Norges miljø- og biovitenskapelige universitet

**Master's Thesis 2021 30 ECTS**
Faculty of Science and Technology

# Traffic Control with the use of Multi-Agent Deep Reinforcement Learning

Mathias Bergane Johansen
Data Science

# Abstract

In the modern society, traffic is a heated topic in everyday conversations and economics. As more and more traffic lights are given real-time access to traffic data from the intersections they are regulating, there is a potential for using multi-agent reinforcement learning to optimize traffic flow. In many cities, numerous intersections are often located close together and influencing each other in a complex pattern. By linking the intersections together for joint information processing based on deep artificial neural networks one can search for a solution that regulates the traffic flow in an optimized fashion. The approach chosen for our study is known as multi-agent deep reinforcement learning. For short we will just use the term multi-agent reinforcement learning.

In the present thesis we generated two different models where the information shared between the intersections is different. The models were tested against a static model representing the classical way of controlling the intersections. From our results, we conclude that multi-agent reinforcement learning represents an interesting potential for improving the traffic flow in a network of intersections. This is especially the case when information is shared between the agents controlling the traffic lights at the various intersections. As this field of study advances, it is likely to expect that even better solutions requiring less computational power may be achieved.

# Acknowledgments

Thanks to my advisor Ulf Indahl and my professors Oliver Tomic and Kristian Liland for all their help during this project, especially Ulf for his help with the structure of the thesis.

# Contents

# List of Figures

# Word List

**Terminology:**

*State* - Information about the environment now.
*Action* - One agent's action on the environment(traffic lights).
*Agent* - Name of the model which is trained in a reinforcement learning case-
*Multi-agent* - Multiple agents combined to solve one problem. Will also be referred to as one model.
*Episode* - An whole simulation from start to termination.
*Simulation time* - Number of time steps in the simulation. One time step is the same as one secound.

# Abbreviations and paremeter descriptions

NN - Neural network
ANN - Artifical neural network
DNN - Deep neral network
RL - Reinforcement learning
DQL - Deep Q-learning
MLP - Multilayer perceptron
CNN - Convolutional neural network
RNN - Recurrent neural network
TD - Temporal-difference
DRL - Deep reinforcement learning
DDQN - Double deep Q-network
MARL - Multi-agent reinforcement learning
GUI - Graphical user interface
ts - Time step
$x$ - input data
$y$ - output date
$t$ - time unit

# Introduction

It is well known that traffic jams cause problems in most cities. Not only are traffic jams a problem for the normal person traveling to work. Congestion and delays are also major problems for those who are dependent on the transportation of goods and services. Such problems do not only increase the costs of transportation but make it difficult to schedule services, deliveries, meetings, and so on. In an article based on research from the transport data company INRIX published in the World Economic Forum in March 2019, it is stated that nearly 87 billion USD was lost due to traffic congestion[3]. Clearly, it will be beneficial to try to reduce the impact of congestion, not only for the time saved by the traveling individuals, but also for the economic gain of the entire society.

Given that the control of traffic lights depends on the time of day and the day itself, the idea of a traffic regulation model being trained based on simulations may lead to a flexible solution. Machine learning applications have become more and more common in various fields due to the increase in available computational power, and the possibility of simulating complex environments millions of times is no longer unusual.

In this thesis, we consider a situation where an agent is trained to control multiple traffic lights. The training is based on trial and error, much similar to the way human children (and humans in general) learn. For making this practically possible, a formal operational description of a traffic environment is required, and the agent must be able to receive and act on data from this environment.

To assess the performance of the trained agent, it must be tested against realistic situations that have not been included in the training process. For this purpose, we have designed an environment that mimics multiple connected intersections in Oslo regulated by traffic lights. Obviously, Oslo is the Norwegian city that may have the biggest benefit of improving the traffic regulation. In 2019, Oslo alone had 32% of the population growth in Norway[4], meaning that the traffic challenges most likely will continue to increase in the future.

# Chapter 1

# Theoretical background

Here, we will provide the basic theory needed for developing an agent capable of using deep Q-learning (DQL) to control multiple traffic lights. Both supervised learning and reinforcement learning (RL) which are two of the main pillars of modern machine learning will therefore be presented. Although our proposed solution is based on RL, the understanding of supervised learning in the context of artificial neural networks (ANN) plays a major part in understanding DQL.

## 1.1 Supervised learning

The field of machine learning (ML) has three main learning pillars: Reinforcement-, supervised- and unsupervised learning, and the various methodologies are influenced by computer science as well as statistics and mathematics.

In supervised learning, the agent is trained on labelled data where the agent is presented with a collection input data ($x$) and corresponding output data ($y$). The goal is then for the agent to develop a model $\hat{f}$ representing some unknown relationship $f$ between $x$ and $y$, so that $y = f(x) + \text{noise}$. A good $f$-approximation $\hat{f}$ should not only be able to predict both for the labelled $(x, y)$-data available to the training process, but also for new data not available in the training process.

### 1.1.1 Artificial neural networks and the perceptron

The dynamics of RL are inspired by the way humans learn. The idea of artificial neural networks is based upon a simplified view on the architecture of the human brain and how the brain works in problem-solving. ANN's are proven to be a good suit for approximating highly non-linear functions. The idea was introduced in the 1940s when Warren McCulloch and Walter Pitts described how neurons could work and interact[5].

An ANN is obtained by linking together multiple artificial neurons in a computational network. One of the first concepts of this idea is Rosenblatt's perceptron(1950s)[6]. The perceptron is constructed by computing a weighted (by the weights $w_1, \ , w_2, ..., w_m$) linear combination of the input values $x = (x_1, \ x_2, ..., x_m)$ including a bias term ($b$). The result of this is the net output

$$z = b + w_1x_1 + w_2x_2 + ... + w_mx_m.$$

The perceptron is mostly used in binary classification problems $(-1, +1)$, meaning that the net output $z$ is finally compared to a threshold value $\theta$ so that class 1 is predicted if $z \geq \theta$ and class $-1$ is predicted otherwise.

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise.} \end{cases}$$

*Figure 1.1: Output of a perceptron based on the threshold $\theta$ [1, Figure from Ch. 2].*

An illustration of the perceptron's functionality is shown in figure 1.2. In this illustration, $w_0$ corresponds to the bias $b$. As indicated, the errors based on the perceptron outputs are used to update the weights during the training process. The later idea of combining several layers of perceptrons into networks of interconnected perceptrons later resulted in the creation of ANN.

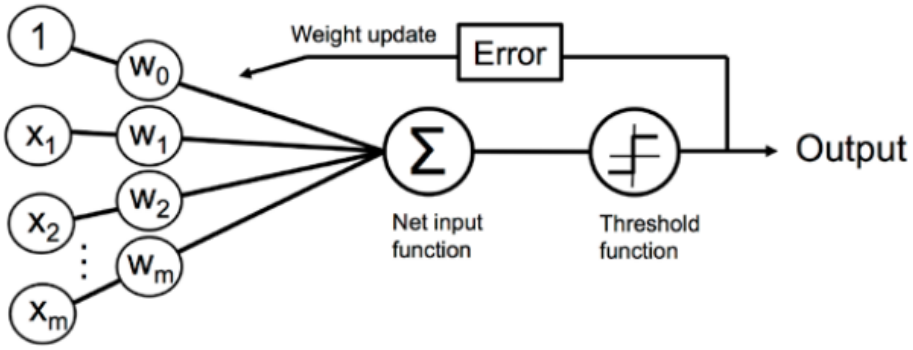



*Figure 1.2: Illustration of the perceptron [1, Figure from Ch. 2].*

The combination of perceptrons into a multilayer ANN is shown in figure 1.4. Depending on the number of combined perceptrons, the number of weights in this model is considerablylarger, i.e. the number of weights represented in the weight matrix $\mathbf{w}^h$ is $(m+1) * d$. By assuming that the nodes in each layer are fully connected to each node in the subsequent layer, we have that all input values are connected to every node in the first hidden layer and so on. This is architecture is often referred to as feed-forward. All networks with more than one hidden layer are regarded as deep artificial neural networks (DNN's). Figure 1.4 shows an output layer with $t$ number of cells, meaning that the MLP is set up to classify the input-data into $t$ different classes.

ANNs with one or more hidden layers using nonlinear activation functions $a(z)$ are capable of representing complex nonlinear relationships between the $x$ and $y$ data.

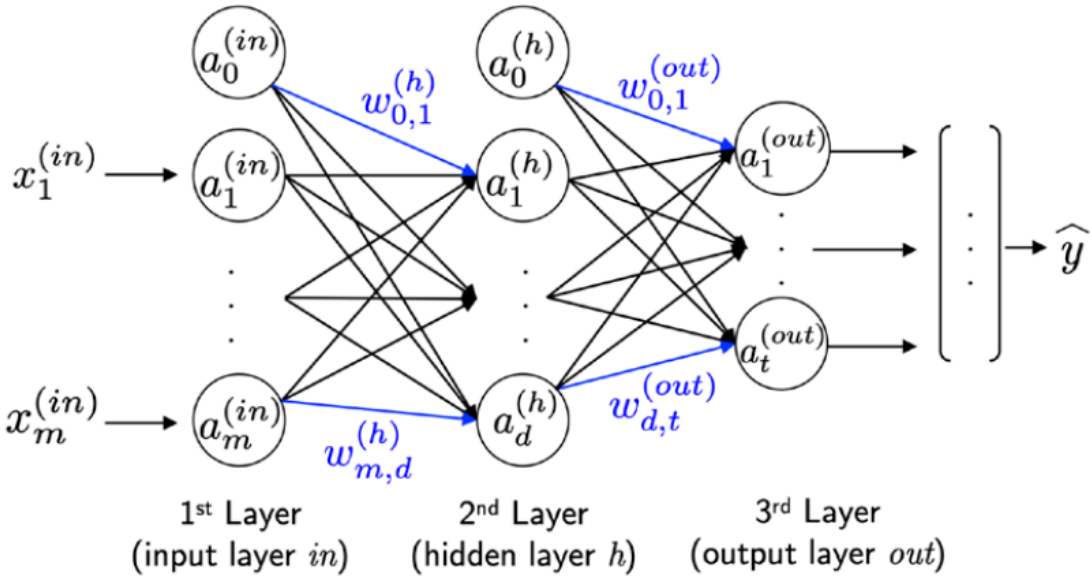



*Figure 1.3: Illustration of a MLP [1, Figure from Ch. 12].*

### 1.1.2 Deep learning

DNN modeling has received much attention in recent years, many thanks to the drastic growth in hardware developments and computational power. This new extended field of research inside ML is often referred to as deep learning(DL).

