the performance of the reinforcement learning algorithm could be interesting.

Expanding the action space is also a point of interest. Throughout the experiments we found that the algorithm favors the shortest duration, implying that allowing for even shorter durations, or completely skipping a phase, could lead to better performance.

Furthermore, the current green light durations are in increments of 5 seconds as we wanted to keep the action space small to ensure fast and steady learning. However, the algorithm might be able to handle an action space where the actions are in increments of 1 second instead, while still being able to learn in a reasonable time.

While some effort has been put into parameter tuning, we believe better parameter values can still be found. Furthermore, while we have used some of the newer methods in reinforcement learning, there are still some extensions to the DDDQN that may improve performance, such as noisy nets [Fortunato et al., 2017], n-step Q-learning [Mnih et al., 2016], and distributional reinforcement learning [Bellemare et al., 2017].

To update the network weights, we used the ADAM optimizer, because of its speed and popularity in the deep learning field. However, an extension of ADAM called stochastic variance reduction for deep Q-learning (SVR-DQN) has recently been proposed by [Zhao et al., 2019]. This method incorporates techniques from the stochastic variance reduced gradient method and supposedly obtains better results, while still retaining some of the speed of ADAM. Thus, testing the SVR-DQN in our setting could help improve performance.

A

Comparison

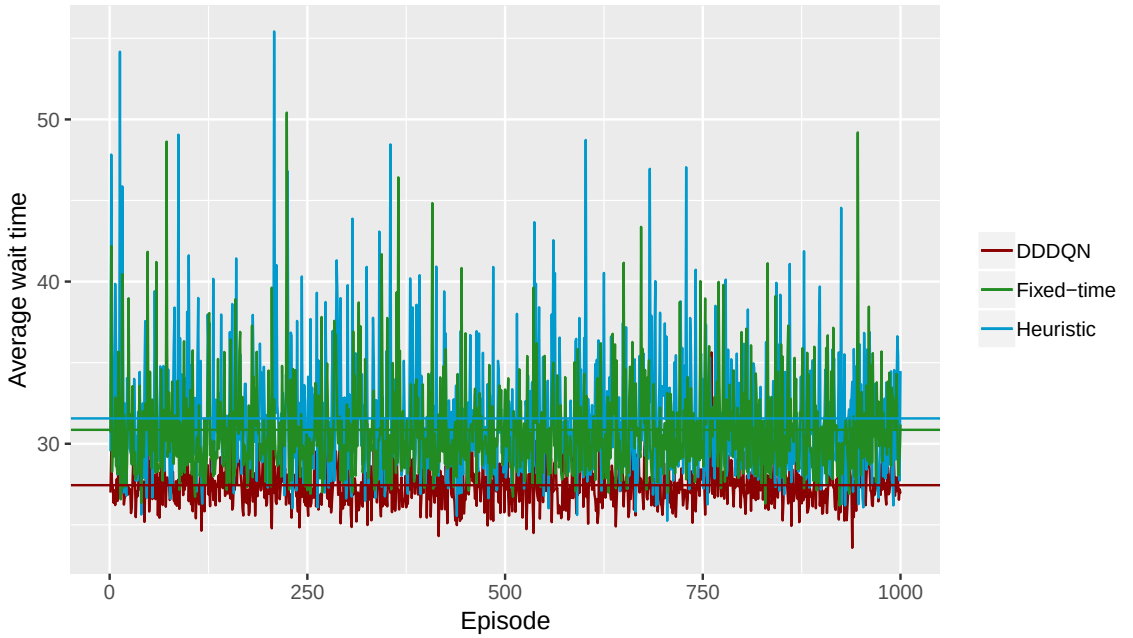


Figure A.1: Average time spent in intersection of the DDDQN, the fixed-time strategy and the heuristic.