The main difference between the more traditional ML modeling and DL is the ability to represent more complex nonlinear functional relationships for the raw input data. In classical ML, hand-crafted feature engineering is often required in preparation of the training data before successfully training a classifier. This is not needed to the same extent in DL, where feature engineering can be considered as an integrated part of the training process. One negative side of DL is that it usually requires training on rather large datasets to perform well.

Besides the basic feed-forward ANN architectures with multiple hidden layers (essentially the DNNs), there are several alternative ways of structuring a network architecture. The convolutional neural networks (CNN's)[1, Ch. 15] represent a family of networks highly popular for image classification problems. The CNN's implement convolutional layers replacing the dense layers in the DNN to evolve convolutional filters representing patterns and shapes appropriate for the images subject to classification. There are also the so-called recurrent neural networks (RNN's)[1, Ch. 16] appropriate for classifying different length sequences of inputs. Briefly explained, RNN's are capable of learning about the early parts of sequences to predict the final (missing) parts as outputs. An important application is the problem of translating sequences of audio or speech provided as inputs into sequences of audio or speech in another language as output[7].

### Cost function

Training of a DNN requires optimization of a cost function $J(\mathbf{W})$ associated with the network architecture, its weight parameters ($\mathbf{W}$) and associated activation functions.

$$J(\boldsymbol{W}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]} \log\left(a_j^{[i]}\right) + \left(1 - y_j^{[i]}\right) \log\left(1 - a_j^{[i]}\right) \tag{1.1}$$

This function takes in $a^{[i]}$ which is the $i$th sample in the dataset transformed by the activation function(sigmoid, ReLU, etc) and $y^{[i]}$ that is the associated output from the dataset in index $i$. For this equation $t$ represents the number of units in the layer,

### Optimization by backpropagation

The goal is then to minimize the cost function. This can be done by finding the derivative of $J(\mathbf{W})$ with respect to each weight in every layer:

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\boldsymbol{W}) \tag{1.2}$$

This is usually obtained by the use of Backpropagation[8] that is one of the most used ways to train a neural network. To be clear, loss and cost functions are not the same but closely related: The cost function is an average of all loss or error while the loss itself is calculated for each layer and used when updating each weight. Under is a figure explaining the steps:

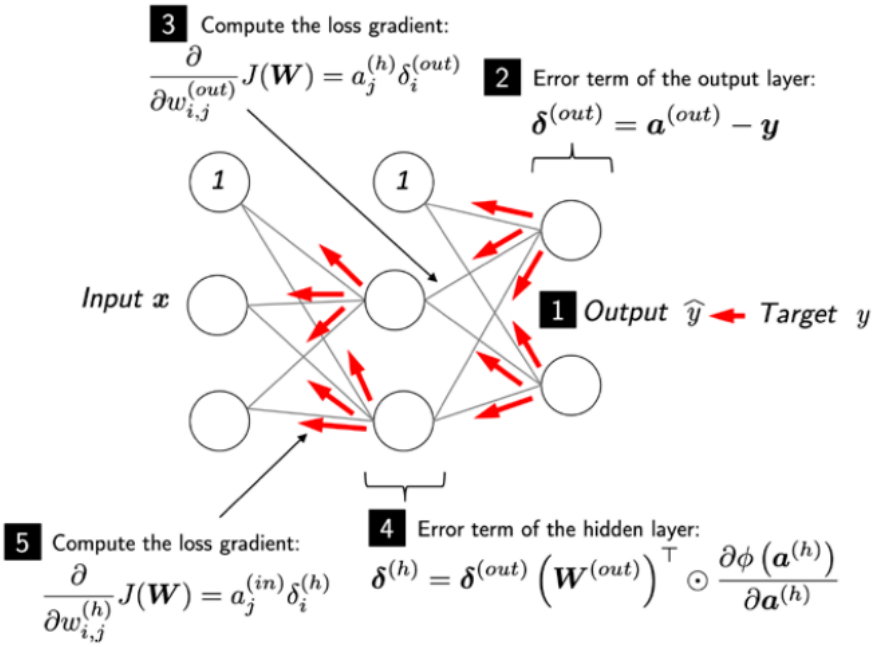



*Figure 1.4: Backpropagation explaining for a three layered neural network [1, Figure from Ch. 12].*

The backpropagation starts when the forward-propagation is done and $\hat{y}$ is calculated (1). Following is the calculation of the error of the output layer. This is the difference between the predicted value and the true value $y$ (2). $\boldsymbol{\delta}^{out}$ is then used together with the activation $\boldsymbol{a}^{(out-1)}$ of the previous layer to compute the loss gradient or the derivative of the cost function with respect to a specific weight $w$ (3). Next, calculate the error term of the hidden layer (out-1). Here the previous error term, the weight matrix used in the connections between this and the earlier layer, and the derivative of the activation function are used (4). Further, process (3) is repeated but now concerning the new error term and earlier activations (5). If the network had several layers, this process could just repeat.

So now that we have the loss gradient for each connection between each node in every layer we can update the weights. Under is the formula for updating the value of one weight. The amount of adjustment to the weight is based on the learning rate $\alpha$. So the number of weight updated in layer $l$ is $i * (j + 1)$, when including a bias every node.

$$w_{i,j}^{l} = w_{i,j}^{l} - \alpha \frac{\partial J(\boldsymbol{W})}{\partial w_{i,j}^{l}}$$

When training a NN this process of forward- and backpropagation is repeated over and over to the model starts to learn the relations between the input data and the output. In this theory, regularization is not taken into account.

## 1.2 Reinforcement learning

As mentioned earlier, reinforcement learning o(RL) is considered to be one of the main pillars of ML. Compared to supervised learning which trains on labelled data, RL aims to train an agent based on the agent's actions in some environment, and the resulting feedbacks from the actions taken. This idea can be compared to how human children learns by trial and error. The following theoretical description of reinforcement learning is influenced by Richard Sutton and Andrew Barto's book: "Reinforcement learning: An introduction"[2]

A helpful illustration of how the agent interacts together with the environment is shown in figure 1.5 below. Here we see how the agent first gets in the state $S_t$ and reward $R_t$, then sends an action $A_t$ to the environment. The environment again responds with a new state $S_{t+1}$ and reward $R_{t+1}$. This process continues until the environment sends information that the episode is terminated. This interaction is a sequence of discrete time steps, $t = 0, 1, 2, ..., n$, and n is the length of the sequence called an episode.

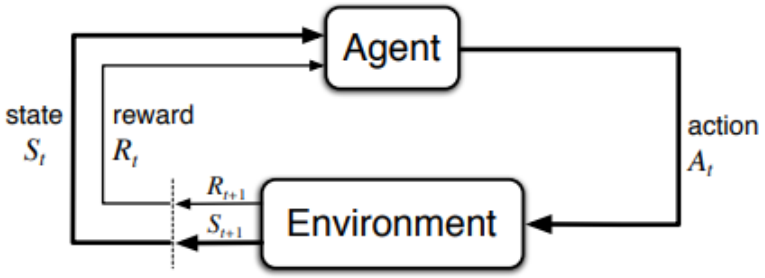



*Figure 1.5: The interaction between an agent and an environment [2, Figure from Ch. 3.1]*

A trained agent takes actions based on its expected reward for that action in a specific state. This means that the immediate reward is not used alone, but in combination with the possible future reward for that action. An example of this is an action that gives a bad immediate reward, but it is exactly this action that lays the groundwork for a big future reward later. This is important since the goal of the agent is to maximize the total reward over the entire episode.

### 1.2.1 Markov decision processes

The interaction between the agent and the environment needs to be represented in a manageable way, and it is here the Markov decision processes comes into play. Markov decision processes or MDPs is a discrete-time stochastic control process and provides a framework for modeling decision making.

An MDP is a 4-tuple of ($S$, $A_s$, $P_a$, $R_a$), where it can be describes as following:

- $S$ is the state space containing every posible state in the environment.
- $A_s$ is all the action available in state $s$. $A$ is called the action space and is a set of all the actions.

- $P_a(s, s')$ represents the probability that action $a$ in state $s$ will lead to state $s'$ in the next state.
- $R_a(s, s')$ represents the immediate expected reward by moving from state $s$ to state $s'$ as a consequence of taking action $a$.

In statistics, a stochastic process or episode which only depends on the present state and noneof its past transitions is often called Markovian or a Markov process. This property is satisfied if we look at $P_a(s, s')$ where the probability for the next state $s'$ is only dependent on the current state $s$ and action $a$.

A Markov decision process has the goal of finding a good "policy" ($\pi$) for the agent, where $\pi : S \rightarrow A$ is a function that maps the states $s \in S$ to available actions $a \in A_s \subseteq A$ so that the expected reward (see below), over a finite or potentially infinite number of steps, is maximized.

### 1.2.2 Reward

So the feedback or reward $r_t$ given to the agent from the environment is often based on what you want to achieve with the agent. If the goal is to win a game of chess, then the feedback may be positive if the agent takes a chess piece from the opponent and negative if it loses one. Reward functions can also be depending on time, so let's say you want to control a car, but the time it takes from a to b is also important. Then for each timestep, a negative reward can be given together with other rewards so the agent is not encouraged to drive slow.

The goal of the agent is to maximize the total reward obtained over the whole episode which has a finite number of steps. The sum of all the rewards from t to the length of the episode T is defined as follows:

$$G_t = r_{t+1} + r_{t+2} + ... + r_T$$

This function can be used to track the performance of the agent but is not often used during the training of the agent. Instead, a function that discounts the future reward based on a factor $\gamma$ is used:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-1} r_T$$

where $0 \leq \gamma < 1$ Gamma or $\gamma$ says something about how significant the future rewards are relative to the immediate rewards. Here $\gamma = 0$ indicates that the future rewards are irrelevant, and the agent will only focus on obtaining the biggest reward in the current simulation step. This may lead to short-term solutions which may not work that well in general. As $\gamma$ gets closer to 1 the agent takes the future rewards more into account, which are more likely to lead to better results. Both small and large values of $\gamma$ may lead to good policies for an agent depending on the nature of its tasks.

### 1.2.3 Temporal-Difference Learning and Q-learning

In this work, the algorithm known as Q-learning will be used. This algorithm comes from an idea in RL called temporal-difference (TD) learning and is a combination of the ideas in dynamic programming (DP) and Monte Carlo methods[1][From Ch. 6]. We will not go too deep into the theory of Monte Carlo methods but will explain the difference between TD and Monte Carlo later.

The Value function $V(S)$ or Bellman equation gives the value of a state and is updated with this equation:[1][From Ch. 6.1]

$$V^{new}(s_t) = V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Here $\alpha$ is the learning rate, $\gamma V(s_{t+1})$ is the discounted value of the next state, $R_{t+1}$ is the reward by going from $s_t$ to $s_{t+1}$ and $V(s_t)$ is the value of the current state. This update is only dependent on the next state $s_{t+1}$ and its discounted value. The information from the next state is an estimate which is used in the update, and these kinds of methods are called bootstrapping methods. It is this idea that comes from dynamic programming. Note that not all TD methods are the same, and this one is called one-step TD or TD(0).

The Value function for an every-visit Monte Carlo method is:

$$V^{new}(s_t) = V(s_t) + \alpha[G_t - V(s_t)]$$

$G_t$ which is the total reward over the whole episode is only available after the episode has ended, which means that the updated $V(s_t)$ is not available before the episode is finished. This is what makes Monte Carlo methods different from TD methods and also makes TD learning faster, especially if you have long episodes.

Q-learning can be used by the agent to learn optimal actions, just like TD learning. The concept of the Q-learning algorithm was first mentioned in Christopher Watkins's paper: "Learning from Delayed Rewards"[9]. Unlike TD learning which tries to optimize the value function, Q-learning is focusing on finding the optimal action-value function. The goal here is to optimize the total reward gained when taking different actions, and not the reward from being in a state. Q-learning is called an off-policy TD control algorithm[1][From Ch. 6.5] and is updated as follows:

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha[R_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{1.3}$$

TD learning is dependent on the probability of going from one state to the next one which is given by the policy, indifference, Q-learning only updates its state-action pairs and the policy only affects which state-action pairs are visited. A policy called Epsilon-Greedy or $\epsilon$-greedy often used in Q-learning will be presented later. Also, $max_a Q(s_{t+1}, a)$ represents the best known reward going from state $s_{t+1}$ to $s_{t+2}$ which is an estimate. Shown below is some pseudo-code for the Q-learning algorithm.

**Algorithm 1** Q-learning

set parameters: learning rate $\alpha \in (0, 1]$, $\epsilon \in (0, 1]$
Q(s, a) is initialized randomly
**for** each episode **do**
    init $S_t$
    **for** each step in episode **do**
        Find $A_t$ by the use of $\pi$ and Q
        Take action $A_t$ in state $S_t$, gets $R_{t+1}$ and $S_{t+1}$
        $Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, A) - Q(S_t, A_t)]$
        $S_t = S_{t+1}$
    **end for**
**end for**

$Q(S, A)$ is often referred to as the Q-table. This means that this function maps every state-action pair to a Q-value. An example of a Q-table is shown in figure 1.1. Here we can see that the Q-value for action a3 has the highest value, and is the estimated best action in state s1

| State-Action | Q_value |
|---|---|
| (s1, a1) | 3 |
| (s1, a2) | 1 |
| (s1, a3) | 5 |

*Table 1.1: Example of a Q-table*

### 1.2.4 Policy

When knowing how Q(s, a) is updated, we can focus on how the state-action pairs are generated during training. Assume you start in state $s$ with the possibility of taking all the action available in $s$. Then you can find the best action by looking into the Q-table for the action which gains the best reward known so far. This can lead to a good result now, but will not likely lead to any discoveries. On the other hand, if the agent also takes some random actions not dependent on earlier knowledge better results can be gained in the long-term. At this stage the concept of exploration vs exploitation comes into play.

To be able to both explore the environment and exploit it, methods like the Epsilon-Greedy have been developed. Epsilon-Greedy is a policy frequently used in Q-learning and can be expressed as follows:

$$a_t = \begin{cases} argmax(Q(s_t)) & \text{with probabilty 1-}\epsilon \\ random\ action\ in\ Q(s_t) & \text{with probability } \epsilon \end{cases}$$

here the policy either chose the best-known action $argmaxQ(s_t)$ or a random action. For Epsilon-Greedy, $\epsilon$ is initialized as one then decays to zero as the number of episodes increases. This means that the Epsilon-Greedy policy will lead to a lot of random action in the early episodes, but in the later stages of the training, the agent will be taking the expected best actions more often. The code

for this is shown below:

---

**for** episode in episodes **do**

$\epsilon$ *= epsilon_decay

**end for**

---

As mentioned earlier, Q-learning represents an off-policy learning strategy. This means that the agent used different policies when interacting with the environment and doing Q-table updates. When interacting with the environment, the agent uses the Epsilon-Greedy policy, but when updating the Q-table and picking the value of states it used the best known value(max Q(s, a)), in other words, a greedy policy.

## 1.3 Deep Q-Learning

Although Q-learning is a good way of mapping values to the specific state-action pair, the amount of computational power and memory needed as the state-action space or Q-table grows can be enormous, i.e. a *curse of dimensionality*[10] situation.

The fact that it seems impossible for an agent to learn the full state-action space the idea of a more generalized solution aroused. The goal was to create a function approximation that could give good results for the whole state-action space given only a subset of the space. One way to achieve this is to combine ideas from deep learning (Section 1.1.2) with the ideas in reinforcement learning into what we now call deep reinforcement learning (DRL). DRL replaces the Q-table which had the scaling problem, with a much more flexible artificial neural network. The possibility of using advanced algorithms has emerged with the rapid increase in available computational power during the later years.

ANN has been combined with many different learning algorithms from RL, but we will primarily focus on the combination with Q-learning known as deep Q-learning (DQL). Since the neural network then takes over some of the calculations we consider a slight adjustment equation (1.3) to make it easy to understand.

$$Q^{new}(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha \overbrace{[R_{t+1} + \gamma max_a Q(s_{t+1}, a)]}^{\text{new learned value}}$$

The *new learned value* is the only calculation during the update that is not handled by the neural network. When it comes to how often the neural network is fitted, or update it can vary. But often the network is doing a fit operation every step in each episode, meaning number of fit operations = number of episodes * length of episode.

Another concept that is used in DQL is *experience replay.*By using DQL we save all the state, action, reward and new state pairs as a transition in a replay memory of a given size. By having this buffer of earlier exploration, one does not only fit the model to the new transitions that come in, but also to some of the older ones. When fitting the model, batch sizes of random state-action

pairs are drawn from the replay memory. The reason for not just fitting the model to the newest state-action pair that comes in is that these pairs can have high correlation and lead to taking the same action over and over again.

It is also possible to use two neural networks to improve the training process. The two networks should share the same architecture but can have different weights. Usually, we have one main model and one target model. The weights from the main model are copied to the target model every N steps. So the target model is used to predict the future Q-table for the next state, meaning it is used to find the maximal Q-value available in the next state ($max_a Q(S_{t+1}, A)$ which is used to calculate the target. This is called *fixed Q-targets*.

The reason for not using the same model for this prediction is that we would then have used the same weights to estimate the target($\hat{y}$) and the Q-value(y). This would lead to an unfavourable dependency between the target and the weights we are changing. As a consequence, every time we are fitting the model we shift both the Q-values and the target values, meaning we are chasing a moving target. Having a target network that is not updated so often contribute to preventing disturbing oscillations during training to speed up the process.