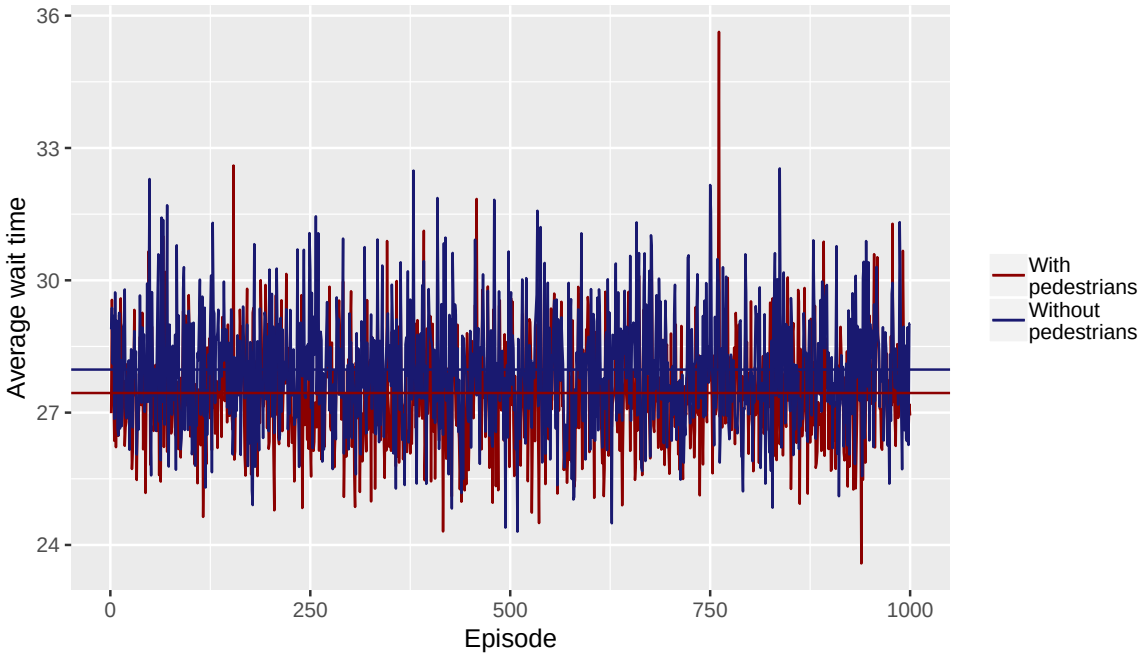


Figure A.2: Average time spent in intersection of a DDDQN trained with pedestrians and a DDDQN trained without pedestrians.

Bibliography

Abdulhai et al., **2003**. Baher Abdulhai, Rob Pringle and Grigoris J. Karakoulas. *Reinforcement Learning for True Adaptive Traffic Signal Control.* Journal of Transportation Engineering, Volume 129 Issue 3 - May 2003, 2003. URL https://www.researchgate.net/publication/228593853_Reinforcement_Learning_for_True_Adaptive_Traffic_Signal_Control.

Bellemare et al., **2017**. Marc G. Bellemare, Will Dabney and Rémi Munos. *A Distributional Perspective on Reinforcement Learning.* CoRR, abs/1707.06887, 2017. URL http://arxiv.org/abs/1707.06887.

Christidis and Rivas, **2012**. Panayotis Christidis and Juan Nicolás Ibáñez Rivas. *Measuring road congestion*, European Commission, 2012. URL https://publications.jrc.ec.europa.eu/repository/bitstream/JRC69961/congestion%20report%20final.pdf.

Fortunato et al., **2017**. Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell and Shane Legg. *Noisy Networks for Exploration.* CoRR, abs/1706.10295, 2017. URL http://arxiv.org/abs/1706.10295.

Gao et al., **2017**. Juntao Gao, Yulong Shen, Jia Liu, Minoru Ito and Norio Shiratori. *Adaptive Traffic Signal Control: Deep Reinforcement Learning Algorithm with Experience Replay and Target Network*, 2017. URL https://arxiv.org/abs/1705.02755.

Genders and Razavi, **2016**. Wade Genders and Saiedeh Razavi. *Using a Deep Reinforcement Learning Agent for Traffic Signal Control*, 2016. URL https://arxiv.org/pdf/1611.01142.pdf.