The last method we will consider builds on the use of two neural networks is called double deep Q-networks (DDQNs) introduced by Hado van Hasselt[11]. Instead of using the target network to find the best-known action, the main network of DDQNs finds the action and the target network predicting the Q-value for that action.These changes are done with the intent of reducing the over-estimation of the Q-value and lead to faster and better training. The training part of the DDQL model updating can be described in more detail by the following pseudo-code:

**Algorithm 2** Double DQL - Train function (agent.train())

1: **if** replay memory $> min\_replay\_memory$ **then**
2: $minibatch = random.sample(replaymemory, size = minibatch\_size)$
3: $current\_states = minibatch[0]$
4: $current\_qvalue\_list = main\_model.predict(current\_states)$
5: $next\_states = minibatch[3]$
6: $future\_qvalue\_list = main\_model.predict(next\_states)$
7: $target\_future\_qvalue\_list = target\_model.predict(next\_states)$
8: $X = [], y = []$
9: **for** $index, (current\_state, action, reward, \_, done)$ in enumerate(minibatch) **do**
10: **if** not done **then**
11: $best\_action = argmax(future\_qvalue\_list[index])$
12: $new\_q\_value = reward + DISCOUNT\_FACTOR * target\_future\_qvalue\_list[index][best\_action]$
13: **else**
14: $new\_q\_value = reward$
15: **end if**
16: $curren\_qvalue\_list[index][action] = new\_q\_value$
17: $X.append(current\_state)$
18: $y.append(current\_qvalue_list[index])$
19: **end for**
20: $main\_model.fit(X, y)$
21: $update\_target\_model\_counter$ += 1
22: **if** $update\_target\_model\_counter == UPDATE\_TARGET\_MODEL\_EVERY$ **then**
23: $target\_model.set\_weights(main\_model.get\_weights)$
24: $update\_target\_model\_counter = 0$
25: **end if**
26: **end if**

The logic that distinguishes the DDQL from the normal DQL is described in line 6, 7, 11 and 12. Here we can see that the main model picks the action, and the target model finds the Q-value. The code also shows how the target model is not updated every time and is dependent on the static variable $UPDATE\_TARGET\_MODEL\_EVERY$.

**Algorithm 3** DQL training logic

```
1: for number of episodes do
2:     current_state = environment.reset()
3:     done = False
4:     while not done do
5:         if random(0, 1) > ϵ then
6:             action = Do the best action
7:         else
8:             action = Do a random action
9:         end if
10:        new_state, reward, done = environment.step(action)
11:        agent.update_replay_memory(current_state, action, reward, new_state, done)
12:        agent.train()
13:        current_state = new_state
14:    end while
15:    ϵ *= EPSILON_DECAY
16: end for
```

Algorithem 3explains the logic for each training episode. First the basic of resetting the environment and setting done to False is done. From there the policy chooses either to take a random or the best action known (line 5-9). The action is then done on the environment and the replay memory is updated with the necessary data (line 10-11). Now the agent can do the train function *.train()*, where algorithem 2 shows the logic inside the function (line 12). This train function does not have to run every training step. The frequency of how often it should run can depend on the length of the episode. At last, after each episode epsilon is reduced by a decay factor EPSILON_DECAY $\in [0, 1]$ often chosen very close to one.

## 1.4 Multi-agent reinforcement learning

Multi-agent reinforcement learning (MARL) is used in cases where multiples agents can interact either sequentially or simultaneously with the environment[12]. In general, the concepts are similar to single-agent RL, but in MARL one is also interested in exploiting the interactions between the different agents. For our project, the goal is to improve the traffic flow not just for one intersection, but over a network of multiple intersections where each is controlled by an agent. The following is an illustration of MARL:

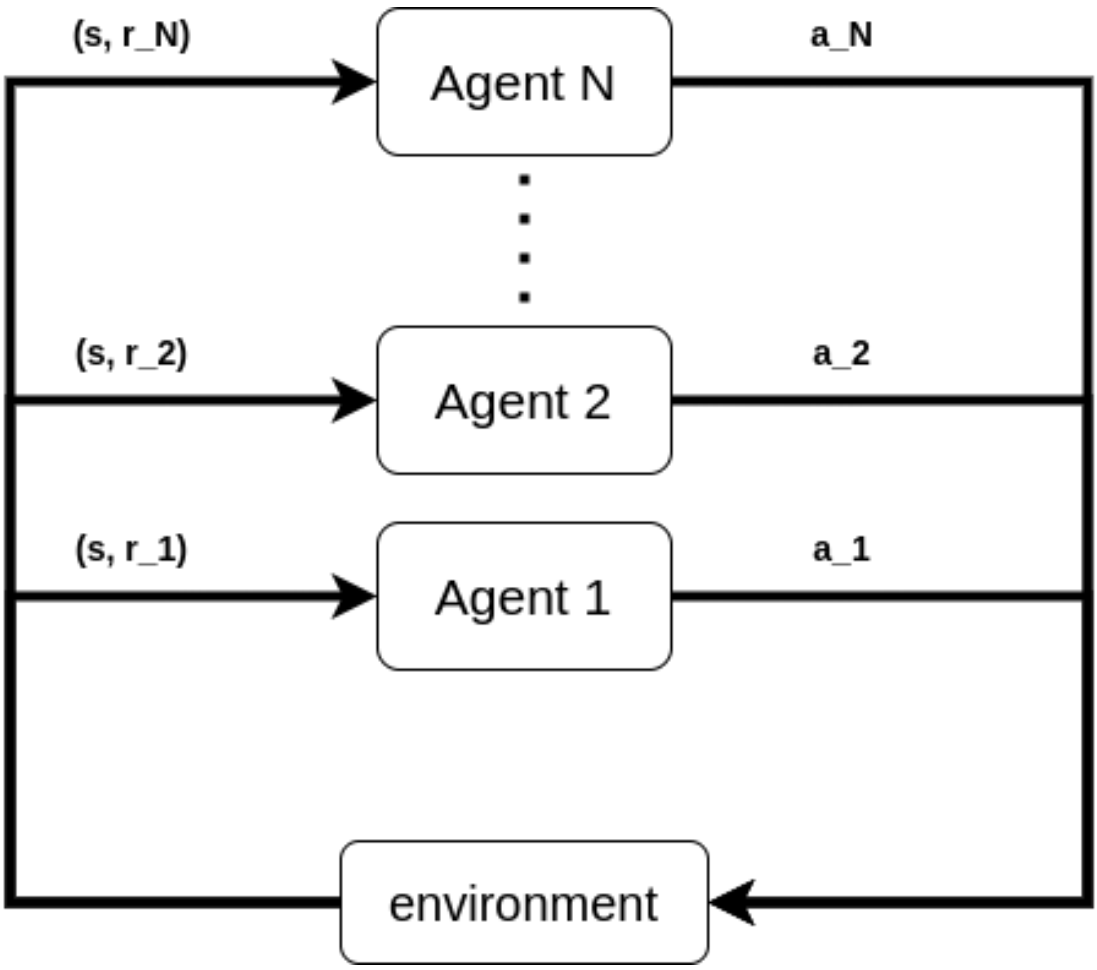



*Figure 1.6: Illustration of multi-agent reinforcement learning, where all the agents act simultaneously*

Figure 1.6 shows how multiple agents take in the same state do their specific actions, and receives their specific rewards. Here all the agents act simultaneously, meaning that all agent actions are executed at the same time.

There are several ways of structuring how the agents interact with each other. One option is called centralized setting where a central controller which sits on information about the whole environment, the states and the actions where the rewards for each agent are all joint together. In this senatrio, all the agents will act as one big team. This approach can lead to a huge growth in the number of state-action pairs, meaning it requires a lot of training to approximate an optimal solution. Another option is called *fully decentralized setting*. As opposed to the centralized option, each agent now only takes actions based on the information known in their part of the environment. In our application this means that each agent is only concerned about regulating their own intersection.

## 1.5 Traffic system

For an agent to be able to control the traffic of an intersection, we need to establish specific states representing the different light combinations. An example of an intersection is shown in figure 1.7. The state-space used is also presented as strings in figure 1.8. This space has the size of 4 and the letters representing the different lights are described in tabel 1.2. The length of a string representing a state depends on the number of incoming lanes times the number of available outgoing lanes that is 4  3 = 12 for this particular example. If we only look at the three first letters we get information about cars coming from the east and if they can go to the right, straight or left accordingly.

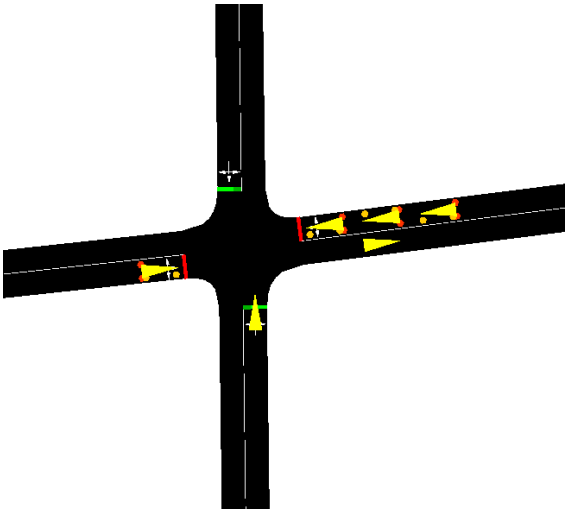


*Figure 1.7: Intersection created in SUMO.*

```
state="rrrGGgrrrGGg"/>
state="rrryyyrrryyy"/>
state="GGgrrrGGgrrr"/>
state="yyyrrryyyrrr"/>
```

*Figure 1.8: Corresponding state space for the intersection. The first state is the active state*

| State-Action | Q-value |
|---|---|
| G | Green light with priority |
| g | Green light without priority |
| y | Yellow light |
| r | Red light |

*Table 1.2: Explanation of the letters in the state string*

In out project, the listed light states in the state-space may not be available at all times. Some basic rules are given to the agent before training:

- A yellow state must always follow a green state.
- The yellow state must be no shorter than the minimum yellow time given.
- The green state must be no shorter than the minimum green time given.

### 1.5.1 Controll now

The traffic lights in Norway are controlled in different ways. Some intersections check for incoming traffic to each lane by detectors, then set a state of the lights over a given time. Traffic lights may also have specific static rules based on the time of the day and incoming traffic. Than the traffic control is based on real data collected from the intersection. This can mean giving more green lights to vehicles going into the city in the mornings and vice versa in the afternoon. Likewise, it can also be a priority for buses and other public transports which needs to be taken into account when distributing green lights. There may also be situations where you would want to reduce the capacity of the intersection to reduce traffic to more sensitive areas.

The amount of data available also differs from intersection to intersection. Some intersections may have a sensor where the queue starts and can detect if vehicles are waiting. It is also possible to have these sensors moved further away from the intersection to give earlier information about incoming vehicles, but then one can not be hundred percent sure that the vehicle is really going into the intersection. There are also cases where image recognition technologies are used to estimate the size of the queues. Such technology can also give information about incoming big trucks or buses. Also GPS technology has been used to track incoming traffic, especially for public transportation.

### 1.5.2 SUMO

Simulation of urban mobility (SUMO) is "an open-source, highly portable, microscopic and continuous traffic simulation package designed to handle large networks"[13]. The networks can be customized with elements like cars, public transports and pedestrians. Also, interactions with the environment can easily be implemented in SUMOs python API or the package called *TraCI*. TraCI lets you set, change and control the behaviour of the environment buy some useful integrated functions. There is also the possibility of simulating without the GUI interface that makes the training of the RL agent much faster.

### 1.5.3 Traffic distribution

In this project, the initialization of each vehicle is random and is independent of the other vehicles. This means that for each time step we use the same probability that a new vehicle will enter the simulation. It is also impossible to initialize two vehicles to the same lane at the same time. In other words, two cars can not drive upon each other. Because of this, the distribution of time between a new vehicle from one lane can either be binomial or Poisson distributed. Recall that the binomial distribution converges towards a Poisson distribution as $n->\infty$, $p->0$ and the mean n*p is held to be constant[14], where $n$ is the number of trials, and $p$ is the probability of a new vehicle initialize onto the road.

The total number of vehicles included in the simulations is a product of the distribution of vehicles to the roads. An exact expression for the number of vehicles from one lane in a simulation is:

$$numberOfVehicles = simulationTime - \sum_{i=1}^{n} timeBetweenVehicle_i$$

Note that $numberOfVehicles$ is also equal to just $n$ which is the number of trails. To obtain the total number of vehicles in the simulation, the $numberOfVehicles$ from each lane ($i = 1, ..., m$) is summed together:

$$\sum_{i=1}^{m} numberOfVehicles_i$$

This final value is a product of binomial or Poisson distributions summed together over all $m$ lanes. Due to the central limit theorem [15], the distribution of the total number of vehicles in each simulation in our case is approaching a normal distribution as the number of simulations grow large.

# Chapter 2

# Methods

## 2.1 Environment configurations

### 2.1.1 Sumo setup

When setting up the SUMO configuration two XML files and one .sumocfg file are used. The .sumocfg file is in XML format and refers to the two other XML files containing detailed information about the traffic environment.

The network file .net.xml has the information about the design of the intersection, containing the connections of the roads and the traffic lights. We give each of the roads the possibility to send the vehicles to all the other outgoing lanes except the one from which it came from. The speed limit of the individual roads is set to 13.89 m/s which is approximately 50 km/h and the state-space for which the traffic lights can be in is initialized.

The other XML file .rou.xml contains information about the vehicles, type of vehicle, and the routes of the vehicles. Which route a vehicle is taking and when it is initialized to the environment is decided by a *flow* function. This function spawns one type of vehicle having a specific route from $A$ to $B$ at a given spawn rate, meaning we have one flow function for every possible route in the network. During the training of the agent, the spawn rate is 0.005 for every flow function. Meaning it is a 0.5% chance of initializing a car from a specific lane $A$ to a lane $B$ every timestep. This is done to try to make the driving patterns as close to real as possible. To ensure that the random effect is truly random, a random seed for each simulation is chosen based on the current system time on the computer.

### 2.1.2 The traffic environment

The traffic environment used in this project is an approximation of some parts of Sannergata and Toftes gata in Oslo (Figure 2.1). These roads are connected with three intersections, whereas in our project every incoming and outgoing lane has two lanes. Each traffic light has the same state-space as in Figure 1.8.

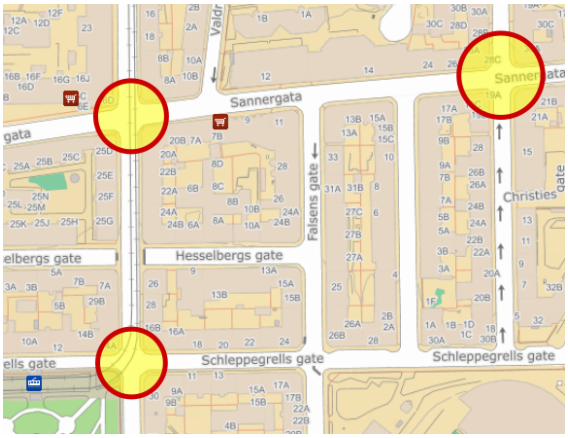



*Figure 2.1: Sannergata and Toftes gate in Oslo, Norway.*

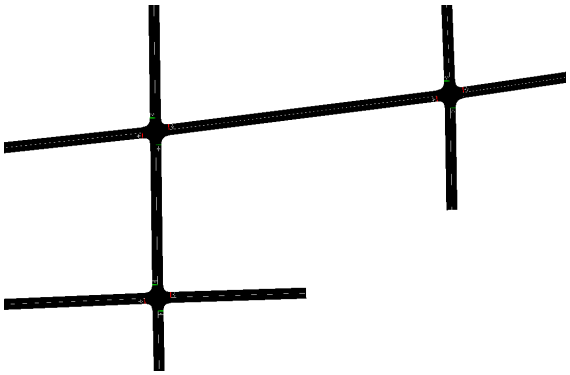


*Figure 2.2: Corresponding approximation of the streets in SUMO.*

The length of each simulation is 500 time steps, whereas each time step is one second. The main reason for not choosing longer episodes is because of the rapidly growing traffic queues that may occur in the early training phase. This could lead to a lot of fit operations on states with a lot of similarities. On the other hand, the reason for not having shorter episodes is because of the number of timesteps at which there are no cars yet to reach the intersection at the start of each simulation. Having each time step equal to one second gives the agent the possibility of acting on the environment every second. This is done in regards to real-time, where the agent needs to get the state from the hardware(sensors, camera, etc) and calculate the new action, all within that second.

## 2.2 Project architecture

To be able to train the agents, we use software implementing models based on the theory explained throughout chapter 1. For fitting the models to our experiment, we need to declare the states, actions, rewards, and how this is collected during a traffic simulation. Figure 2.3 gives an overview of the workflow. Below, we will explain distinct behaviours in our environment which needs to be handled. An example of this is the the case of "locked state" for a traffic light.