Goodfellow et al., **2016**. Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning.* The MIT Press, 2016. ISBN 0262035618.

Kingma and Ba, **2014**. Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*, 2014. URL https://arxiv.org/abs/1412.6980.

Kwong et al., **06 2011**. Yit Kwong, Nurmin Bolong, Aroland Kiring, Soo Yang, Kenneth Tze and Kin Teo. *Q-Learning Based Traffic Optimization in Management of Signal Timing Plan.* International Journal of Simulation: Systems, Science and Technology, 12, 2011. URL https://www.researchgate.net/publication/264889551_Q-Learning_Based_Traffic_Optimization_in_Management_of_Signal_Timing_Plan.

LA and Bhatnagar, **2011**. P. LA and S. Bhatnagar. *Reinforcement Learning With Function Approximation for Traffic Signal Control.* IEEE Transactions on Intelligent Transportation Systems, 12(2), 412–421, 2011. URL https://www.researchgate.net/publication/224203001_Reinforcement_Learning_With_Function_Approximation_for_Traffic_Signal_Control.

Liang et al., **2018**. Xiaoyuan Liang, Xunsheng Du, Guiling Wang and Zhu Han. *Deep Reinforcement Learning for Traffic Light Control in Vehicular Networks*. CoRR, abs/1803.11115, 2018. URL `http://arxiv.org/abs/1803.11115`.

Lin, **1992**. Long-Ji Lin. *Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching*. Machine Learning, 8, 293-321, 1992. URL `https://link.springer.com/content/pdf/10.1007/BF00992699.pdf`.

Mannion et al., **05 2016**. Patrick Mannion, Jim Duggan and Enda Howley. *An Experimental Review of Reinforcement Learning Algorithms for Adaptive Traffic Signal Control*. pages 47–66, 2016. doi: 10.1007/978-3-319-25808-9_4.

Mnih et al., **2016**. Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver and Koray Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning*. page 1928–1937, 2016.

Pol, **2016**. Elise Van Der Pol. *Deep Reinforcement Learning for Coordination in Traffic Light Control*, 2016. URL `http://www.elisevanderpol.nl/papers/vanderpol_mscthesis.pdf`.

Ruder, **2016**. Sebastian Ruder. *An overview of gradient descent optimization algorithms*, 2016. URL `https://arxiv.org/abs/1609.04747`.

Schaul et al., **2015**. Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. *Prioritized Experience Replay*, 2015. URL `https://arxiv.org/abs/1511.05952`.

Sutton and Barto, **2015**. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2015.

van Hasselt et al., **2015**. Hado van Hasselt, Arthur Guez and David Silver. *Deep Reinforcement Learning with Double Q-learning*, 2015. URL `https://arxiv.org/abs/1509.06461`.

Wang et al., **2015**. Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot and Nando de Freitas. *Dueling Network Architectures for Deep Reinforcement Learning*, 2015. URL `https://arxiv.org/abs/1511.06581`.

Watkins and Dayan, **05 1992**. Christopher Watkins and Peter Dayan. *Technical Note: Q-Learning*. Machine Learning, 8, 279–292, 1992. doi: 10.1007/BF00992698. URL `https://www.researchgate.net/publication/220344150_Technical_Note_Q-Learning`.

Xie et al., **2014**. X. Xie, S. F. Smith, T. Chen and G. J. Barlow. *Real-time traffic control for sustainable urban living*. pages 1863–1868, 2014.

Yin and Menendez, **07 2019**. Biao Yin and Monica Menendez. *A Reinforcement Learning Method for Traffic Signal Control at an Isolated Intersection with Pedestrian Flows*. pages 3123–3135, 2019. doi: 10.1061/9780784482292.270.

Zhao et al., **2019**. Wei-Ye Zhao, Xi-Ya Guan, Yang Liu, Xiaoming Zhao and Jian Peng. *Stochastic Variance Reduction for Deep Q-learning*, 2019. URL `https://arxiv.org/abs/1905.08152`.