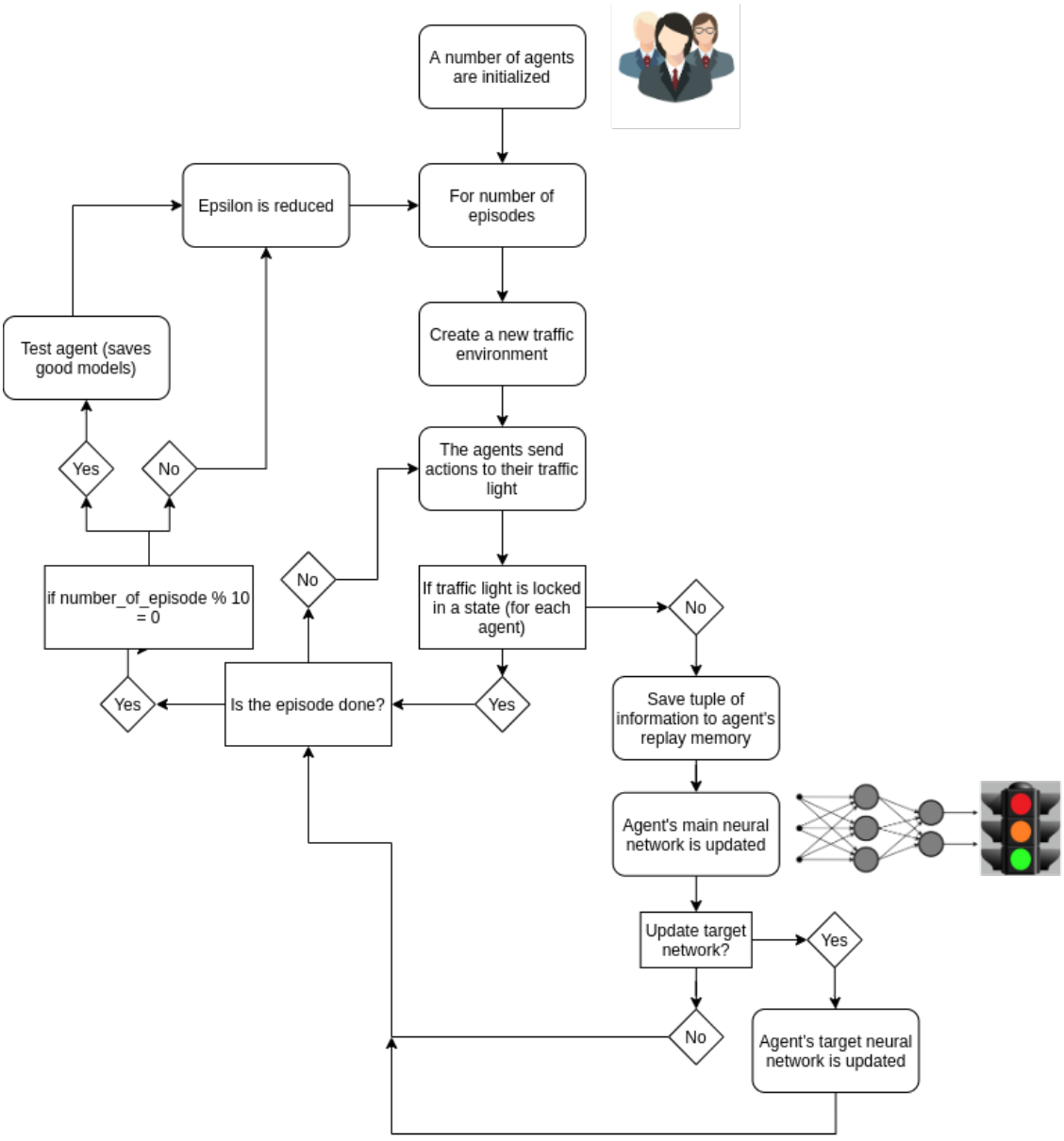



*Figure 2.3: Flowchart of the workflow*

Our set-up requires three agents for controlling the traffic lights - one for each intersection. These three agents are all created with the DDQL logic described in algorithm 2 and 3.

Since we have multiple agents acting together in one environment, a MARL approach is required. As mentioned in Section 1.4, there are several ways for the agents to interact with each other. For the centralized setting, a state-action pair would look like $(s, a_1, a_2, a_3)$ and even a more general reward function for all the agents could be used. Even though this is likely to lead to the best results, the training time and computational power needed to produce good results for the scenario were unfortunately not available to this project (different interaction settings were tested to see the

potential on a lower amount of simulations). The project was therefore restricted to considering only two different situations. One of these situations was fully decentralized where each agent only had information about their own intersection and light state. For the other situation, each agent had access to the light states of the other agents. Since all the agents were acting simultaneously, their shared light state was the previous one, and not the one which the agents were changing to. Below these settings will be referred to as scenario 1 and 2.

The architecture of the neural network used in each agent is the same with two hidden layers of size 32 and 16 nodes. Between the two layers, there is a dropout layer for regularization of the model. The size of the input layer is the same as the size of the states and the size of the output layer is the same as the number of the actions. The reason for not having a deeper network is because of the limitations of the hardware, meaning we wanted a network that did not require too much computational power to update. It also seemed like the solution to the problem would not be extremely complex, and the 32x16 network would manage to find fairly good approximations.

### 2.2.1 State space

lthough we have described the various possible states of the traffic lights, we have not described the states that the agent takes in when calculating the best possible action. These states should usually have the information needed to find a good policy. The state given to the agent in scenario 1 for the intersection in Figure 1.7 is shown in the array below. This array is fed into the first layer of the neural network, meaning the size of the state is the same as the size of the input layer in the network.

$$[\overbrace{3,1,1,0,}\ \overbrace{0}\ ]$$

the same state, but in scenario 2:

$$[\overbrace{3,1,1,0,}\overbrace{0,1,0}]$$

For both scenarios, the number of cars in each lane is represented by the blue part of the array. The red part indicate the current traffic light state(s) the intersection or intersections. For our project 0 and 2 represent the two possible green states and the number 1 represents any yellow state. The reason for not including every yellow state as a separate value is to reduce the size of the state space. This is because one never wants to change to a specific yellow state, as they are a result of the previous green state. Another *simplification* that is done, is to reduce the number of car value to a maximum of seven even if it is more than seven cars in the lane. Meaning the values in the blue part of the array can never be bigger than seven. This is primarily done to reduce the state space, but it also reflects the hardware limitations. A camera that uses machine learning to count the number of cars in the lane may not be able to count more than a given number of cars. With these limitations, there are $7^4 * 3$ and $7^4 * 3 * 3$ number of different states in scenario 1 and 2, respectively.

The idea of using the number of cars in queue and not in the lane was also considered. The reason for not pursuing this approach where that the cars are in queue if they have a speed of zero. So if

a lane with a large queue gets greed priority, the subsequent state may show zero cars in the queue since all the cars are now in motion. This can lead to a misconception regarding the state.

### 2.2.2 Action space

As mentioned earlier, the actions are events altering the behaviour of the environment. In our application, the only thing that can do this is the traffic lights, meaning each state of light combination is an action. Since half of the states are and yellow states, the size of the action space can get quite large without much difference in the actions. To reduce the number of actions we combine the action of going to a yellow state or not change state to one action. From this restriction the total number of actions available will be the number of green states + one.

Even though the agent gets information for every time step(ts), there will be situations where the agent is not allowed to do actions. These situations are when the intersection is going from one green state to another. During these intervals, the traffic lights are first forced to change to the belonging yellow state of the old state for a minimum duration of three(static variable) time steps. Then the traffic lights are changed to the green state requested at the start of the interval for at least a minimum duration of five(static variable) time steps. It is not until this interval of eight time steps is over that a new action can be executed on the environment.

### 2.2.3 Traffic rewards

We have previously mentioned the rewards, where the goal of the agents is to maximize the total rewards during an episode (1.2.2). How the reward is defined and how it is calculated plays a huge part when it comes to how the agent behaves. A sensible reward may be the goal itself. Meaning if the goal of the episode is to get as many cars through the intersection, the number of cars passed between each action should also be taken as the reward. A disadvantage concerning this choice of reward may be that during training the episode's length is fixed, while the number of cars can vary. If the total amount of cars entering the environment is low, it may be easy for the agent to get everyone through in time. Since it does not matter how fast the cars get through the intersection, only that they get through during the episode, some unnecessary queues may occur.

Our approach will focusing on the accumulated waiting time of each vehicle in the queues as the negative reward. The goal of the agent will then be to minimize the total waiting time not just for one time step, but for the whole episode. When we are considering multiple intersections together in one system it is important to distinguish between each vehicles waiting time at each intersection. Meaning the waiting time of $car_1$ in intersection 1 is not counted when the same car is waiting in intersection 2. The pseudo-code for these waiting time calculations are given here:

**Algorithm 4** Calculate waiting time

```
dict of cars is initialized at the start of the simulation
total_waiting_time = 0
for lane in lanes do
    cars = get_cars(lane)
    for car in cars do
        car_waiting_time = get_waiting_time(car)
        if car not in car_dict then
            car_dict[car] = lane: car_waiting_time
        else
            car_dict[car][lane] = car_waiting_time - sum(car_dict[car] expect when key == current lane)
        end if
        total_waiting_time + car_dict[car][lane]
    end for
end for
```

### 2.2.4 Generating data

When it comes to how the data is generated, we have in section 1.3 mentioned that during a transition in the environment the tuple containing (state, action, reward, new state) is stored in a replay memory. From the three previous subsections, we have explained the content of each of the three first variables. The last variable in the tuple *new state* contains the resulting state by taking the action in the specific state. As mentioned earlier, the agent is not always able to act on the environment because of the rule regarding green and yellow minimum time. This affects the creation of new tuplets because a tuple is only created when an action is done on the environment. It is possible to have the agent storing the tuple even though the action does not have an impact. But since the action will not affect the reward, the agent's neural network will be fitted to a lot of unnecessary tuplets that may slow the learning process.

The reward for each tuple is the total waiting time for all the cars in the specific intersection when the action is fully executed. . This means that if the agent's action is to stay at the same traffic state the waiting time in the next state is the reward. But if the action changes from one greed traffic state to another, the waiting time is not calculated until the min green and yellow times are completed. This is done to make it easier for the agent to measure the effect of its actions.

## 2.3 Hardware and software

We will now list the hardware used when training the agents. The reason for this is because the runtime can affect how the agent is designed. Even though a GPU most likely would have been faster regarding training the neural network, we did not have this available. Instead, we used an Intel Core i5-8250U CPU with 4 cores and a maximum turbo frequency of 3.4 GHz. For this project, memory was never a problem and was nothing to be concerned with (size of replay memory).

The computer running the calculations required download and execution of the SUMO software which is available for most operating systems. Multiple Python libraries have been used during this project. TensorFlow [16] is used to create the neural networks and their behavior, but not the agents themselves. Also, to be able to interact with the data points in a good way, NumPy [17] was used. In addition to these, some minor packages used for convenience, but TensorFlow, NumPy and SUMOs TraCI were the major contributors.

## 2.4 Testing

For testing the agents, a model using only sensors, static time intervals and a queue system was created. This logic is a basic queue system used in some traffic intersections today. Here the states are distributed after the sensors discover new cars coming into the intersections. The traffic lights then change to the requested state or add their request to the queue if another state interval is already running. If the queue already has the state stored it will not be added once more, meaning the length of the queue can never be bigger than the number of green states. Then after the given interval, the state is removed from the queue and a new state is take in. To be clear, these traffic states are also known as the actions available to the model.

The criteria that the models are tested upon are the total waiting time for all cars that have been in the intersections during the simulation. This value is obtained by taking the sum of the values in the dictionary $car\_dict$ from algorithem 4 at the end of the episode.

# Chapter 3

# Results

## 3.1 Parameters

To obtain interesting results, different parameters had to be set. We divided the parameters into two groups (1 and 2). Parameter group 1 contained the parameters defining the amount of data, how they were generated and stored. The parameters for group 1 are shown in Table 3.1. For the size of the replay memory, two alternatives were used in training. The size of the mini-batch was set to 32. Meaning each time the model does a *fit* operations, 32 random X and y values stored in the replay memory are used to update the network. The size was found based on trial and error. A larger size can lead to poor regularization, and a smaller size may lead to less effective training[18].

| Parameter | Value |
|---|---|
| Replay memory size | 100000, 5000 |
| Mini-batch size | 32 |
| Episode length | 500 |
| Number of episodes | 500 |
| Epsilon decay | 0.99 |
| Min epsilon | 0.001 |

*Table 3.1: Group 1 parameters*

The parameters for group 2 were the parameters affecting how the agent updates itself and the architecture. Table 3.2 shows these parameters. Note that the discount factor has two values that are used together with the best replay memory value. We use *Adam* and *ReLU* as the optimizer and activation function in our models. Adam is an algorithm used to optimize stochastic gradient descent when training a neural network [19]. Even though our network architecture is not that deep, we chose ReLU activation functions to avoid slow training caused by the *vanishing gradient problem*[20].

| Parameter | Value |
|---|---|
| Discount | 0.95, 0.5 |
| NN optimizer | Adam |
| NN activation function | ReLU |
| Learning rate | 0.01 |
| Update target network | every 500th .train() |
| Dropout rate | 0.2 |

*Table 3.2: Group 2 parameters*

## 3.2 Training results from both scenarios

Here we present the training result from the two different scenarios that have three different parameter setups during training. The training result shows the sum of the negative waiting time for each car in each intersection over the whole episode. This means that the values in these plots (3.1 - 3.3 and 3.7 - 3.9) are based on both random and best known actions from the agents. The graphs related to these results show values between -1 000 000 and 0 in the vertical direction, meaning a value of zero indicated no waiting time at all. Each point in the line plot covers an interval of ten episodes, and the average value (red line) is plotted together with the max and min values (blue filling) over these ten years. Finally, epsilon decay values are plotted (green line). As mentioned earlier, the epsilon-values (between 0 and 1) defines the probability of choosing a random action in the Q-learning algorithm.

Based on the training results from each 10th episode, the agents are tested as explained in Section 2.4. Here the goal is to obtain the lowest score possible, and the graphs show values between 0 and 12 000 (3.4 - 3.6 and 3.10 - 3.12). The green line is still the epsilon value.

As mentioned in Section 2.2.4, Algorithm 2 updating the neural network of the agents is only executed when an action has an impact on the environment since a lot of heavy computational power is required. We found that the number of executions of this function is between 63 and 500 in each episode for each agent. Since the agents are only tested once every 10th episode you can argue that a lot of changes may have gone unnoticed over all these fit operations between the 10 episodes. The test is also only done on one episode, meaning the traffic which is randomly generated in this episode may be biased to a specific solution. So ideally one would want to test the agents more frequently but since the execution of the test function took close to half a minute, we decided not to increase this.

Regarding the runtime of each episode, approximately 90% of the execution time was used to storing tuples and updating the networks, while 10% was used on modifying and creating the environment (about 10 seconds). As the number of episodes increased during training, the runtime of each episode stabilized between 80-120 seconds (updating the networks caused most of the variation). Together with the time it took to test every 10th episode, there was a total runtime of up to 14 hours for a group of agents to be trained over 500 episodes.

### 3.2.1 Scenario 1

In this scenario each agent had only information about the state of its own traffic light. We have trained and tested three different cases for each scenario where the two first has a discount factor of 0.95 and 0.5 together with a replay memory of 100 000. For the last configuration, we used the best performing discount factor of 0.95 together with a small replay memory of only 5 000.

By considering the corresponding three plots in Figure 3.1-3.3, we can see that the configuration in Figure 3.1 outperforms the two other versions when it comes to having a small negative waiting time. This plot shows that its results have a relatively low variance (the blue filling), and the values are neither decreasing nor increasing significantly towards the end of the 500 episodes. In comparison, the results in Figure 3.2 and 3.3 shows good performance at the start of the training but decreases significantly as the number of episodes increases, but both show a larger variance in the results (size of blue filling) than for the first configuration. What distinguishes the second and third configuration is the *spiky* behavior in Figure 3.3 and that its y-value is more consistently decreasing.

**Accumulated waiting time during training**

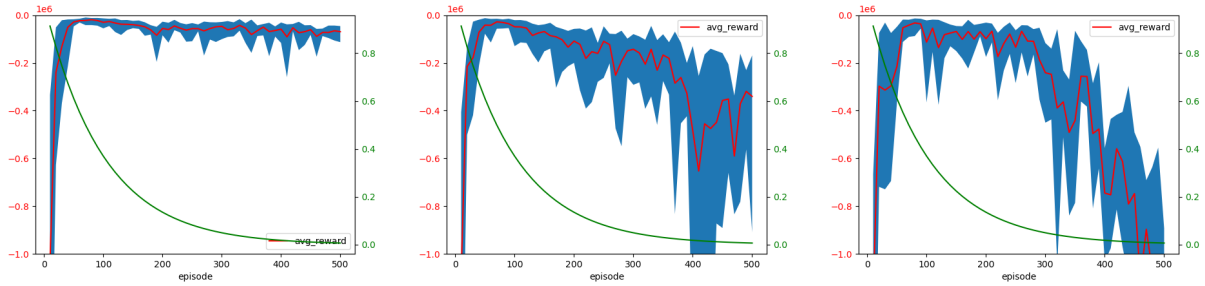



*Figure 3.1: Discount = 0.95, replay momory size = 100000*

*Figure 3.2: Discount = 0.50, replay momory size = 100000*

*Figure 3.3: Discount = 0.95, replay momory size = 5000*

Figure 3.4, 3.5, 3.6 show the tested results for the connected training plots over. For all the plots a spiky pather is occurring, and the value between each 10th episode varies a lot. Not surprisingly, Figure 3.4 shows a plot of more consistent low values, and one test value even goes below 2 000 (episode 200). For the two other plots, the y-value is generally a bit higher and has bigger differences.

**Total waiting time, tested every 10th episode**

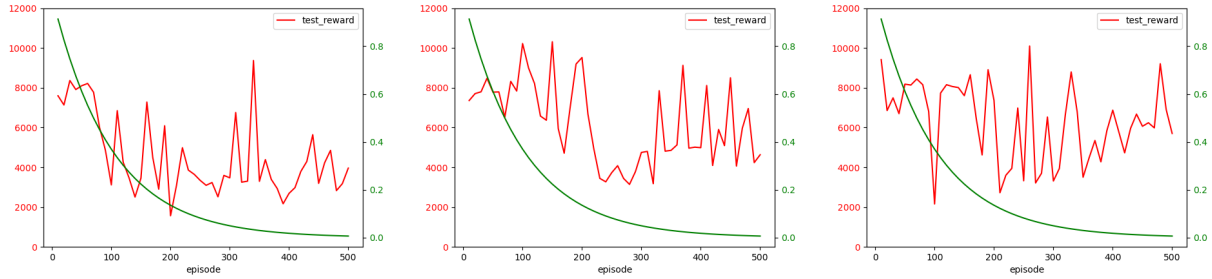



*Figure 3.4: Discount = 0.95, replay momory size = 100000*




*Figure 3.5: Discount = 0.50, replay momory size = 100000*




*Figure 3.6: Discount = 0.95, replay momory size = 5000*

### 3.2.2 Scenario 2

In this scenario, we considered a modification of the same three configurations where each agent was given information about the light states at the other intersections. As described in Section 2.2.1 the agents' states were then larger leading to slightly slower training. Similar to the training plots for scenario 1, the first case (Figure 3.7) with a discount factor of 0.95 and a replay memory of 100 000 performed best with training results quite similar to those from scenario 1 in Figure 3.1. However for the two other plots 3.8 and 3.9 the results looked slightly better than in scenario 1. The variances are still larger than in the first scenario, but the y-values do not seem to decrease that much or at all over time. For Figure 3.9 we still see a large difference in the average value similar to scenario 1.

**Accumulated waiting time during training**

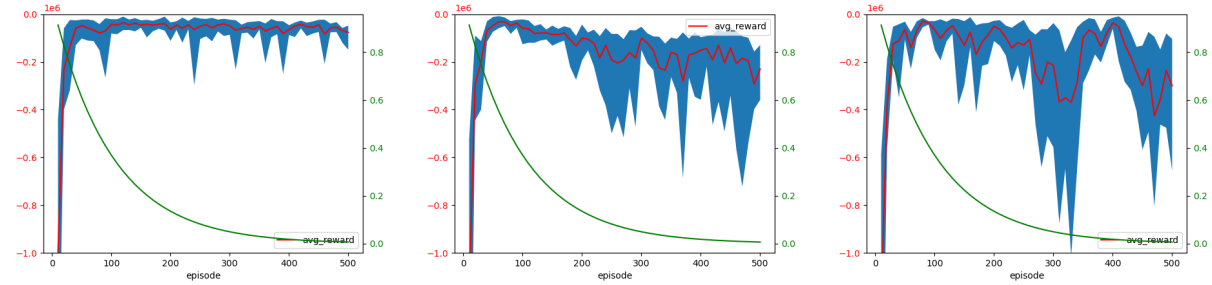



*Figure 3.7: Discount = 0.95, replay momory size = 100000*




*Figure 3.8: Discount = 0.50, replay momory size = 100000*




*Figure 3.9: Discount = 0.95, replay momory size = 5000*

As in scenario 1, the first configuration resulted in the overall lower values. Figure 3.10 shows some spikes but still a large number of values between 2 000 and 4 000, and some even below 2 000. The two other configurations resulted in poorer results, but Figure 3.12 shows some relatively good (low) values towards the end.

**Total waiting time, tested every 10th episode**

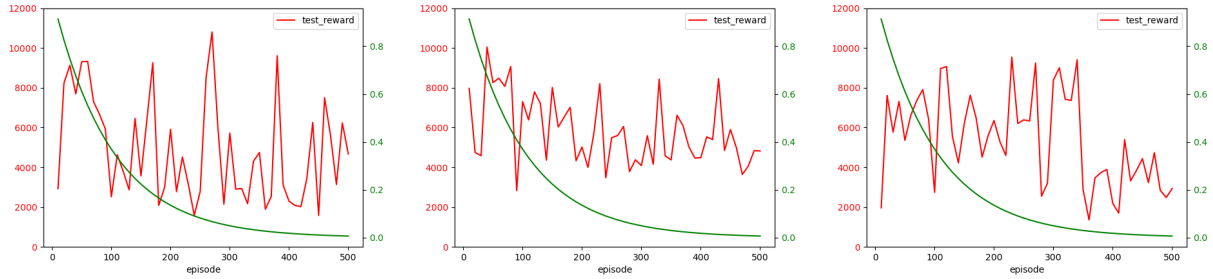



*Figure 3.10: Discount = 0.95, replay momory size = 100000* *Figure 3.11: Discount = 0.50, replay momory size = 100000* *Figure 3.12: Discount = 0.95, replay momory size = 5000*

## 3.3 Test results

Finally, we selected the best performing configurations of scenarios 1 and 2 together with the more static model described in Section 2.4. These three models were tested based on a total of 10 episodes by calculating and comparing the average $\mu$ and standard deviation $\sigma$ of the total waiting time. We set the Traffic simulator to spawn cars with three different rates for 10 different episodes, one with the same as during training, one with twice the amount of traffic coming from the north and south lanes, and one with twice the amount of traffic from all lanes.

| **Type of Traffic** | **Static test model** | **Scenario 1** | **Scenario 2** |
|---|---|---|---|
| Normal | $\mu$: 2202.4 $\sigma$: 417.0 | $\mu$: 2272.9 $\sigma$: 389.3 | $\mu$: 1866.8 $\sigma$: 548.9 |
| More from North and South | $\mu$: 4854.0 $\sigma$: 1750.1 | $\mu$: 5517.5 $\sigma$: 1422.4 | $\mu$: 4110.7 $\sigma$: 463.1 |
| More from alle lanes | $\mu$: 9200.8 $\sigma$: 1096.8 | $\mu$: 10866.4 $\sigma$: 1606.9 | $\mu$: 7310.0 $\sigma$: 1018.6 |

*Table 3.3: Test results, $\mu$ = Total waiting time of all cars in one episode averaged over 10 episodes, $\sigma$ = The standard deviation of this value*

From Table 3.3 it is clear that the chosen model from scenario 2 gave the best performance (lowest average value and lowest variance except in the case of normal traffic) for all three settings of the traffic simulator. Moreover, the static model performed better than the model from scenario 1 for all settings. Possible explanations of why the models performed as observed for the testing episodes will be presented in the discussion below.

# Chapter 4

# Discussion and future work

Based on the training- and test results, it is fair to conclude that multiple RL agents can control a traffic system. It seems that the models for both scenarios work without blocking lanes or disturbing the traffic unnecessarily in any major way, but based on the test results it seems that scenario 2 performs better than scenario 1. With the use of SUMOs GUI interface, we could observe how the different models controlled the intersections more closely. From this, we saw that the model from scenario 2 seemed to hold the same state for longer periods to save time in the long run. This model also preferred to have the lights stay in a yellow state if there were no incoming traffic. This enabled the agents to change to the desired state faster (without having to wait for an earlier action of essentially no impact). We assume that both of these differences have evolved from the additional information each agent was given about the other agents' light states.

As pointed out earlier, traffic generated in each episode can be biased towards a specific model, and an episode with less traffic will always be positive regarding the waiting time which we are scoring the models against. With this in mind, an exception rule could have been invoked during testing if the number of cars entering the environment during an episode did not exceed a threshold to slightly improve the quality of the test results.

By focusing on the three different parameter setups for each scenario, patterns seem to repeat themselves. For Figure 3.3 and 3.9 we see that the average value starts oscillating with around 20 episodes between each extremal point. This is most likely due to the small replay memory available to the agents during the learning process. When new tuples of data points are entered into the memory, critical tuples of data may be suppressed and important knowledge forgotten when the agent updates its network without including information from these data points. This small buffer may also lead to overfitting on a lot of similar data points, which may have been the case in Figure 3.3. The case based on the parameter combination behind Figure 3.2 and 3.8 looks to be unable to find the best solution over time. For this case, the agents do not seem to emphasize the future reward as much as the immediate reward. This seems to contribute to the agent fitting its neural network more and more towards selfish actions, which in this case do not contribute to making a better model.

Another aspect to notice is that the vehicles in our simulated environment are only depending on other vehicles, which is not so usual. In real-life scenarios, pedestrians, bicyclists and other unexpected events may affect the driving situation and inhibit optimal driving for the vehicles. For

future work, it will be interesting to add pedestrians and bicyclists to the intersection, in addition to adding more randomness into the driving behavior of each vehicle. By the latter, we mean that it is not realistic to assume that every vehicle reacts optimally to the traffic light changes and the other cars behavior. Such additional inclusions may lead to models which can perform better in real-life scenarios and is more robust against unexpected events.

Since the computational resources (GPUs etc.) required to obtain good models ended up been quite extensive, our project was prevented from exploring as sophisticated models as we ideally would have liked (such as more complex and realistic descriptions of the intersections). At the beginning of the project, we considered some intersections where from each direction we had two and three incoming lanes. By doing this we almost doubled and tripled the size of the states, and this resulted in an extreme growth in the size of the state space. Unfortunately, this led to extremely slow training on our equipment. However, with sufficient computational resources, such increases in model complexity would be straightforward to handle. Another issue that looks even more promising is to couple even more intersections together and having the agents act more centralized by sharing more information. From our results in Tabel 3.3, it is clear that it helps to manage the overall flow of the traffic when the agents share more information with each other (scenario 2). Ideally, all the intersections in a city controlled by a common system could be combined to share information. Such a system would of course require a lot of computational resources to be trained properly. The state-action space of an agent with access to all other intersections would be enormous, but with some simplifications represented in well-chosen neural network architecture (including the possibility of convolutional neural networks), a useful representation may be found.

# Chapter 5

# Conclusion

In this thesis, we have looked into the possibility of applying multi-agent reinforcement learning to control multiple traffic intersections beneficially. The motivation behind studying this problem was a desire to challenge the classical approaches to traffic light control, and investigate if such alternative solutions potentially could be scaled up to handle large realistic traffic networks. By using a reward function that calculated the waiting time at the intersections after the actions were executed, we obtained agents with good Q-value pairs for the states and actions. We trained two different models where the less decentralized alternative (scenario 2) showed better performance.

Since we generated three different types of traffic patterns during the final testing we introduced the trained models to unknown behavior. This seemed too complicated for the scenario 1 model that was also outperformed by the static reference model. On the other hand, the scenario 2 model outperformed the static reference model as well as the scenario 1 model. To conclude, the scenario 2 model seems to be a useful solution to the traffic regulation problem such as the one encoded in our simulation tool, also when it came to handling unseen traffic patterns that were not represented during the Deep Q-learning training process.

Although we experienced some problems concerning the available computational resources, we believe that with the right equipment available, multi-agent reinforcement learning based on deep neural networks may be a very powerful solution to the problem of efficiently controlling traffic flow in urban areas.

# Bibliography


[1] S. Raschka and V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2, 3rd Edition*. Packt Publishing, 2019.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] Sean Fleming, Senior Writer, Formative Content, "Weforum - INRIX," [Online; accessed 9-03-2021].

[4] Oslo kommune, "folkemengde-og-endringer - Oslo kommune," [Online; accessed 9-03-2021].

[5] P. W. McCulloch, W.S., "A logical calculus of the ideas immanent in nervous activity.,"

[6] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[7] S. K. Mahata, D. Das, and S. Bandyopadhyay, "Mtil2017: Machine translation using recurrent neural network on statistical machine translation," *Journal of Intelligent Systems*, vol. 28, 05 2018.

[8] H. J. Kelley, "Gradient theory of optimal flight paths," *Ars Journal*, vol. 30, no. 10, pp. 947–954, 1960.

[9] C. Watkins, "Learning from delayed rewards," 01 1989.

[10] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[11] H. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), vol. 23, Curran Associates, Inc., 2010.

[12] K. Zhang, Z. Yang, and T. Baar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," 2021.

[13] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic traffic simulation using sumo," in *The 21st IEEE International Conference on Intelligent Transportation Systems*, IEEE, 2018.

[14] J. I. Hoffman, *Biostatistics for Medical and Biomedical Practitioners*. 2015.

[15] "Central limit theorem." `https://www.britannica.com/science/probability-theory/The-central-limit-theorem`. [Online; accessed 23-11-2020].

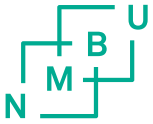
[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[17] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[18] M. P. Perrone, H. Khan, C. Kim, A. Kyrillidis, J. Quinn, and V. Salapura, "Optimal mini-batch size selection for fast gradient descent," 2019.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[20] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2019.




**Norges miljø- og biovitenskapelige universitet**
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway